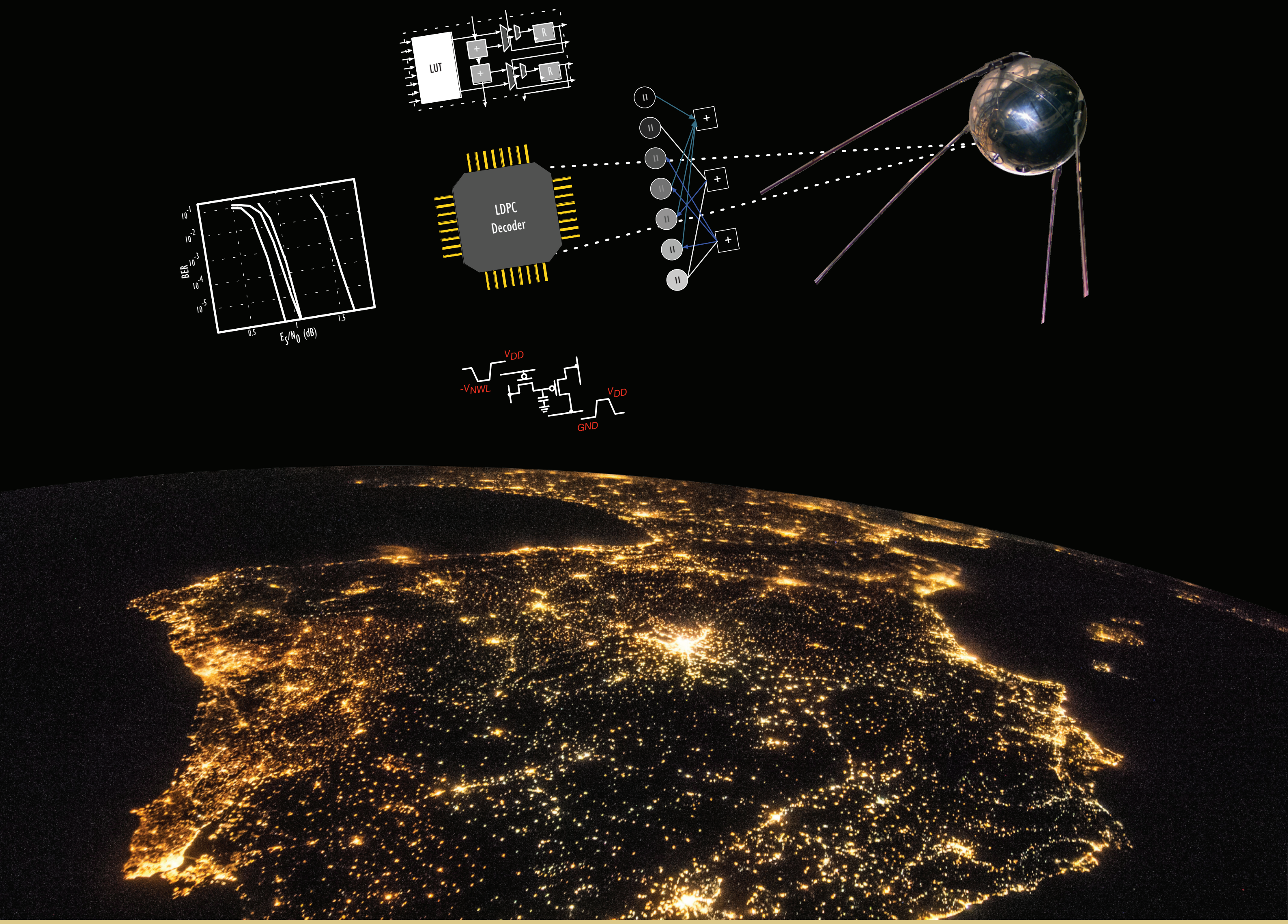


João Maria Duarte Andrade

Design Space Exploration of LDPC Decoders on Programmable and Reconfigurable Architectures

Tese de Doutoramento em Engenharia Electrotécnica e de Computadores, ramo de especialização em Telecomunicações, orientada pelo Professor Doutor Vitor Manuel Mendes Silva e Professor Doutor Gabriel Falcão Paiva Fernandes, e apresentada ao Departamento de Engenharia Electrotécnica e de Computadores da Faculdade de Ciências e Tecnologia da Universidade de Coimbra

Setembro de 2015

· U C ·

Universidade de Coimbra

Background of cover image source: *Earth Observatory*, NASA.

· U C ·

UNIVERSIDADE DE COIMBRA

FACULDADE DE CIÊNCIAS E TECNOLOGIA

DEPARTAMENTO DE ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES

Design Space Exploration of LDPC Decoders on Programmable and Reconfigurable Architectures

João Maria Duarte Andrade

(Mestre)

Dissertação para a obtenção do Grau de Doutor em

Engenharia Electrotécnica e de Computadores

Tese de Doutoramento em Engenharia Electrotécnica e de Computadores, ramo de especialização em Telecomunicações, orientada pelo Doutor Vitor Manuel Mendes da Silva e pelo Doutor Gabriel Falcão Paiva Fernandes, e apresentada ao Departamento de Engenharia Electrotécnica e de Computadores da Faculdade de Ciências e Tecnologia da Universidade de Coimbra.

Coimbra

Setembro 2015

João Maria Duarte Andrade: *Design Space Exploration of LDPC Decoders on Programmable and Reconfigurable Architectures*, Tese de Doutoramento em Engenharia Electrotécnica e de Computadores, ramo de especialização em Telecomunicações, Faculdade de Ciências e Tecnologia, Universidade de Coimbra © Setembro 2015

This Thesis was supported by Instituto de Telecomunicações under the grants PEst-OE/EEI/LA0008/2011, PEst-OE/EEI/LA0008/2013 and UID/EEA/50008/2013, and by Fundação para a Ciência e Tecnologia doctoral scholarship grant SFRH/BD/78238/2011 under the QREN - POPH - Type 4.1 - Advanced Formation programmes financed by the Fundo Social Europeu (FSE) and the national funding of MEC.

Acknowledgments

Aos meus orientadores, Professor Vitor Silva e Professor Gabriel Falcão, tenho a agradecer toda a ajuda, apoio e discussões que melhoraram o meu trabalho. Tenho de lhes reconhecer a disponibilidade em rever e melhorar as minhas contribuições científicas. Graças ao Vitor pude contornar sempre mais uma parede quando nelas embatia e manter em vista o caminho a percorrer. Devo ao encorajamento do Gabriel várias viagens Mundo fora, e as estadias no EPFL e na Xilinx Research Labs.

Agradeço ao Professor Marco Gomes, bem como ao João Amaro, ao Sinédoque, e colegas de laboratório, a ajuda prestada, procrastinação activa e sanidade proporcionadas. Aos investigadores do INESC-ID, Frederico Pratas, Professor Pedro Tomás e Professor Leonel Sousa, agradeço a hospitalidade demonstrada e colaborações realizadas. Ao Filipe Silva devo a disponibilidade do seu sofá no 16ème, ao João Nunes uma desculpa para voltar a Verbier e ao Carlos Oliveira o regresso à minha *alma mater* desportiva.

I thank David Novo, Nithin George, Kimon Karras and Professor Joseph R. Cavallaro, for their kindness, support, influence and academic collaboration. To Chantal Schneeberger I owe "mille mercis" for her effortless way to integrate people at LAP, making my life much easier. I would like to acknowledge Professor Paolo Ienne and Michaela Blott for hosting me at LAP, EPFL and at Xilinx Research Labs. Professor Shinichi Yamagiwa has granted access to the GPU-cluster without which simulations would still be running now.

Sem o apoio incondicional da minha família não seria possível ter chegado aqui. Aos meus Pais, Dá e Jasa, agradeço o permanente incentivo para aprender mais e maximizar o meu capital humano, bem como todo o apoio e incessante comunicação no meu ano fora. À Nanã e ao JotaPê, tenho a agradecer os magníficos momentos que me proporcionaram com o João Pequenino e a Maria Rita. Agradeço também à família Beldroega toda a ajuda prestada e cuidados dispensados, em especial à Margarida na minha ausência.

Last, but not the least, estou eternamente endividado para com a Margarida, o meu maior pilar de sanidade e amor. Dedicar-lhe estas poucas linhas é o mínimo, embora parca recompensa, para alguém sem cuja presença teria sido impossível concluir o que se iniciou há já longo tempo.

A todos, muito obrigado,

Aos meus Avós, Luísa e João, Inha e João Maria,
E, à Margarida,

Everyday life is like programming, I guess.
If you love something you can put beauty into it.

- Donald Knuth

Abstract

Low-density parity-check (LDPC) codes are capacity-approaching linear block codes widely employed for digital communication systems and storage. However, the realization of LDPC decoders is a very challenging process due to the numerical complexity associated with binary, and especially, with non-binary LDPC codes. Whereas *very large scale integration* (VLSI) technology provides the necessary means to allow the realization of efficient LDPC decoders that meet both low latencies and high decoding throughputs, the development process behind *application-specific integrated circuit* (ASIC) and *field-programmable gate array* (FPGA) decoders is error-prone, protracted and is an endeavor captured by low-level micro-architecture and silicon details that pose high *non-recurring engineering* (NRE) costs.

In this Thesis, we explore efficient ways to overcome the challenges associated with the development of binary and non-binary LDPC decoders on both programmable and reconfigurable hardware. We propose methodologies that leverage on the immense computational power of multicore *graphics processing unit* (GPU) architectures applied to binary and non-binary LDPC decoders, not only for achieving the very high data rates required for nowadays communications, but also for very fast Monte Carlo *bit error rate* (BER) simulation, essential for the study of new LDPC codes.

Having exploited the potential of parallel computing on programmable hardware and identified its shortcomings, we extend our proposed methodology to reconfigurable hardware. The developed FPGA-decoders explore different *high-level synthesis* (HLS) programming models, based on dataflow, loop-annotated and wide-pipeline architectures. From the performance analysis of these accelerators, we identify the key guidelines to the design of efficient LDPC decoders under each approach. Finally, we propose algorithm- and silicon-level procedures to boost the LDPC decoders energy efficiency. Namely, we propose *gear-shift* techniques, and incorporation of unreliable memory storage along with BER degradation mitigation strategies.

Keywords

LDPC codes, Galois field, Reconfigurable computing, Parallel computing, High-level synthesis, Unreliable memory storage

Resumo

Os *códigos definidos por matrizes de teste de paridade esparsas* (LDPC) são bastante poderosos em sistemas de comunicação digital e armazenamento de dados, por operarem quase à capacidade do canal. No entanto, a realização de descodificadores LDPC é um processo desafiante devido à complexidade associada aos códigos LDPC binários e, em particular, aos não-binários. Apesar de a *tecnologia de integração em larga escala* (VLSI) ter capacidade para a realização de descodificadores LDPC que cumpram a baixa latência e o elevado ritmo de transmissão de dados, os processos de desenvolvimento em *circuitos integrados de aplicação específica* (ASIC) ou em *circuitos lógicos programáveis* (FPGA) são morosos, conduzem facilmente a erros e são pautados por detalhes minuciosos ao nível do silício e da micro-arquitectura que elevam os custos não *recorrentes de engenharia* (NRE).

Esta tese aborda estratégias para superar os desafios inerentes ao desenvolvimento de descodificadores binários e não-binários em arquitecturas programáveis e reconfiguráveis. Deste modo, são propostas metodologias que exploram a imensa capacidade computacional de arquitecturas *multicore*, como *processadores gráficos* (GPUs), aplicadas a descodificadores binários e não-binários, não só atingindo os elevados ritmos de dados necessários aos sistemas de comunicação actuais, mas também permitindo rápida simulação de Monte Carlo para a caracterização da *taxa de erros* (BER), essencial para o estudo de novos códigos e algoritmos.

A metodologia proposta para arquitecturas programáveis é extendida a arquitecturas reconfiguráveis. Os descodificadores FPGA desenvolvidos tiram partido de síntese de alto-nível (HLS), baseada em modelos *dataflow*, *loop-annotated* e *wide-pipeline*. Através da análise da performance obtida em cada abordagem, propõem-se linhas orientadoras para o desenvolvimento de descodificadores de elevados desempenhos. Finalmente, são propostos métodos ao nível algorítmico e do silício de melhoria da eficiência energética dos descodificadores propostos. As técnicas desenvolvidos utilizam técnicas de *gear-shift* e de armazenamento de dados em memórias não-fiáveis, para as quais são introduzidas estratégias de diminuição da degradação de BER.

Palavras Chave

Códigos LDPC, Campos de Galois, Computação reconfigurável, Computação paralela, Síntese de alto-nível, Armazenamento em memória não-fiável

Contents

Page

List of Figures

List of Tables

List of Algorithms

List of Symbols

Galois Field Notation

α	Primitive element of GF(q), root of the primitive polynomial $p(x)$
$p(x)$	Primitive polynomial of GF(q)
q	Dimension of the Galois field
2^m	Dimension of the binary extension field

LDPC Code Notation

c_j	j-th element (bit or symbol) of codeword $\mathbf{c}$
d_c	CN degree
d_v	VN degree
$\mathbf{G}$	LDPC code generator matrix
$\mathbf{H}$	LDPC code parity-check matrix
$\mathbf{F}$	LDPC code protograph matrix
$\mathbf{I_K}$	$K \times K$ identity matrix
h_{ij}	Element h (in $\mathbf{H}$) in row i and column j
hf_{ij}	Element hf (in $\mathbf{H_f}$) in row i and column j
$\mathbf{P}$	Systematic LDPC code parity sub-matrix
$\mathbf{I}$	Systematic LDPC code information sub-matrix
K	Number of information symbols in the LDPC code
M	Number of parity-check restrictions in the LDPC code
M_f	Number of rows in the LDPC code protograph
N	Number of symbols in the LDPC code
N_f	Number of columns in the LDPC code protograph
$M(n)$	Set of CNs, with cardinality d_c, connected to VN$_n$ (over GF(2))
$N(m)$	Set of VNs, with cardinality d_v, connected to CN$_m$ (over GF(2))

$C(v)$	Set of CNs, with cardinality d_c, connected to VN$_v$ (over GF(2^m))
$V(c)$	Set of VNs, with cardinality d_v, connected to CN$_c$ (over GF(2^m))
R	LDPC code information rate
r_f	LDPC regularity factor
z_f	QC-LDPC expansion factor

Message-Passing Algorithm Notation

m	CN index in GF(2)
n	VN index in GF(2)
c	CN index in GF(q)
v	VN index in GF(q)
p_n	*A-priori* probability of symbol n
$q_{nm}^{(i)}$	Probability message exchanged between VN n and CN m, at iteration i
$r_{mn}^{(i)}$	Probability message exchanged between CN m and VN n, at iteration i
$Q_n^{(i)}$	*A-posteriori* probability of symbol n, at iteration i
α_{nm}	Sign of $L(q_{nm})$
β_{nm}	Magnitude of $L(q_{nm})$
$\mathbf{m}_v(x)$	*A-priori pmf* of symbol v
$\mathbf{m}_{cv}^{(i)}(x)$	*Pmf* exchanged between CN c and VN v, at iteration i
$\mathbf{m}_{cv}^{(i)}(z)$	Fourier-domain *pmf* exchanged between CN c and VN v, at iteration i
$\mathbf{m}_{vc}^{(i)}(x)$	*Pmf* exchanged between VN v and CN c, at iteration i
$\mathbf{m}_{vc}^{(i)}(z)$	Fourier-domain *pmf* exchanged between VN v and CN c, at iteration i
$\mathbf{m}^{*}{}_{v}^{(i)}(x)$	*A-posteriori pmf* of symbol v, at iteration i
$\mathbf{M}_{cv}^{(i)}(x)$	LLR vector exchanged between CN c and VN v, at iteration i
$\mathbf{M}_{vc}^{(i)}(x)$	LLR vector exchanged between VN v and CN c, at iteration i
$\mathbf{M}^{*}{}_{v}^{(i)}(x)$	*A-posteriori* LLR vector of symbol v, at iteration i

Mathematical Operators

$p(\cdot)$	Probability operator
$L(\cdot)$	Log-likelihood ratio operator
$\boxplus$	Boxplus-sum operator

$F(\cdot)$	Fourier transform
$F^{-1}(\cdot)$	Inverse Fourier transform
W_N^n	Twiddle factors of the Fourier transform
$W(\cdot)$	Walsh-Hadamard transform

Transmission System Notation

e_n	Error symbol n introduced by the communication channel
$\mathbf{r}$	Received codeword
$\mathbf{s}$	Syndrome vector
$\mathbf{u}$	An information containing message
$\mathbf{v}$	A codeword corresponding to message $\mathbf{u}$
x_n	Modulated symbol n
y_n	Modulated symbol n transmitted over the communication channel
P_{FU}	Power of the LDPC decoder*functional units*
P_{Mem}	Power of the LDPC decoder *memory*
P_c	Power of the LDPC decoder *channel memory*
P_m	Power of the LDPC decoder *messages memory*
P_{s0}	Probability of bit-cell failure under the stuck-at 0 channel model
$\mathbf{e}$	Error pattern introduced in the transmitted codeword
$Qx.y$	Fixed-point representation with $x - y$ sign and magnitude bits and y decimal bits
V_{DD}	Transistor supply voltage

List of Acronyms

10gigE	10 gigabit ethernet
2C	Two's complement data representation
AGU	Address generation unit
ALM	Adaptive logic module
ALU	Arithmetic and logic unit
ALUT	Adaptive LUT
API	Application programming interface
APU	AMD accelerated processing unit
ASIC	Application-specific integrated circuit
AVX	Advanced vector extensions
AWGN	Additive white Gaussian noise
BCH	Bose-Chaudhuri-Hocqueghem
BER	Bit error rate
BP	Belief propagation
BpC	Block-per-codeword
BpN	Block-per-node
BPSK	Binary phase-shift keying
BRAM	Block RAM
BSC	Binary symmetric channel
CAD	Computer-aided design
CCS	Compressed column storage
CDF	Cumulative distribution function

CLB	Configurable logic block
CMMB	China multimedia mobile broadcasting
CMOS	Complementary metal–oxide–semiconductor
CN	Check node
CpC	Core-per-codeword
CPU	Central processing unit
CRS	Compressed row storage
CSC	Compressed sparse column
CSR	Compressed sparse row
CTM	Close to the metal
CU	Compute unit
CUDA	Compute Unified Device Architecture
CUFFT	CUDA FFT
cuRAND	CuRAND library
DDG	Data-dependency graph
DE	Density evolution
DFE	Dataflow engine
DFG	Dataflow graph
DFT	Discrete Fourier transform
DIT	Decimation in time
DMA	Direct memory access
DRAM	Dynamic RAM
DRT	Data retention time
DSL	Domain-specific language
DSP	Digital signal processor
DVB 2	2nd generation DVB
DVB-C2	DVB - cable 2nd gen.
DVB-RCS2	DVB - receive channel satellite 2nd gen.
DVB-S2	DVB - satellite 2nd gen.

DVB-T2	DVB - terrestrial 2nd gen.
DVFS	Dynamic voltage and frequency scaling
ECC	Error-correcting code
eDRAM	Embedded dynamic RAM
EMS	Extended min-sum
EXIT	Extrinsic information transfer chart
FEC	Forward error correction
FF	Flip-flop
FFT	Fast Fourier transform
FFT-SPA	FFT sum-product algorithm
FFTW	Fastest Fourier transform in the West
FIFO	First-in first-out
FLOPs	Floating-point operations per second (usually GFLOPs or TFLOPs)
FPGA	Field-programmable gate array
FSM	Finite-state machine
FU	Functional unit
FWHT	Fast Walsh-Hadamard transform
GF(2)	Binary field
GF(2^m)	Binary extension field
GF(q)	Galois field of dimension q
GPGPU	General-purpose GPU
GPU	Graphics processing unit
HARQ	Hybrid automatic repeat request
HDL	Hardware description language
HLS	High-level synthesis
HPC	High performance computing
HSPA	High speed packet access
IB	InfiniBand (computer-networking comm. standard)
IDB	Improved differential binary algorithm

iFFT	Inverse fast Fourier transform
II	Initiation interval
i.i.d.	Independent and identically distributed
ILP	Instruction level parallelism
IN	Information node
IP	Intellectual property (used in the context of hardware cores)
IPC	Iterative parity-check
IpC	Instructions per clock
IR	Intermediate representation
IRRWBF	Impl.-efficient reliability ratio-based weighted bit-flipping alg.
ISA	Instruction set architecture
LDPC	Low-density parity-check
LDPC-IRA	LDPC Irregular-Repeat-Accumulate
LE	Logic element
LFSPA	Log-Fourier SPA
LLC	Last level cache
LLR	Log-likelihood ratio
LLRV	Log-likelihood ratio vector
LLVM	LLVM compiler infrastructure
LPFSPA	Log permutation FFT-SPA
LSPA	Logarithmic sum-product algorithm
LTE	Long term evolution
LUT	Lookup-table
MDA	Mixed-domain algorithm
MFU	Macro-FU
MIG	Memory interface generator
MIMD	Multiple-instruction multiple-data
MIMO	Multiple-input multiple-output
MPI	Message-passing interface

MSA	Min-sum algorithm
MSB	Most significant bit
NMSA	Normalized min-sum algorithm
NRE	Non-recurring engineering
OpenACC	Open Accelerators
OMSA	Offset min-sum algorithm
OpenCL	Open Computing Language
OpenMP	Open Multi-Processing
OS	Operating system
OTN	Optical transport network
PAL	Programmable array logic
PDF	Probability density function
PG	Progressive edge growth
PLB	Processor local bus
PLC	Power-line communication
PLRA	Parity likelihood ratio algorithm
pmf	Probability mass function
PN	Parity node
PpE	Pixel-per-edge
PRNG	Parallel random number generator
PW	Piecewise
PWM	Pulse-width modulation
QC-LDPC	Quasi-cyclic LDPC
QKD	Quantum-key distribution
QoS	Quality of service
RA	Repeat-accumulate
RAM	Random-access memory
RGU	Request generator unit
ROI	Region of interest

ROM	Read-only memory
RS	Reed-Solomon
RTL	Register-transfer level
SAC	Stuck-at channel
SAZC	Stuck-at 0 channel
SCC	Single-chip cloud computer
SCMSA	Self-corrected min-sum algorithm
SDK	Software development kit
sign-mag	Sign-magnitude data representation
SIMD	Single-instruction multiple-data
SIMT	Single-instruction multiple-thread
SLiC	Simple Live CPU
SM	Stream multiprocessor
SMS	Swing modulo scheduling
NMSA	Normalized min-sum algorithm
SNR	Signal-to-noise ratio
SoC	System on a chip
SOpenCL	Silicon-to-OpenCL
SP	Scalar processors
SPA	Sum-product algorithm
SPE	Synergistic processing element
SPIR	Standard Portable Intermediate Representation
SPMD	Single-program multiple-data
SRAM	Static RAM
SSE	Streaming SIMD extensions
SYCL	C++ programming model for OpenCL
TBB	Thread building blocks
Tcl	Tool command language
TDMP	Turbo-decoding message-passing schedule

TDP	Thermal dissipation power
TP	Thread processor
TpC	Thread-per-codeword
TpE	Thread-per-edge
TPMP	Two-phased message-passing schedule
TpN	Thread-per-node
TpNpC	Thread-per-node-per-codeword
VHDL	VHSIC hardware description language
VLIW	Very long instruction word
VLSI	Very large scale integration
VN	Variable node
WAR	Write-after-read
WebCL	Web computing language
WHT	Walsh-Hadamard transform
Wi-Fi	Wi-Fi
WiMAX	Worldwide interoperability for microwave access
WPAN	Wireless personal area network

List of Listings

1 Introduction

Contents

1.1 Motivation

Low-density parity-check (LDPC) codes are a class of very powerful *error-correcting code*s (ECCs) codes. They were first introduced in the early sixties[20,21], but failed to capture the attention of the scientific community. Despite some notable exceptions[22–24], this has been mainly attributed to the lack of computational power available to realize a practical LDPC decoder. Resurfacing in the early nineties, they appeared as patent-expired capacity-approaching codes[25–29] in direct competition with Turbo codes[30]. They have enjoyed a high popularity ever since, as attested by the number of standards that have adopted them in their *forward error correction* (FEC) systems: NASA deep space communication[31]; IEEE 802.3an (10gigE), 802.11n (Wi-Fi), 802.15 (WPAN), 802.16 (WiMAX)[32–35]; the ETSI 2$^{\text{nd}}$ generation digital video broadcasting (DVB 2) for satellite (DVB-S2), terrestrial (DVB-T2), cable (DVB-C2) and return channel via stellite (DVB-RCS2)[36–39]; the ITU-T G.9960 for home network (G.hn) *power-line communication* (PLC), G.709 for the *optical transport network* (OTN)[40,41]; and the 3GPP LTE (4G)[42]. Furthermore, ongoing discussions regarding their use as a suitable coding scheme for 5G communication are in place, such as in WirelessHD and Wireless Gigabit Alliance[43–45], both in their block code construction, only now over *Galois fields of dimension q* (GF(q))[46,47] or in their spatially-coupled construction[48–50].

The resurfacing of LDPC codes is intricately connected with the tremendous improvements of the computational power that, driven by the exponential growth in the number of transistor per chip due to *very large scale integration* (VLSI) technology[51,52], allowed practical LDPC decoder implementations to capacity-approaching performance at very high decoding throughputs and moderately low latency. A vast majority of the former is performed using *application-specific integrated circuit* (ASIC) technology, whose development incurs in high *non-recurring engineering* (NRE) costs, largely due to protracted and error-prone *register-transfer level* (RTL)-based design flows. Given that most progresses in LDPC codes ultimately take into consideration a practical decoder implementation, we have assisted to many advances in LDPC decoding techniques, code construction and decoding architecture that help reduce the overbearing costs of ASIC-based decoders. Nonetheless, as complexity of design continues to grow exponentially, alternative design methods should be procured as alternative computing paradigms thrived in the very time frame of LDPC codes advance.

Whereas the bulk of the growth in computational power up until the early noughties has been performed riding the scaling of the clock frequency of operation[53], power dissipation has effectively put an end to this trend. Thus, to overcome the power wall, single-core processors turned multicore, with several cores integrated onto the same processor die—dual-core soon lead to quad- and six-core, with *high performance computing* (HPC) cores going as far as 72-core—with multithreaded execution support and vector extensions providing *single-instruction multiple-data* (SIMD) execution model instructions to previously all-exclusive *multiple-instruction multiple-data* (MIMD) architectures. These functionalities provided *central processing unit* (CPU) architectures with high parallel computing capabilities, as a mean to prolong the growth in computational power, and also to address an ever growing gap between memory bandwidth and processing capabilities, the so-called memory wall[54]. Almost simultaneously, *graphics processing unit*s (GPUs) had grown into highly specialized graphics-oriented processors, with the unification of the pixel shader with the texture shader, along with the definition of more sophisticated instructions sets coupled data parallel programming models based on high-level languages, introducing a new class of parallel processors to the general-purpose realm[55]. In fact, current generations of x86 CPUs evolved to integrate a GPU core onto the same die, leading the way into heterogeneity of the processors in general-purpose computer systems.

Nonetheless, while this new breed of parallel architectures, providing TFLOPs of arithmetic power either through the execution of vector extensions, or by spawning thousands of concurrent computing threads, is able to compete for similar orders of magnitude of decoding throughput of ASIC technology, it neither addresses the low latency required for most standards, as typical techniques trade low latency for high throughput, nor the required low power for most devices that actually incorporate a FEC system. As a consequence, and similar to this tradeoff in LDPC decoding, we have assisted to the rise of *field-programmable gate array*s (FPGAs) from reconfigurable substrates providing "glue-logic" functionality, to a technology targeted for when high computing performance also requires low power[56,57]. Furthermore, as *high-level synthesis* (HLS) progressed based on data parallel programming models, in their turn based on C/C++, the NRE required to develop an FPGA-based processor has greatly diminished. All in all, these architectures provide distinct features to researchers and must be appropriately exploited for the role that suits them best.

The development of LDPC decoders which are appropriately tuned to the constraints of today's digital communication systems follows a design flow where different types of computer architectures are more suited to providing results at certain phases than others. In particular, the best architecture for rapid validation of a new class of LDPC code or LDPC decoding algorithm can have significantly different traits than those required for the rapid empirical evaluation of the *bit error rate* (BER) behavior and coding gain. Naturally, the final architecture for field deployment of the LDPC decoder will borrow traits from each computer architecture utilized through the design process, in a flow that aims at gathering the pros from each phase while leaving out the cons.

1.2 Objectives

The work carried out in this Thesis aims at providing insight regarding a number of key issues regarding the complex case of LDPC decoding on multiple, and disjoint, computer architectures. For the one, not only do the LDPC code constraints, whether driven by a standard or a service requirement, motivate a number of design decisions when sweeping the possible design solutions of the decoder architecture, but also the opposite is true. Certain decoding architectural parameter decisions are tightly coupled to the LDPC decoding operation design space in the way they influence its sweep. For the other, nowadays we are confronted with a multitude of programming models, computer architectures and computing substrates and in the absence of a universal methodology to narrow down the design space exploration to what are the solutions of interest, we propose our own methodology to proceed within such a rich and variate design space concentrating our efforts in the design features that pertain to the goal of maximizing the performance and efficiency of the developed LDPC decoders.

Consequently, taking into account 1) LDPC code service- or standard-driven constraints, 2) programming model suitable or available for the underlying computing substrate, 3) tradeoff between the attainable performance and NRE costs associated with a given programming model/computing substrate, 4) flexibility constraints imposed by each particular type of programming model/computing substrate, and 5) the feedback between all these decoding and computing design decisions, we propose a methodology to efficiently explore the design space of LDPC decoders on modern computing systems (c.f Figure 1.1).

Thus, having identified the overall objective to provide an efficient and rapid way to perform the required prototyping and validation of an LDPC decoder with particular characteristics or for certain operation points, across the aforementioned design space, we split this into a set of multiple objectives within this larger problem at hand. For LDPC decoders on programmable hardware it follows that the objectives are

i) providing a conceptual framework for the rapid validation of algorithmic traits concerning LDPC decoding, namely at the numerical level of the algorithm, both in data representation and implementation of the arithmetic instructions, and at the scheduling of the decoding itself;

ii) assessing how different computing systems within programmable architectures accelerate the BER Monte Carlo simulations in order to harness sufficient computa-

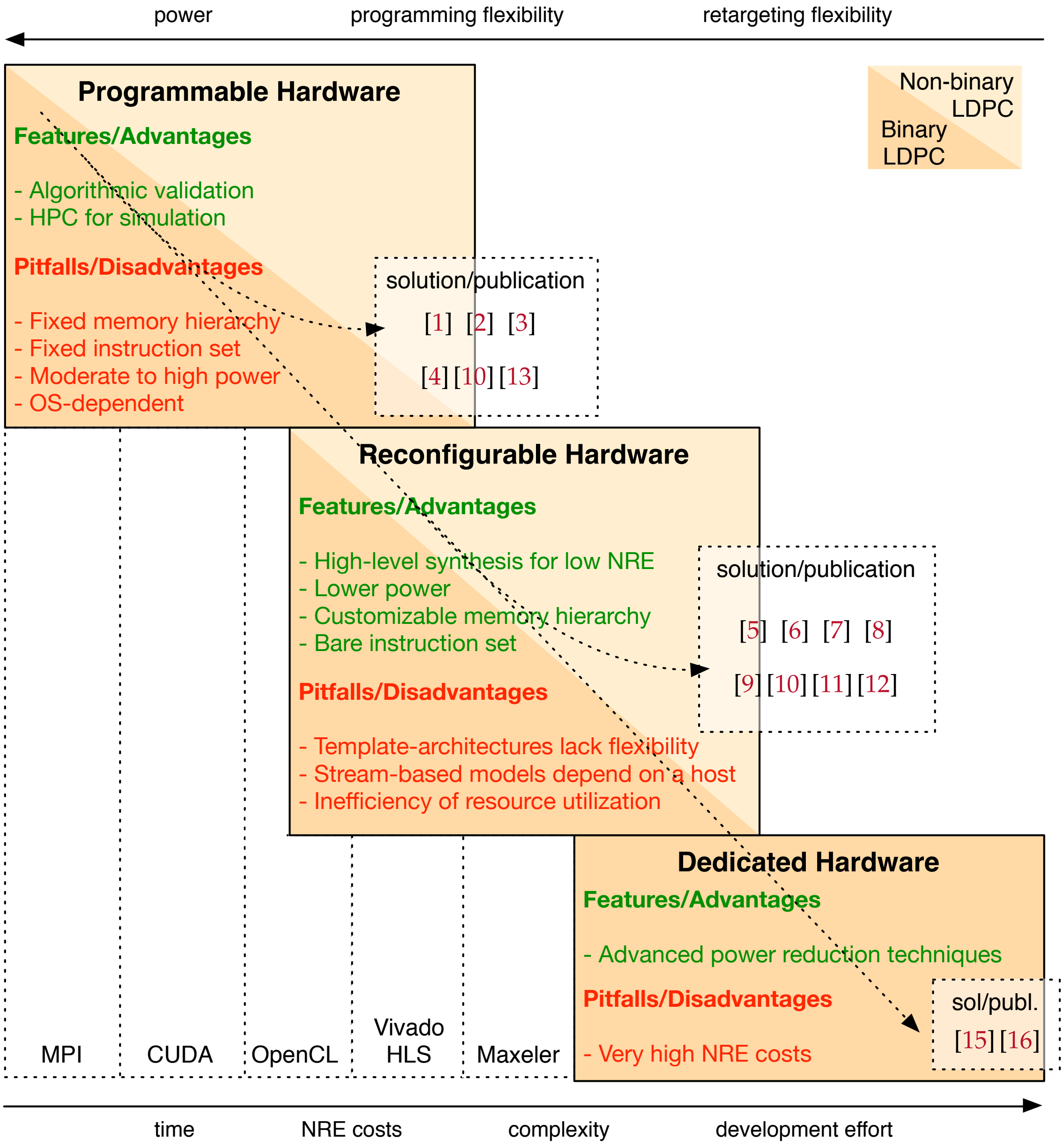


Figure 1.1: Design space exploration methodology followed for a multi-domain scenario composed of different computer architectures provided and mapped for distinct computing substrates.

tional power to be to visualize with statistical significance regions yielding very low BER, i.e. when operating for high *signal-to-noise ratio* (SNR) values;

iii) optimization of task- and data-parallelism levels to optimum levels within different programmable architectures on both homogeneous and on distributed systems;

iv) development of LDPC decoders that are compliant to the specified input constraints which are ready either for deployment or whose gained insights can be utilized for the ongoing design space exploration on another computing substrate.

Thus, partially based on the effects observed from being constrained by instruction-set-defined architectures, it follows that on reconfigurable hardware objectives can be summarized as

i) defining what programming models are most appropriate for each decoding solution in particular, considering the cases of binary and non-binary LDPC decoding;

ii) evaluation of the LDPC decoder performance for each explored HLS-based model, and how it influences/constrains the design space exploration;

iii) tuning of optimized memory hierarchies within the flexibility provided by the underlying programming model;

iv) efficient maximization of the reconfigurable logic at hand so as to provide maximum parallelism of computation without incurring into routing bottlenecks or clock frequency of operation reduction that impair the overall performance.

Finally, due to the evermore growing pressure with the realization of 100% reliable silicon, we explore how unreliable memory storage can be incorporated onto LDPC decoding designs, backed by the fact that 90% of the area is devoted to data storage. Hence, we highlight another objective as

i) study of gear-shift techniques applied to Min-sum-based decoders enabling power reductions at negligible BER degradation by reducing the number of iterations required to successfully decode a codeword;

ii) the study of the inherent error resilience of LDPC decoders under unreliable memory storage and algorithmic- and silicon-level strategies that can be employed to mitigate the BER degradation that comes with unreliable memory storage.

1.3 Main Contributions

The main contributions of this Thesis can be summarized in the following list.

i) Acceleration of Monte Carlo LDPC BER characterization on distributed computing systems, in particular, clusters of GPUs, exploring parallelism in *single-instruction multiple-thread* (SIMT) and *single-program multiple-data* (SPMD) computation models and incorporating scalability of the designed simulators; the developed simulators have been extensively used as the computation backbone with which results were obtained for numerous of the subsequent works herein detailed. In particular, this work was communicated in:

[1] J. Andrade, G. Falcao, V. Silva, S. Yamagiwa, and L. Sousa, *Encyclopedia of Computer Science and Technology*. Taylor & Francis, 2015, ch. Accelerating Conventional Processing Using GPU Clusters: LDPC Decoders.

[2] G. Falcao, J. Andrade, V. Silva, S. Yamagiwa, and Sousa, "Stressing the BER simulation of LDPC codes in the error floor region using GPU clusters," in *International Symposium on Wireless Communication Systems (ISWCS 2013)*, Aug 2013, pp. 1–5.

ii) Empirically showing that the self-correction technique allows for coding gains that are loosely independent of the SNR, as opposed to other correction methods applied to the *min-sum algorithm* (MSA). This work was communicated in:

[3] J. Andrade, G. Falcao, V. Silva, J. Barreto, N. Goncalves, and V. Savin, "Near-LSPA performance at MSA complexity," in *Communications (ICC), 2013 IEEE International Conference on*, June 2013, pp. 3281–3285.

iii) Efficient GPU-based decoders for *FFT sum-product algorithm* (FFT-SPA) decoding of high-order non-binary LDPC codes. This work was communicated in:

[4] J. Andrade, G. Falcao, V. Silva, and K. Kasai, "FFT-SPA Non-binary LDPC Decoding on GPU," in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, May 2013, pp. 5099–5103.

iv) Efficient LDPC decoders design using HLS approaches that allow faster design cycles than conventional RTL-based; however, as these approaches typically offer less performance than RTL, we propose methodologies for closing in on this performance gap. This work was communicated in:

[5] J. Andrade, G. Falcao, V. Silva, M. Owaida, N. Bellas, C.D. Antonopoulos, and P. Ienne, "Towards High-Throughput with Low-Effort Programming: From General-Purpose Manycores to Dedicated Circuits," in *DATE'13: Workshop on Designing for Embedded Parallel Computing Platforms: Architectures, Design Tools, and Applications (DEPCP)*, March 2013.

[6] J. Andrade, F. Pratas, G. Falcao, V. Silva, and L. Sousa, "Combining flexibility with low power: Dataflow and wide-pipeline LDPC decoding engines in the Gbit/s era," in *Application-specific Systems, Architectures and Processors (ASAP), 2014 IEEE 25th International Conference on*, June 2014, pp. 264–269.

[7] J. Andrade, G. Falcao, and V. Silva, "Flexible design of wide-pipeline-based WiMAX QC-LDPC decoder architectures on FPGAs using high-level synthesis," *Electronics Letters*, vol. 50, no. 11, pp. 839–840, 2014.

[8] J. Andrade, N. George, K. Karras, D. Novo, V. Silva, P. Ienne, and G. Falcao, "Fast Design Space Exploration Using Vivado HLS: Non-binary LDPC Decoders," in *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, May 2015, pp. 97–97.

[9] J. Andrade, N. George, K. Karras, D. Novo, V. Silva, P. Ienne, and G. Falcao, "From low-architectural expertise up to high-throughput non-binary ldpc decoders: Optimization guidelines using high-level synthesis," in *Field Programmable Logic and Applications (FPL), 2015 25th International Conference on*, Sept 2015, pp. 1–8.

[10] F. Pratas, J. Andrade, G. Falcao, V. Silva, and L. Sousa, "Open the Gates: Using High-level Synthesis towards programmable LDPC decoders on FPGAs," in *Global Conference on Signal and Information Processing (GlobalSIP), 2013 IEEE*, Dec 2013, pp. 1274–1277.

[11] J. Andrade, G. Falcao, V. Silva, and K. Kasai, "Flexible non-binary LDPC decoding on FPGAs," in *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*. IEEE, May 2014, pp. 1936–1940.

[12] M. Owaida, G. Falcao, J. Andrade, C. Antonopoulos, N. Bellas, M. Purnaprajna, D. Novo, G. Karakonstantis, A. Burg, and P. Ienne, "Enhancing Design Space Exploration by Extending CPU/GPU Specifications Onto FPGAs," *ACM Trans. Embed. Comput. Syst.*, vol. 14, no. 2, pp. 33:1–33:23, Feb. 2015.

v) Proposing scalable methods for the computation of the *fast Fourier transform* (FFT) and *fast Walsh-Hadamard transform* (FWHT) on SIMT-based architectures and FPGA wide-pipeline accelerators, which are critical to the decoding of non-binary LDPC without explicit computation in the Galois Field;. This work was communicated in:

[13] J. Andrade, G. Falcao, and V. Silva, "Optimized Fast Walsh–Hadamard Transform on GPUs for non-binary LDPC decoding," *Parallel Computing*, vol. 40, no. 9, pp. 449 – 453, 2014.

[14] J. Andrade, V. Silva, and G. Falcao, "From OpenCL to gates: The FFT," in *Global Conference on Signal and Information Processing (GlobalSIP), 2013 IEEE*, Dec 2013, pp. 1238–1241.

vi) Power reduction of existing LDPC decoder designs by (1) leveraging Min-Sum-based gear-shift decoders and (2) by incorporation of unreliable memory modules; proposing of a methodology to provide error mitigation for unreliable memories based on repair and *most significant bit* (MSB) protection. These works were communicated, respectively, in:

[15] J. Andrade, G. Falcao, and V. Silva, "Accelerating and Decelerating Min-Sum-based Gear-shift LDPC Decoders," in *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*, April 2015, pp. 5099–5103.

[16] J. Andrade, A. Vosoughi, G. Wang, G. Karakonstantis, A. Burg, G. Falcao, V. Silva, and J. Cavallaro, "On the performance of LDPC and turbo decoder architectures with unreliable memories," in *Signals, Systems and Computers, 2014 48th Asilomar Conference on*, Nov 2014, pp. 542–547.

[17] J. Mu, A. Vosoughi, J. Andrade, A. Balatsoukas-Stimming, G. Karakonstantis, A. Burg, G. Falcao, V. Silva, and J. Cavallaro, "The Impact of Faulty Memory Bit Cells on the Decoding of Spatially-Coupled LDPC Codes," in *Signals, Systems and Computers, 2015 49th Asilomar Conference on*, 2015.

1.4 Outline

The following parts of this Thesis are divided in six chapters. In Chapter 2, an overview is given to LDPC codes both in their binary (GF(2)) and non-binary (GF(q) and GF(2^m)) definition. Therein, the main message-passing decoding algorithms found in the literature are surveyed. This characterization has gathered the arguments for why we focus

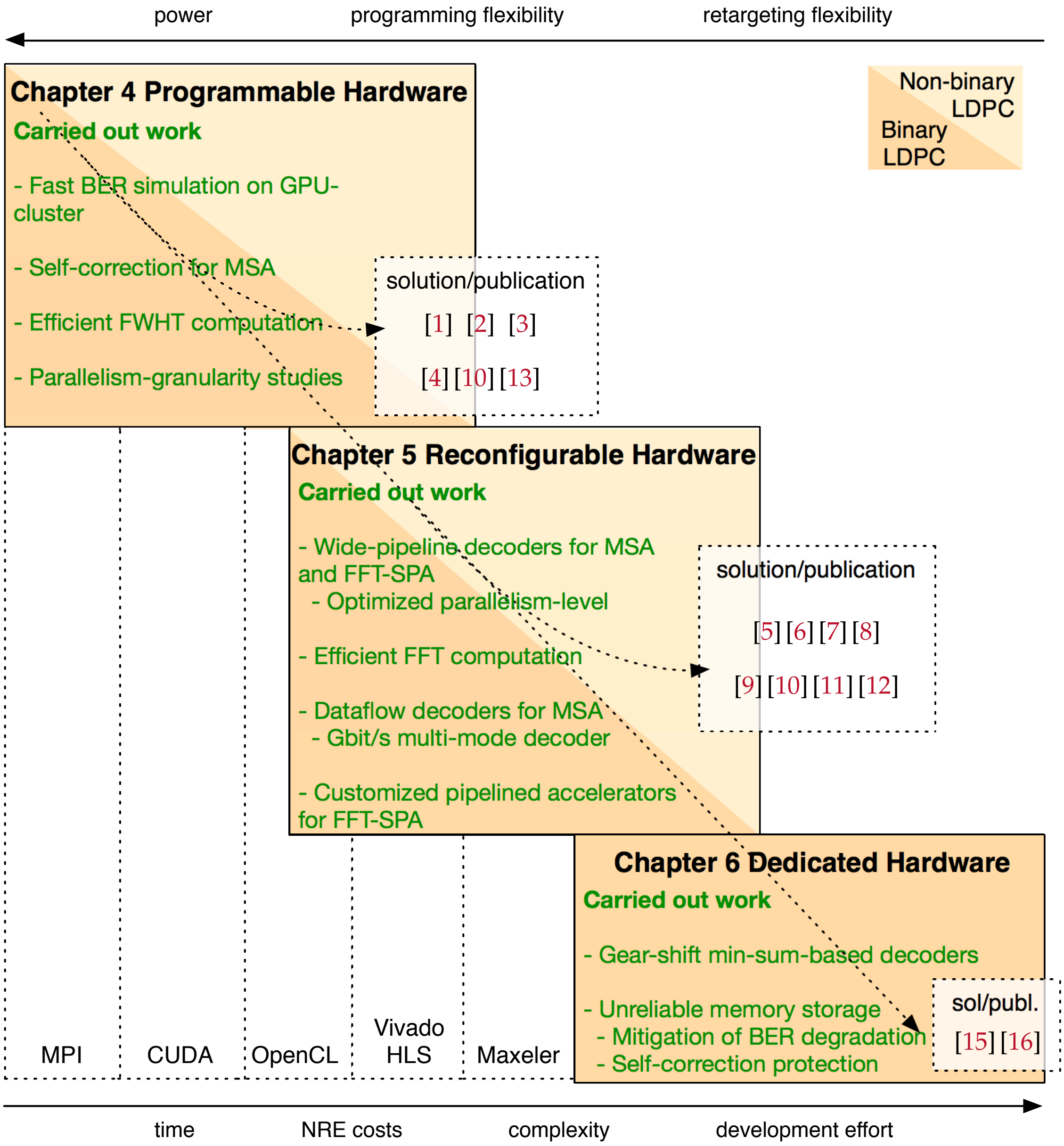

power
programming flexibility
retargeting flexibility
Non-binary LDPC
Binary LDPC
Chapter 4 Programmable Hardware
Carried out work
- Fast BER simulation on GPU-cluster
- Self-correction for MSA
- Efficient FWHT computation
- Parallelism-granularity studies
solution/publication
[1] [2] [3]
[4] [10] [13]
Chapter 5 Reconfigurable Hardware
Carried out work
- Wide-pipeline decoders for MSA and FFT-SPA
- Optimized parallelism-level
- Efficient FFT computation
- Dataflow decoders for MSA
- Gbit/s multi-mode decoder
- Customized pipelined accelerators for FFT-SPA
solution/publication
[5] [6] [7] [8]
[9] [10] [11] [12]
Chapter 6 Dedicated Hardware
Carried out work
- Gear-shift min-sum-based decoders
- Unreliable memory storage
- Mitigation of BER degradation
- Self-correction protection
sol/publ.
[15] [16]
MPI
CUDA
OpenCL
Vivado HLS
Maxeler
time
NRE costs
complexity
development effort

Figure 1.2: Thesis organization divided into three main chapters, for LDPC decoders on programmable and reconfigurable hardware, and for improving the power efficiency of LDPC decoders on dedicated hardware.

on certain algorithms for the realization of programmable and reconfigurable LDPC decoders. Chapter 3 surveys the state-of-the-art in programmable LDPC decoders, with a special focus on multicore GPU architectures, while also covering CPU decoders and the recently new field of HLS-based LDPC decoders for reconfigurable substrates. Furthermore, algorithmic- and silicon-level techniques to reduce the power drawn by LDPC decoders are also covered. Namely, an overview is given to gear-shifting techniques and to unreliable memory systems. Chapters 4, 5 and 6 represent the bulk of the developed work in a design space exploration flow that moves from programmable hardware LDPC decoders, to reconfigurable computing LDPC decoders and finally arrives at energy-efficiency concepts available only for dedicated LDPC decoders (c.f. Figure 1.2). Chapter 4 describes the work carried out for the design space exploration of LDPC decoders on programmable hardware using both vendor-specific and cross-platform data parallel programming models. Chapter 5 describes the work carried out for the design space exploration of LDPC decoders on reconfigurable substrates. In particular, the work within presented can be divided onto three different architectural approaches *i)* Java-based dataflow accelerators, *ii)* C/C++-based loop-annotated accelerators and *iii)* OpenCL-based wide-pipeline accelerators, assessing the performance of each approach against the other and with the decoders in Chapter 4. Chapter 6 is dedicated to two methodologies related with boosting the energy-efficiency of dedicated LDPC decoders, gear-shifting at the algorithmic-level, and incorporation of unreliable memory storage at the silicon-level along with BER degradation mitigation strategies proposed at the algorithmic- and silicon-level. Finally, Chapter 7 draws conclusions and sets directions for future work regarding the realization of efficient LDPC decoders.

2

LDPC Codes Fundamentals

Contents

In this chapter, the concepts behind *low-density parity-check* (LDPC) codes are introduced, with a special focus given to the message-passing algorithms involved in the decoding of LDPC codes, both in their *binary field* (GF(2)) and in their *binary extension field* (GF(2^m)) definition[20,47,58]. In 1948, Shannon lay the foundations for information theory when proving that information could be reliably transmitted across a noisy channel if the information transmission rate is lower than the capacity of said channel[29]. However, the mathematical tools to devise "perfect" codes were not provided, thus giving birth to the information theory field. Reliable transmission is achieved by employing *forward error correction* (FEC), whereupon redundancy is added to the transmitted information in some analytical form. Over the years, numerous coding schemes have been proposed—*Reed-Solomon* (RS), *Bose-Chaudhuri-Hocqueghem* (BCH), Golay, Turbo- and LDPC codes, among others—from which Turbo-codes and LDPC codes stand out due to enabling decoding thresholds that are capacity-approaching[59].

2.1 Linear Block Codes

The topology of a transmission system can be simplified as seen in Figure 2.1, which places a special focus on the FEC system within. In this scenarion, the transmitter **Tx** is bound to send a message **u** composed of K bits that is encoded by the encoder block before transmission through the channel. There, the encoder produces a codeword **v** according to some analytical method composed of N bits. **u** and **v** have a a one-to-one correspondence, and thus, where there are 2^K distinct messages there are also 2^k codewords. A block code $\mathbb{C}$ of length N and 2^K codewords is a linear (N, K) code iff its codewords form a K-dimensional subspace of the vector space of all the N-tuples over GF(2). Moreover, any linear combination of codewords is also a codeword.

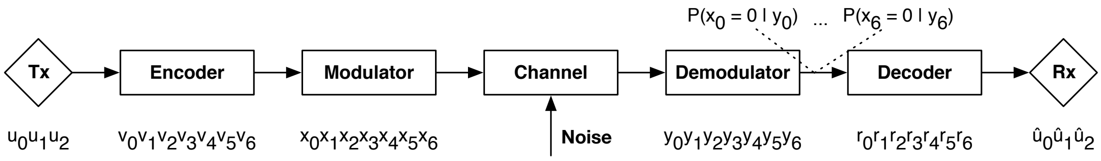


Figure 2.1: Digital transmission system with forward error correction.

2.1.1 Generator Matrix

Since $\mathbb{C}$ defines a K-dimensional subspace of a N-dimensional space, it is possible to find K linear independent codewords in $\mathbb{C}$ that define the remaining $2^K - K$ codewords as linear combinations of the former. Writing the K independent codewords as

$\mathbf{g}_0$, $\mathbf{g}_1, \cdots, \mathbf{g}_{K-1}$, and organizing them in the rows of a $[K \times N]$ matrix, we can build the generator matrix $\mathbf{G}$ as

$$\mathbf{G} = \begin{bmatrix} \mathbf{g}_0 \\ \mathbf{g}_1 \\ \vdots \\ \mathbf{g}_{K-1} \end{bmatrix} = \begin{bmatrix} g_{0,0} & g_{0,1} & \cdots & g_{0,N-1} \\ g_{1,0} & g_{1,1} & \cdots & g_{1,N-1} \\ \vdots & \vdots & \ddots & \vdots \\ g_{K-1,0} & g_{K-1,1} & \cdots & g_{K-1,N-1} \end{bmatrix}, \tag{2.1}$$

and it can be readily seen that the message $\mathbf{u}$ has a correspondent codeword $\mathbf{v}$ given by

$$\begin{aligned} \mathbf{v} &= \mathbf{u} \cdot \mathbf{G} \\ &= u_0\mathbf{g}_0 + u_1\mathbf{g}_1 + \cdots + u_{K-1}\mathbf{g}_{K-1}. \end{aligned} \tag{2.2}$$

The generator designation stands out in (2.2) as the rows of $\mathbf{G}$ generate any codeword.

For simplicity in the decoder at the receiver side, it is desirable that linear block codes be systematic, i.e., $\mathbf{u}$ should be contained in $\mathbf{v}$ in plain format. This way, the message $\mathbf{u}$ can be retrieved from slicing the codeword $\mathbf{v}$ at the correct bit positions. For instance, in (2.4), the $v_K, v_{K+1}, \cdots, v_{N-1}$ bits correspond to $\mathbf{u}$ and the remaining $N-K$ bits, $v_0, v_1, \cdots, v_{K-1}$ bits are designated as parity bits. Generator matrices of systematic linear block codes have the form

$$\mathbf{G} = \begin{bmatrix} \mathbf{g}_0 \\ \mathbf{g}_1 \\ \mathbf{g}_2 \\ \vdots \\ \mathbf{g}_{K-1} \end{bmatrix} = \begin{bmatrix} \mathbf{P} & \mathbf{I}_K \end{bmatrix} = \left[\begin{array}{cccc|ccccc} p_{0,0} & p_{0,1} & \cdots & p_{0,N-K-1} & 1 & 0 & 0 & \cdots & 0 \\ p_{1,0} & p_{1,1} & \cdots & p_{1,N-K-1} & 0 & 1 & 0 & \cdots & 0 \\ p_{2,0} & p_{2,1} & \cdots & p_{2,N-K-1} & 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ p_{K-1,0} & p_{K-1,1} & \cdots & p_{K-1,N-K-1} & 0 & 0 & 0 & \cdots & 1 \end{array}\right]. \tag{2.3}$$

Under the systematic generator matrix $\mathbf{G}$ we are able to write

$$\begin{aligned} v_{N-K+i} &= u_i \Leftarrow 0 \le i < K \\ v_j = u_0 p_{0,j} + u_1 p_{1,j} + \cdots &+ u_{K-1} p_{K-1,j} \Leftarrow 0 \le j < N-K, \end{aligned} \tag{2.4}$$

thus observing that the parity bits are obtained from the linear combination on which the message bits participate according to the $p_{i,j}$ values.

2.1.2 Parity-Check Matrix

Knowing that for any $[K \times N]$ generator matrix $\mathbf{G}$, in general all matrices with K independent rows, there exists a $[(N-K) \times N]$ matrix $\mathbf{H}$ such that any vector in the row space, i.e. the codeword set $\mathbb{C}$, is orthogonal to the rows of $\mathbf{H}$. Thus,

$$\mathbf{v} \times \mathbf{H}^\top = 0, \tag{2.5}$$

for all codewords $\mathbf{v} \in \mathbb{C}$. For a systematic generator matrix $\mathbf{G}$ the matrix $\mathbf{H}$ can be readily found

$$\mathbf{H} = \begin{bmatrix} \mathbf{I}_{N-K} & \mathbf{P}^\top \end{bmatrix}. \tag{2.6}$$

Using (2.5) yields

$$v_j + u_0 p_{0,j} + u_1 p_{1,j} + \cdots + u_{K-1} p_{K-1,j} = 0 \Leftarrow 0 \leq j < N-K, \tag{2.7}$$

which is the same as (2.4). Thus, the parity-check matrix $\mathbf{H}$ completely defines a linear block code.

Regularity The characteristics of the parity-check matrix $\mathbf{H}$ can be used to define the code against its regularity or irregularity. In particular, by defining d_c (2.8) and d_v (2.9) as the number of non-null elements per row and per column

$$d_c(i) = \sum_{j=0}^{N-1} h_{ij},\ i \in \{0, 1, \cdots, M-1\} \tag{2.8}$$

$$d_v(j) = \sum_{i=0}^{M-1} h_{ij},\ j \in \{0, 1, \cdots, N-1\}, \tag{2.9}$$

a (d_v, d_c)-code is said to be regular if $d_c(i){=}X, X{\in}\mathbb{Z},\ \forall i$ and $d_v(j){=}Y, Y{\in}\mathbb{Z},\ \forall j$. If, however, $d_c(i)$ or $d_v(j)$ take different values across different rows or columns, the code is said to be irregular. Usually, irregular codes combine multiple d_v degrees while attempting at keeping the d_c degree constant[59]. While regular codes are simpler in parity-check matrix structure than irregular ones, binary irregular codes usually possess better performance, and are thus, the predominant choice for standards using LDPC codes for their FEC systems. However, irregularity is employed in a structured fashion to keep the $\mathbf{H}$ layout simple. A concept used in *density evolution* (DE) is the one of degree polynomials which define the relative proportion of degrees in the Tanner graph, written as

$$\begin{aligned} p_{d_v}(x) &= \frac{1}{N} \sum_{i=0}^{N} x^{d_v(i)} \\ p_{d_c}(x) &= \frac{1}{M} \sum_{i=0}^{M} x^{d_c(i)}, \end{aligned} \tag{2.10}$$

with $p_{d_v}(x)$ the *variable node* (VN) degree polynomial and $p_{d_c}(x)$ the *check node* (CN) one. There are several LDPC code construction methods that commit different regularity and structure to a code $\mathbf{H}$ [59].

Progressive Edge Growth Codes *Progressive edge growth* (PG) is a construction method which balances the random distribution of the non-null elements in a regular (d_v, d_c)-

code in order to maximize its decoding performance. Within the **H** structure there is no evident pattern which can be exploited for a compact representation. These codes are widely available in the Encyclopedia of Sparse Graph codes[60], henceforth designated as Mackay codes.

LDPC-IRA Codes *LDPC Irregular-Repeat-Accumulate* (LDPC-IRA) codes, such as those in use in the ETSI 2nd generation standards (2^{nd} *generation DVB* (DVB 2)), are based on two principles. The first is of *repeat-accumulate* (RA), whereupon codes have systematic construction and the parity bits can be easily calculated given the double diagonal triangular inferior parity sub-matrix **P** structure. This zig-zag resembling structure allows for the following recursion to hold

$$\begin{aligned} c_{N-K} &= \sum_{j=0}^{N-K-1} h_{0j}c_j \\ c_{N-K+i} &= c_{N-K+i-1} + \sum_{j=0}^{N-K-1} h_{ij}c_j,\ i \in 1,\cdots,K-1, \end{aligned} \tag{2.11}$$

keeping the encoder design as simple as possible. The parity sub-matrix is also easily accounted using the RA principle. Finally, the information sub-matrix **I** is designed with a shifting column rule throughout its construction[61], whereupon a set of independent columns gets cyclically shifted downwards, with a shift value q, for a number of times $r_f - 1$ to generate r_f number of columns of the matrix. This way, the only required information to represent **H** is the set of independent columns, the shift value and the number of times a column is shifted.

Quasi-Cyclic LDPC Codes *Quasi-cyclic LDPC* (QC-LDPC) codes are constructed from a protograph matrix[62], also designated as base matrix. Essentially, a matrix **F**, with size (M_f, N_f), with elements defined over $\mathbb{R}$ will be expanded by a factor z_f, the expansion factor, by replacing each element with sub-matrices of size $(z_f \times z_f)$. When the element in the protograph is finite a cyclically shifted identity matrix, with size $(z_f \times z_f)$, is inserted. The number of positions shifted is given by $f_{ij} \mod z_f$. Infinity elements in the protograph are replaced by null sub-matrices. For instance, matrix **F** in (2.12) is expanded to matrix $\mathbf{H}_1$ (2.13) when an expansion factor of $z_f = 4$ is applied.

$$\mathbf{F} = \begin{bmatrix} 0 & 3 & \infty \end{bmatrix} \tag{2.12}$$

$$\mathbf{H}_1 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.13}$$

The advantage of using protograph-based codes, as defined by some standards[34,35,63], is that the code block length can be adjusted by changing z_f, as it can be readily seen that $N = z_f \times N_f$, and independently of the code block length, the structure of $\mathbf{H}$ is completely accounted for by $\mathbf{F}$ and z_f, allowing simplicity of representation, more often than not helped by the existence of a RA sub-matrix, or quasi-RA[64], in the protograph for simple encoding.

2.1.3 Coding Rate

Linear block codes add a certain amount of redundancy to an information message of a given length. The coding is defined as the ratio of information in the overall encoded word. Thus, it can be defined as $R = K/N$. For the particular case of regular (d_v, d_c)-LDPC codes, i.e. when d_v and d_c are constant across the whole graph, the coding rate can be written as $R = (d_c - d_v)/d_c$. Since $R < 1$, it stands out that $d_v < d_c$ for all linear codes, as $d_v = d_c(1 - R)$.

2.1.4 Syndrome and Error Detection

In Figure 2.1, the transmission channel input is a codeword $\mathbf{v}$, and the output is a N-length word $\mathbf{r}$ composed by the $\mathbf{v}$ and $\mathbf{e}$ quantifying the noise introduced by the channel, thus

$$\mathbf{e} = \mathbf{r} + \mathbf{v} \implies \mathbf{r} = \mathbf{v} + \mathbf{e}, \tag{2.14}$$

and replacing $\mathbf{r}$ in (2.5) produces

$$\begin{aligned}
\mathbf{s} &= \mathbf{r} \times \mathbf{H}^\top \\
\mathbf{s} &= \underbrace{\mathbf{v} \times \mathbf{H}^\top}_{0} + \mathbf{e} \times \mathbf{H}^\top \\
\mathbf{s} &= \mathbf{e} \times \mathbf{H}^\top.
\end{aligned} \tag{2.15}$$

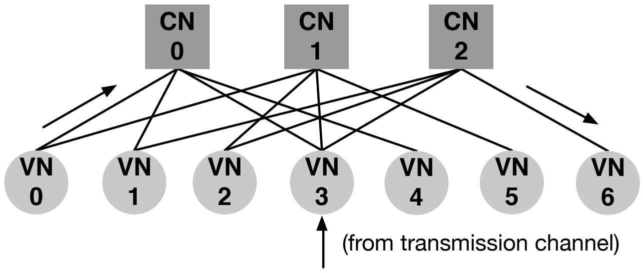
The vector $\mathbf{s}$ is designated the syndrome vector, and $\mathbf{s}$=0 iff $\mathbf{r}$ is a codeword and $\mathbf{s}\neq 0$ if $\mathbf{r}$ is not a codeword. It is also possible that a sustained error is not detected if $\mathbf{e}$ is a codeword, i.e. an undetected error occurs. However, design of robust codes takes into account this type of error and also aims at reducing such probability.

2.2 LDPC Codes over GF(2)

LDPC codes are a class of linear block code, characterized by their sparse parity-check matrices, which possess capacity-approaching characteristics[28,65–67]. First proposed by Gallager in the early 1960s[68,69], LDPC codes full potential was only realized thirty years later, with the exceptions of Zyablov *et al.*, Tanner and Wiberg[22,24,70–74], when in the early nineties, LDPC codes resurfaced through the works of Mackay and Neal[60,75–77].

2.2.1 Tanner Graph

The parity-check matrix **H** is the adjacency matrix to a graph representation of any linear block code that is designated as Tanner graph[72], a representation useful for graph definition of codes, but also for the depiction of the message-passing decoding algorithms employed in LDPC decoding. In their bipartite graph notation, there are two types of nodes which are connected by edges, 1) VNs correspond to the codeword bits, and thus, correspond also to columns of **H** , and 2) CNs correspond to the parity-check restrictions of the code, i.e. to rows of **H** . Furthermore, in the case of systematic LDPC codes, VNs can be further divided into *information node*s (INs), those corresponding to the bits of the **u** , and into *parity node*s (PNs), those corresponding to the parity bits. There is an edge connecting each type of node whenever the corresponding entry in **H** is a non-null element. The cardinality of edges connected to a node is designated as degree or weight, and the notation is d_v and d_c, respectively, for VNs and CNs. Also, the set $M(n)$ represents the CNs adjacent to VN n, while the set $N(m)$ represents the VNs adjacent to CN m.



$$\mathbf{H} = \begin{bmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{bmatrix}$$

Figure 2.2: LDPC code over GF(2) in matrix and Tanner graph representation. The message-passing procedure used by LDPC decoding algorithms is also shown.

2.3 LDPC Decoding Algorithms over GF(2)

The most prominent decoding algorithms for LDPC codes defined over GF(2) are overviewed herein. Among them, soft-decoding algorithms are the most prolific in the field. For the one they provide better decoding performance, also designated as coding gain, and also they are numerically less complex than hard-decoding methods[47]. Soft-decoding algorithms are characterized by operating on likelihood domains, instead of computing the most likely error vector through the bit state of the codeword bits, as hard-decoding methods do, an *a-posteriori* likelihood estimate is inferred from the channel received likelihoods and the Tanner graph induced constraints. Soft-decoding algorithms can be summarized as an organized series of steps.

i) **Initialization**: the demodulator computes the likelihood p_n associated with VN$_n$ and this information is broadcast to all VNs which initialize the messages $q_{nm}, \forall m$;

ii) **CN processing**: the CNs receive the q_{nm} messages from their adjacent VNs and compute r_{mn} messages according to the algorithm's CN update rule;

iii) **VN processing**: the VNs receive the r_{mn} messages from their adjacent CNs and compute new q_{nm} messages according to the VN update rule;

iv) **Hard decoding**: the VNs also compute Q_n messages from the r_{mn} messages received and produce a bit state according to this estimate;

v) **Repeat process**: go back to *i)* until a maximum number of iterations is reached or a valid codeword is produced.

2.3.1 Sum-Product Algorithm

The *sum-product algorithm* (SPA), also known as *belief propagation* (BP), was proposed by Gallager in his seminal work on LDPC codes[68,69]. It is a message-passing algorithm, in the sense that reliability messages are passed between VNs and CNs constrained by the Tanner graph of the LDPC code. The SPA is formalized in Algorithm 2.1.

The SPA is characterized by its heavy numerical complexity, thus there is a strong motivation to mitigate the effects of such complexity. Especially in the CN processing step which is the most complex of all, in effect, requiring the calculation of a convolution operation.

2.3.2 Logarithmic Sum-Product Algorithm

Due to the hardware realization of the SPA being complex, with the numerical nature of the algorithm consisting mainly of multiplications, precautions are required to ensure low-quantization errors. Moreover, as multiplication comes as an expensive operation to implement, a suitable domain change to the *log-likelihood ratio* (LLR) domain will replace all the multiplications with additions. By definition, the LLR operator, $L(\cdot)$, applied to a random binary variable u performs the following operation

$$L(u) = \log\left(\frac{p(u{=}0)}{p(u{=}1)}\right). \tag{2.21}$$

Rewriting the SPA to change its domain to LLR yields the *logarithmic sum-product algorithm* (LSPA), formalized in Algorithm 2.2.

Algorithm 2.1 Sum-product algorithm

```
Initialization: $q_{mn}^{(0)}(x{=}1) = p_n = p(v_n{=}1|y_n)$
while $\mathbf{v} \cdot \mathbf{H}^{\top} \neq 0$ or $i < \text{Iterations}_{\text{MAX}}$ do
  CN Processing:
      $r_{mn}^{(i+1)}(x{=}0) = \frac{1}{2} + \frac{1}{2} \prod_{n' \in N(m)\backslash n} \left(1 - 2q_{n'm}^{(i)}(x{=}0)\right)$   (2.16)
      $r_{mn}^{(i+1)}(x{=}1) = 1 - r_{mn}^{(i+1)}(x{=}0)$   (2.17)
  VN Processing:
      $q_{nm}^{(i+1)}(x{=}0){=}k_{nm}(1 - p_n) \prod_{m' \in M(n)\backslash m} r_{m'n}^{(i+1)}(x{=}0),\ k_{nm}{\Leftarrow}\sum_x q_{nm}^{(i+1)}(x{=}0) = 1$   (2.18)
      $q_{nm}^{(i+1)}(x{=}1){=}k_{nm}p_n \prod_{m' \in M(n)\backslash m} r_{m'n}^{(i+1)}(x{=}1),\ k_{nm}{\Leftarrow}\sum_x q_{nm}^{(i+1)}(x{=}1) = 1$   (2.19)
  a-posteriori decoding:
      $Q_n^{(i+1)}(x{=}0) = K_{nm}(1 - p_n) \prod_{m \in M(n)} r_{mn}^{(i+1)}(x{=}0),\ K_n{\Leftarrow}\sum_x Q_n^{(i+1)}(x{=}0) = 1$   (2.20)
  Hard-decoding: $\hat{c}_n{=}1{\Leftarrow}Q_n(1) > Q_n(0),\ \hat{c}_n{=}0{\Leftarrow}Q_n(1) < Q_n(0)$
end while
```

Algorithm 2.2 Logarithmic sum-product algorithm

```
Initialization: $L(v_n) = q_{mn}^{(0)} = \log \frac{P(v_n = 0|y_n)}{P(v_n = 1|y_n)} = \log \frac{1 - p_n}{p_n}$
while $\mathbf{v} \cdot \mathbf{H}^{\top} \neq 0$ or $i < \text{Iterations}_{\text{MAX}}$ do
  CN Processing:
      $L(r_{mn}^{(i)}) = \left( \prod_{n' \in N(m)\backslash n} \alpha_{n'm}^{(i-1)} \right) 2\tanh^{-1} \left( \sum_{n' \in N(m)\backslash n} \tanh \left( \frac{\beta_{n'm}^{(i-1)}}{2} \right) \right)$   (2.22)
      $\alpha_{nm} = \text{sgn}\left(L\left(q_{nm}\right)\right),\ \beta_{nm} = |L\left(q_{nm}\right)|$
  VN Processing:
      $L(q_{nm}^{(i)}) = L(v_n) + \sum_{m' \in M(n)\backslash m} L(r_{m'n}^{(i)})$   (2.23)
  a-posteriori decoding:
      $L(Q_n^{(i)}) = L(v_n) + \sum_{m' \in M(n)} L(r_{mn}^{(i)})$   (2.24)
  Hard-decoding: $\hat{c}_n = \begin{cases} 1 \Leftarrow L(Q_n) < 0 \\ 0 \Leftarrow L(Q_n) > 0 \end{cases}$
end while
```

Simplifications to the LSPA have a powerful motivation, despite the majority of arithmetic functions performed being additions, there is a dependency on the transcendental functions tanh and $\tanh^{-1}$ that requires the use of complex instructions for any decoder implementation. Thus, several authors have proposed different approximation strategies allowing fast and simplex computation of the transcendental functions required. In[78], *piecewise* (PW) linear and *lookup-table* (LUT) approximations to the tanh function were proposed, which achieve a decoding performance close to the original LSPA[78,79], while 2-D LUT approximations have also been proposed[67].

Other approaches are based on the fact that tanh and $\tanh^{-1}$ are monotonically increasing and have an odd symmetry, i.e. $\tanh(x) = -\tanh(-x)$ and $\tanh^{-1}(x) = -\tanh^{-1}(-x)$. Hence, we can separate $L(U \boxplus V)$ in signal and magnitude[68,80], thus writing

$$\begin{aligned} L(U \boxplus V) &= 2\tanh^{-1}\left(\tanh\left(\frac{L(U)}{2}\right)\tanh\left(\frac{L(V)}{2}\right)\right) && (2.25)\\ &= \operatorname{sgn}\left(L(U)\right)\operatorname{sgn}\left(L(V)\right) \times \phi\left(\phi\left(|L(U)|\right) + \phi\left(|L(U)|\right)\right) && (2.26)\\ &= \operatorname{sgn}\left(L(U)\right) \times \operatorname{sgn}\left(L(V)\right) \times \min\left(|L(U)|, |L(V)|\right) + \log\left(\frac{1+e^{-|L(U)+L(V)|}}{1+e^{-|L(U)-L(V)|}}\right) && (2.27) \end{aligned}$$

where $\phi(x) = \log\left(\frac{e^x+1}{e^x-1}\right)$. The function $\phi(x)$ is an involution transform, i.e. $\phi(\phi(x)) = x$, which allows the CN Processing expression to be expressed in terms of the function $\phi(x)$ in Algorithm 2.3, using Gallager's approach (2.26) resursively.

Algorithm 2.3 CN Processing for the LSPA using Gallager's approach

$$L(r_{mn}^{(i)}) = \left(\prod_{n' \in N(m)\setminus n} \alpha_{n'm}^{(i-1)}\right)\phi\left(\sum_{n' \in N(m)\setminus n} \phi\left(\frac{\beta_{n'm}^{(i-1)}}{2}\right)\right) \qquad (2.28)$$

The "tanh rule" (2.25) can be expressed by applying the Jacobian logarithm twice (2.27).

The Jacobian approach can be successfully deployed on the LSPA algorithm in a recursive fashion[81]. For such purpose two auxiliary sets need to be defined, f_i and b_i, and the LSPA following the Jacobian approach is shown in Algorithm 2.4, with the underlying term $\log(1+e^{-|x|})$ approximated through a LUT search or a PW approximation[82].

2.3.3 Min-Sum Decoding Algorithm

The *min-sum algorithm* (MSA) is a widely used sub-optimal decoding algorithm for LDPC codes[3,83–85]. It is sub-optimal in the sense that the MSA was derived through a majorant approximation of the LSPA CN processing. Based on the fact that in (2.27)

Algorithm 2.4 CN Processing for the LSPA using the Jacobian approach

Suppose $N(m) = (n_1, n_2, \cdots, n_{d_c})$ and the incoming messages $L(q_{n_1}), L(q_{n_1}), \cdots, L(q_{n_{d_c}})$

The auxiliar set f_i is $f_1 = L(q_{n_1})$, $f_2 = f_1 \oplus L(q_{n_1})$, ..., $f_{d_c} = f_{d_c-1} \oplus L(q_{n_{d_c}})$

The auxiliar set b_i is $b_{d_c} = L(q_{n_{d_c}})$, $b_{d_c-1} = b_{d_c} \oplus L(q_{n_{d_c-1}})$, ..., $b_{n_1} = b_{n_2} \oplus L(q_{n_1})$

$$L(q_{nm}) = \begin{cases} L(b_2), i = 1 \\ L(f_{i-1} \oplus b_{i+1}), i = 2, 3, ..., d_c - 1 \\ L(f_{d_c-1}), i = d_c \end{cases} \tag{2.29}$$

$|L(U \boxplus V)| \leq \min\left(|L(U)|, |L(V)|\right)$, the MSA relaxes on the $\log\left(\dfrac{1+e^{-|L(U)+L(V)|}}{1+e^{-|L(U)-L(V)|}}\right)$ term in (2.27) to process the CN processing as summarized in Algorithm 2.5.

Algorithm 2.5 CN Processing for the MSA

$$L(r_{mn}^{(i)}) = \left(\prod_{n' \in N(m) \backslash n} \alpha_{n'm}^{(i-1)}\right) \min_{n' \in N(m) \backslash n} \left(\beta_{n'm}^{(i-1)}\right) \tag{2.30}$$

$$\alpha_{nm} = \operatorname{sgn}\left(L\left(q_{nm}\right)\right), \tag{2.31}$$

$$\beta_{nm} = \left|L\left(q_{nm}\right)\right|$$

The MSA is computationally less complex, since it requires no further numerical operation than additions. However, for a given LDPC code, $L(r_{mn})$ obtained through the LSPA and $L(\hat{r}_{mn})$ obtained by the MSA, the following inequality holds

$$|L(\hat{r}_{mn})| > |L(r_{mn})|, \tag{2.32}$$

which has been dully noted in[85] and several corrections have been proposed to the MSA formulation which address the overestimation of the β messages.

Scaled Min-Sum Algorithm The MSA CN approximation can be improved through a normalizing factor γ, designated as *normalized min-sum algorithm* (NMSA) that is shown in Algorithm 2.6, such that $|\gamma| > 1$.

Algorithm 2.6 CN Processing for the normalized-MSA

$$L(r_{mn}^{(i)}) = \left(\prod_{n' \in N(m) \backslash n} \alpha_{n'm}^{(i-1)}\right) \frac{\min_{n' \in N(m) \backslash n} \left(\beta_{n'm}^{(i-1)}\right)}{\gamma} \tag{2.33}$$

Offset Min-Sum Algorithm In addition to a normalizing factor, the MSA can also benefit from LLR scaling[86], designated as *offset min-sum algorithm* (OMSA) that is shown in Algorithm 2.7, in one of two approaches. The value of constant c is estimated via DE[86,87] and in the first approach, it is always subtracted from the q_{mn} messages. In another approach[85], it quantizes the β messages below a given c towards 0 and subtracts c to the remaining messages.

Algorithm 2.7 CN Processing for the offset-MSA

Approach 1 (2.34), approach 2 (2.35):

$$L(r_{mn}^{(i)}) = \left(\prod_{n' \in N(m) \backslash n} \alpha_{n'm}^{(i-1)} \right) \min_{n' \in N(m) \backslash n} \left(\beta_{n'm}^{(i-1)} - c \right) \tag{2.34}$$

$$L(r_{mn}^{(i)}) = \left(\prod_{n' \in N(m) \backslash n} \alpha_{n'm}^{(i-1)} \right) \max \left(\min_{n' \in N(m) \backslash n} \left(\beta_{n'm}^{(i-1)} - c, 0 \right) \right) \tag{2.35}$$

Self-Corrected Min-Sum Algorithm Another approach to address the overestimation of the $L(q_{nm})$ messages has been proposed that instead of introducing a correction to every $L(q_{nm})$ message in every iteration, the correction occurs only when a signal change would happen in the $L(q_{nm})$ messages under regular MSA decoding[3,83]. In this event, the algorithm introduces an erasure to the $L(q_{nm})$ message, setting the reliability of each bit state to equiprobable (2.21). The formulation of the *self-corrected min-sum algorithm* (SCMSA) is shown in Algorithm 2.8 and is .

Algorithm 2.8 VN Processing for the SCMSA

$$L^* \left(q_{mn}^{(i)} \right) = L\left(p_n\right) + \sum_{m' \in M(n) \backslash m} L\left(r_{m'n}^{(i)} \right)$$

$$L(q_{mn}^{(i)}) = \begin{cases} 0, \; L^* \left(q_{mn}^{(i)} \right) \times L\left(q_{mn}^{(i-1)} \right) < 0 \wedge L\left(q_{mn}^{(i-1)} \right) \neq 0 \\ L^* \left(q_{mn}^{(i)} \right), \text{ otherwise.} \end{cases} \tag{2.36}$$

2.4 LDPC Codes over GF(q)

In this section, we overview LDPC codes extended to *Galois field of dimension q* (GF(q)) and corresponding decoding algorithms. Due to the FEC purposes served by LDPC codes, special focus will be given to the GF(2^m) case. The aforementioned decoding algorithms are highly optimized for the particular case of codes defined over GF(2). In reality, their generalized version over GF(q), which also includes the GF(2^m) case, are significantly more complex. In short, LDPC codes defined over GF(q) have parity-check matrices whose elements h_{ij} are defined over GF(q) themselves. Furthermore, instead of bits, message symbols defined over the Galois field are encoded into codeword symbols. In the particular case of GF(2^m), the symbols are m-tuples of bits[a)].

2.4.1 Factor Graph

The Tanner graph in GF(q) (factor graph)is an incomplete tripartite factor graph composed of permutation nodes and of VNs and CNs. The permutation nodes are included between VNs and CNs, as seen in Figure 2.3. They are included due to the parity-check restriction

$$\left(\sum_{i \in M(j)} h_{ij}(x) c_j(x) \mod p(x) \right) = 0, \tag{2.37}$$

where $p(x)$ is a $m-1$ degree primitive polynomial to GF(2^m). Each CN$_j$ evaluates the parity-check restriction to each $i_j(x)$, $x \in$ GF(2^m) (2.38), which does not allow for a straightforward evaluation. In order to do so, (2.38) must be normalized by $h_j^{-1}(x)$ (2.39), which is similar to the binary parity-check equation (2.4).

$$\sum_{i' \in M(j) \setminus i} h_{i'j}(x) c_j(x) = h_{ij}(x) c_j(x) \mod p(x) \tag{2.38}$$

$$h_{ij}^{-1}(x) \times \sum_{i' \in M(j) \setminus i} h_{i'j}(x) c_j(x) = c_j(x) \mod p(x) \tag{2.39}$$

The actual effect of the factor $h_{ij}^{-1}(x)$ in (2.39) is a permutation, due to the closure property of GF(2^m)[47]. Thus, if for instance the $q_{nm}(x)$ messages are grouped in a *probability mass function* (*pmf*) vector we observe the following

$$\mathbf{m}_{vc}(x) = \begin{bmatrix} m_{vc}(0) \\ m_{vc}(1) \\ m_{vc}(\alpha) \\ \vdots \\ m_{vc}(\alpha^{2^m-2}) \end{bmatrix}, \ \mathbf{m}_{vc}(\alpha \cdot x) = \begin{bmatrix} m_{vc}(0) \\ m_{vc}(\alpha^{2^m-2}) \\ m_{vc}(1) \\ m_{vc}(\alpha) \\ \vdots \end{bmatrix}, \ \mathbf{m}_{vc}(\alpha^2 \cdot x) = \begin{bmatrix} m_{vc}(0) \\ m_{vc}(\alpha^{2^m-3}) \\ m_{vc}(\alpha^{2^m-2}) \\ m_{vc}(1) \\ \vdots \end{bmatrix} \cdots, \tag{2.40}$$

[a)] Arithmetic over GF(q) is defined in Appendix B.

and likewise

$$\mathbf{m}_{vc}(x) = \begin{bmatrix} m_{vc}(0) \\ m_{vc}(1) \\ m_{vc}(\alpha) \\ \vdots \\ m_{vc}(\alpha^{2^m-2}) \end{bmatrix}, \ \mathbf{m}_{vc}(\alpha^{-1} \cdot x) = \begin{bmatrix} m_{vc}(0) \\ m_{vc}(\alpha) \\ \vdots \\ m_{vc}(\alpha^{2^m-2}) \\ m_{vc}(1) \end{bmatrix}, \ \mathbf{m}_{vc}(\alpha^{-2} \cdot x) = \begin{bmatrix} m_{vc}(0) \\ m_{vc}(\alpha^2) \\ \vdots \\ m_{vc}(1) \\ m_{vc}(\alpha) \end{bmatrix} \cdots . \tag{2.41}$$

The introduction of the permutation nodes is depicted in Figure 2.3 and shows the inclusion of the latter in a portion of the Tanner graph accounting for a degree 3 parity-check restriction. The permutations nodes account for the cyclic shift in (2.38) and (2.39),

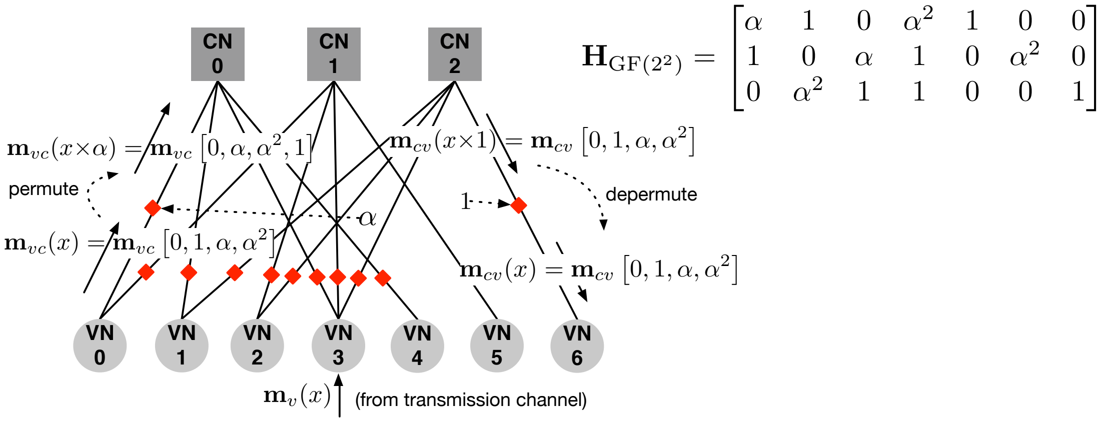

CN 0
CN 1
CN 2
$\mathbf{H}_{\mathrm{GF}(2^2)} = \begin{bmatrix} \alpha & 1 & 0 & \alpha^2 & 1 & 0 & 0 \\ 1 & 0 & \alpha & 1 & 0 & \alpha^2 & 0 \\ 0 & \alpha^2 & 1 & 1 & 0 & 0 & 1 \end{bmatrix}$
$\mathbf{m}_{vc}(x \times \alpha) = \mathbf{m}_{vc}\,[0, \alpha, \alpha^2, 1]$
permute
α
$\mathbf{m}_{vc}(x) = \mathbf{m}_{vc}\,[0, 1, \alpha, \alpha^2]$
$\mathbf{m}_{cv}(x \times 1) = \mathbf{m}_{cv}\,[0, 1, \alpha, \alpha^2]$
1
depermute
$\mathbf{m}_{cv}(x) = \mathbf{m}_{cv}\,[0, 1, \alpha, \alpha^2]$
VN 0
VN 1
VN 2
VN 3
VN 4
VN 5
VN 6
$\mathbf{m}_v(x)$
(from transmission channel)

Figure 2.3: Factor graph extension to the Tanner graph with inclusion of the permutation nodes accounting for the non-null elements of **H** . Also show is the general message-passing procedure used by the decoding algorithms over GF(q).

where the permutation is executed for messages traveling from VN to CN and the depermutation are executed on messages traversing on the opposite direction, as seen in Figure 2.3. Thus, the *pmfs* messages exchanged in the graph are no longer sorted in ascending order. Instead, messages are permuted when they traverse the permutation nodes in the direction VN-CN and depermuted when traversing in the direction CN-VN.

2.5 LDPC Decoding Algorithms over GF(q)

The numerical complexity associated with the decoding of non-binary LDPC codes is higher than that of binary ones. This is not only due to the finite field dimension, but also due to the higher number of symbol combinations that validate any given parity-check restriction of the code. Likewise, the soft-decoding procedure can be divided into the following steps.

i) **Initialization**: the demodulator computes all the probabilities in the *pmf* $\mathbf{m}_v(x)$ for $x \in$ GF(2^m) and this information is broadcast to all VNs which initialize the messages $\mathbf{m}_{vc}(x), \forall c$;

ii) **Permutation**: the $\mathbf{m}_{vc}(x)$ messages are permuted when traversing the edge towards their adjacent CNs;

iii) **CN processing**: the CNs receive the $\mathbf{m}_{vc}(x)$ messages from their adjacent VNs and compute $\mathbf{m}_{cv}(x)$ messages according to the algorithm's CN update rule;

iv) **Depermutation**: the $\mathbf{m}_{cv}(x)$ messages are depermuted when traversing towards their adjacent VNs;

v) **VN processing**: the VNs receive the $\mathbf{m}_{cv}(x)$ messages from their adjacent CNs and compute new $\mathbf{m}_{vc}(x)$ messages according to the VN update rule;

vi) **Hard decoding**: the VNs also compute $\mathbf{m}^*{}_{vc}(x)$ messages from the $\mathbf{m}_{cv}(x)$ messages received and compute the most likely symbol state according to this estimate;

vii) **Repeat process**: go back to *i)* until a maximum number of iterations is reached or a valid codeword is produced.

2.5.1 Sum-Product Algorithms

The SPA presented in Algorithm 2.1, is a formulation tuned to GF(2). A general formulation of the SPA, which is able to work on all GF(q), is presented in Algorithm 2.9. Due to the LDPC code parity-check restrictions over GF(q) (2.37), it is required to evaluate the equation (2.42) for any combination of $|V(c) \setminus v|$ elements taking q possible values. This constitutes the motivation to pursue decoding algorithms with lower numerical complexities even if at the cost of sub-optimality. It should be noted that the numerical complexity O($\cdot$) takes into account only the CN processing numerical complexity, as the CN processing is typically much more intensive than the VN processing.

Davey and Mackay Sum-Product Algorithm Davey and Mackay propose an approach on which the CN processing in Algorithm 2.9 is broken down into two parcels[46]. They define the partial sums $\sigma_{ck} = \sum\limits_{v:v \leq k} H_{cv} x_v^{'}$ and $\rho_{cv} = \sum\limits_{v:v \geq k} H_{cv} x_v^{'}$. Then, by fixing $a \in$ GF(q) it is possible to compute $p(\sigma_{ck} = a)$ for each $k \in V(c)$, and for c, v if they are successive indexes in $V(c)$ we may define

$$p(\sigma_{ck} = x) = \sum_{\{s,t:H_{cv}t+s=x\}} p(\sigma_{cv} = s)\mathbf{m}_{cv}(t), \tag{2.45}$$

Algorithm 2.9 SPA generalized to GF(q)

```
Initialization: $\mathbf{m}_v(x) = \mathbf{m}_{vc}^{(0)}(x) = p(v_n = x|y_n)$
while $\mathbf{v} \cdot \mathbf{H}^\top \neq 0$ or $i <$ Iterations$_{\text{MAX}}$ do
    CN Processing:
        $$\mathbf{m}_{cv}^{(i)}(x) = \sum_{\mathbf{v}:c_v=x} p(z_c = 0|\mathbf{v}) \prod_{v' \in V(c)\backslash v} \mathbf{m}_{v'c}(x) \quad (2.42)$$
    With $p(z_m = 0|c)$ being either 0 or 1 according to whether $\mathbf{v}$ satisfies parity-check $m$.
    VN Processing:
        $$\mathbf{m}_{vc}^{(i)}(x) = k_{vc} p_v(x) \prod_{c \in C(v)\backslash v'} \mathbf{m}_{c'v}(x),\ k_{vc} \Leftarrow \sum \mathbf{m}_{vc}^{(i)}(x) = 1 \quad (2.43)$$
    a-posteriori decoding:
        $$\mathbf{m}^{*(i)}_{v}(x) = K_v p_v(x) \prod_{c \in C(v)} \mathbf{m}_{vc}(x),\ K_v \Leftarrow \sum \mathbf{m}^{*(i)}_{v}(x) = 1 \quad (2.44)$$
    Hard-decoding: $\hat{c}_n = \arg\max_x \mathbf{m}_v^*(x)$
end while
```

and ρ_{mk}

$$p(\rho_{ck} = x) = \sum_{\{s,t:H_{cv}t+s=x\}} p(\rho_c = s)\mathbf{m}_{cv}(t). \tag{2.46}$$

Substitution of (2.45) and (2.46) in (2.42) yields Algorithm 2.10. This approach allows

Algorithm 2.10 CN Processing for the Davey and Mackay SPA

```
CN Processing:
    $$\mathbf{m}_{cv}(x) = p\left((\sigma_{c(v-1)} + \rho_{c(v+1)} = z_c - H_{cv}x\right) \quad (2.47)$$
    $$= \sum_{\{s,\ t:s+t=z_c-H_{cv}a\}} p(\sigma_{c(v-1)} = s)p(\rho_{c(v+1)} = s) \quad (2.48)$$
```

for a recursive search and computation of the symbols which validate each parity-check equation. However, simplicity is not the main characteristic of this algorithm, rendering a very high numerical complexity of $\text{O}(M \cdot d_c \cdot 2^{2^m})$. Thus, it scales exponentially with the binary extension field dimension, making fast decoding close to impossible. In fact, due to this, only low order binary extension fields, up to GF(2^3), were considered initially[46]. There are other factors that also detract the use of this as a formulation of the SPA. Namely, the arithmetic nature of the algorithm, constituted mainly by multiplications, which are expensive operations, the more expensive due to its GF(2^m) domain, and the need to normalize the probabilities at the VN processing, an even more costly operation. Inevitably, further refinements were proposed.

Logarithmic Sum-Product Algorithm

Wymeersch *et al.* [88] proposed a redefinition of the decoding domain to the log-likelihood domain in order to improve the numerical complexity and the nature of the arithmetic operations executed. Due to its log-likelihood domain, this algorithm is referred as LSPA.

The LSPA in GF(q) requires the definition of *log-likelihood ratio vectors* (LLRVs), which store $q-1$ LLRs, with each element defined by $L(v=x)=\log[p(v=x)/p(v=0)]$, where VNs are initialized as follows

$$L_{ch}(c_v)_c = \sum_{v:\psi_1(a_c)_v=+1} \frac{2v_{lc+v}}{\sigma}, \tag{2.49}$$

with $\psi_1(a_c)_v$ the inversion of the mapping function from GF(2^m) to $\{-1,1\}$ [88]. In the log-likelihood domain, multiplications become additions and thus, the VN processing and *a-posteriori* decoding phases formulated in Algorithm 2.9 can be rewritten to sums of LLRVs. As in the binary case, the CN processing is not straightforward even in log-domain, as the domain changes introduces a non-linear ⊞-like dependency. This dependency is dealt with by the extension of the ⊞ function to GF(2^m) [89]

$$\begin{aligned}
L(a_1v_1+a_2v_2) &= \boxplus(L_1,L_2,A_1,A_2)\\
&= \log\left(e^{L_1(A_1^{-1}\alpha_i)}+e^{L_2(A_2^{-1}\alpha_i)}+\sum_{x\in\mathrm{GF}_0(2^m)\backslash\alpha_i A^{-1}} e^{L_1(x)+L_2(A_2^{-1}(\alpha_i-xA_1))}\right)\\
&- \log\left(1+\sum_{x\in\mathrm{GF}} e^{L_1(x)+L_2(A_2^{-1}A_1x)}\right).
\end{aligned} \tag{2.50}$$

By using (2.50) the Wymeersch LSPA is formalized in Algorithm 2.11. The VN processing step becomes a simple sum of LLRVs similar to the binary LSPA. The logarithm of the sum of exponents in (2.51) can be efficiently dealt with through the Jacobian logarithm

Algorithm 2.11 CN Processing for the Wymeersch LSPA

$$\begin{aligned}
L(\sigma_{cj}) &= L(\sigma_{c(j-1)}+H_{cj}x_j)\\
L(\rho_{cj}) &= L(\rho_{c(k+1)}+H_{cj}x_k)\\
L(r_{cv}(x)) &= L(H_{cv}^{-1}\sigma_{c(v-1)}+H_{cv}^{-1}\rho_{c(v+1)}),
\end{aligned} \tag{2.51}$$

$$\begin{aligned}
max^*(x,y) &= \log(e^x+e^y)\\
&= max(x,y)+\log(e^{-|x1-x2|}).
\end{aligned} \tag{2.52}$$

The predominant numerical nature of this algorithm is less expensive, as the majority of multiplications are converted to additions. However, the CN processing introduces the computation of the $\boxplus$ function which requires the computation of logarithms of a sum of exponents. Although it can be efficiently computed through the Jacobian logarithm (2.52), the need of computing a logarithm term still prevails. Moreover, and despite a lower numerical complexity $\mathrm{O}(M \cdot d_c(2^m-1)^2)$ still scales with the square of the field dimension.

FFT Sum-Product Algorithm

The *FFT sum-product algorithm* (FFT-SPA) results from preliminary works by Mackay and Byers and further explored by Barnault and Declercq[77,90,91].

The *fast Fourier transform* (FFT) inclusion stems from the fact that the CN processing in Algorithm 2.9 is as a convolution of probabilities. Since the convolution property of the Fourier Transform $\mathsf{F}(\cdot)$ yields

$$x * y = \mathsf{F}^{-1}\left\{\mathsf{F}(x) \cdot \mathsf{F}(y)\right\}, \tag{2.53}$$

where $\mathsf{F}^{-1}(\cdot)$ is the inverse Fourier transform, the Fourier transform of a convolution of probabilities is the product of the probabilities' tranforms.

The discrete-time Fourier Transform can be computed by the FFT algorithm, and for codes defined over GF(2^m) it further simplifies to the Walsh-Hadamard transform, which can be efficiently computed by the *fast Walsh-Hadamard transform* (FWHT)—which can be thought of the FFT with the twiddle coefficients defined over 0 and π. Thus, the CN processing of FFT-SPA is simplified from its formulation in Algorithms 2.9 and 2.10 to the one in Algorithm 2.12. Since the FFT-SPA exploits the Fourier domain properties in the convolution of the *pmfs* in the CN processing, the operations in GF(2^m) previously necessary in aforementioned approaches are no longer necessary. Naturally, prior and after the CN processing, the *pmfs* must be changed accordingly to the appropriate domain by applying the FWHT—the involution property can be explored herein as the inverse of the transform is the transform itself. The numerical complexity is lowered to $\mathrm{O}(M \cdot d_c(2^m)m)$, which allows the decoding of higher order fields in a feasible timespan.

Algorithm 2.12 CN Processing for the FFT-SPA

All steps follow the SPA in Algorithm 2.9 except,

$$\begin{aligned} \mathbf{m}_{cv}^{(i)}(x) &= \mathsf{F}^{-1}\left(\prod_{v' \in V(c) \backslash v} \mathsf{F}\left(\mathbf{m}_{v'c}(x)\right)\right) \\ &= \mathsf{W}\left(\prod_{v' \in V(c) \backslash v} \mathsf{W}\left(\mathbf{m}_{v'c}(x)\right)\right) \end{aligned} \tag{2.54}$$

Log Permutation FFT Sum-Product Algorithm

Song and Cruz[92] proposed a variation of the FFT-SPA operating on the log-domain, the *log permutation FFT-SPA* (LPFSPA). However, whereas in the LSPA messages were expressed in LLR, under this approach messages are changed to its log representation, i.e. $p(x) \rightarrow \log(p(x))$. The log-domain usage is to exploit multiplications becoming additions, which can be directly replaced in the VN processing and *a-posteriori* processing steps. However, as so happens with the former approaches, the problem lies in the CN processing.

Rewriting (2.42) to accommodate the domain change easily leads to (2.54). However, the domain change poses the non-trivial task of computing the FFT on the log-domain, since the multiplicands are *pmf* which may possess negative values. This can be overcome by defining a log-like function LG as follows

$$\begin{aligned} LG : \mathbb{R} &\rightarrow \{-1, 1\} \times \mathbb{R} \\ u = (u^{'}, u^{''}) &= (\operatorname{sgn}(v), \log|v|), \end{aligned} \tag{2.55}$$

where $\mathbb{R}$ is the field of reals and the inverse $LG^{-1} : \{-1, 1\} \times \mathbb{R} \rightarrow \mathbb{R}$

$$v = u^{'} e^{u^{''}}, \tag{2.56}$$

and the basic arithmetic operations $+$, $-$, $\times$, $\div$ are defined, so that the CN processing of LPFSPA can be defined as well[92].

This approach possesses the same numerical complexity as the FFT-SPA, which is $\mathrm{O}(M \cdot d_c \cdot m \cdot 2^m)$. Although there are also no operations over GF(2^m), the dependency on a logarithm of the sum of exponents remains, or else simplifying through the Jacobi logarithm, a logarithmic dependency is inevitable.

Log Fourier Sum-Product Algorithm

The *log-Fourier SPA* (LFSPA) has been proposed by Kasai and Sakaniwa[93,94] and addresses the asymmetry in numerical complexity between the CN and the VN processing. Since the bottleneck in decoding time will be caused by the most complex processing, even fully parallel decoding cannot provide fast decoding times. Thus, the LFSPA focus on balancing the numerical complexity of the CN and VN processing.

In order to do so, a logarithm-like function $\Gamma(x) : [-1,1] \rightarrow \text{GF(2)} \times [-\infty, 0]$ is defined as

$$\Gamma(x) = (\text{sgn}_{\text{GF(2)}}(x), \log(|x|)) \in \text{GF(2)} \times [-\infty, 0]$$
$$\text{sgn}_{\text{GF(2)}}(x) = \begin{cases} 0 \in \text{GF(2)},\ x > 0 \\ \text{randomly choose 0 or 1},\ x = 0 \\ 1 \in \text{GF(2)},\ x < 0 \end{cases}, \tag{2.57}$$

and for any non-zero $x, y \in \mathbb{R}$, $\Gamma(x \cdot y) = \Gamma(x) + \Gamma(y)$, with $\Gamma^{-1}(\cdot)$ well-defined. The LFSPA formal outline is shown in Algorithm 2.13. Furthermore, increasing the numerical complexity of the VN processing is not as critical as increasing the CN processing. The known best performing non-binary LDPC codes are $(2, d_c)$-codes[95], which for a given rate R gives a CN weight $d_c = 2/(1-R)$. Thus, for very high rate LDPC codes d_c becomes high and the CN processing complexity also scales with it, which motivates the transfer of complexity to the VN processing. Moreover, if the VN is fixed at 2, even a complex VN poses no inconvenience due to the low number of nodes each VN is connected to. The LFSPA increases the VN complexity to $\text{O}(N \cdot d_v \cdot 2^m)$ while reducing the CN processing to $\text{O}(M \cdot d_c \cdot 2^m)$.

Tensorial Sum-Product Algorithm The SPA present in Algorithm 2.9 is applicable to every finite field extension and non-extension fields. However, in the specific case of GF(2^m), we can rely on a tensorial likelihood representation[85,91,96]. This is possible due to the extension field m-tuple definition

$$\mathbf{u}(x) = \sum_{k=1}^{m} u_k \cdot x^{K-1},\ u_k \in \{0,1\},\ \forall k \in \{1, ..., q\}, \tag{2.69}$$

which allows elements in GF(2^m) to be written as m-size tensor with dimension 2. Hence, instead of indexing the messages in the form $\mathbf{m}_{vc}(x),\ x \in \text{Gf}(2^m)$, for messages travelling from VNs to CNs, messages can be indexed by (2.69) in the form $\mathbf{m}[\mathbf{u}(x)] = \mathbf{m}[u_1, ..., u_m]$.

The SPA is essentially the same using one or the other representation. However, the tensorial formulation not only allows for a more comprehensive description of the decoding algorithm, but also allows a structured indexing of the messages.

Moreover, under the tensorial representation the permutation and de-permutation may be accomplished by a permutation of the m-tuple coefficients that index the messages. Thus, the permutation and depermutation in (2.70) is indexed by the m-tuple coefficients similar to permuting and depermuting using the symbol representation in (2.40) and (2.41),

$$\alpha \cdot \mathbf{m}[u_1, u_2, ..., u_{2^m-2}] = \mathbf{m}[u_2, ..., u_{2^m-2}, u_1], \tag{2.70}$$

Algorithm 2.13 Log-Fourier Sum-product Algorithm

```
Initialization:
    $\mathbf{m}'_v(x) = p(c_v = x|y_v),\ x \in \mathrm{GF}(2^m) \quad (2.58)$
    $\mathbf{M}'_v(z) = \sum_{x \in \mathrm{GF}(2^m)} \mathbf{m}_v(z)(-1)^{z \cdot x},\ z \in \mathrm{GF}(2^m) \quad (2.59)$
    $\mathbf{m}_v(z) = \Gamma(\mathbf{M}'_v(z)),\ z \in \mathrm{GF}(2^m) \quad (2.60)$
while $\mathbf{v} \cdot \mathbf{H}^\top \neq 0$ or $i <$ Iterations$_{\mathrm{MAX}}$ do
    CN Processing:
        $\tilde{\mathbf{m}}^{(i)}_{vc}(z) = \mathbf{m}^{(i)}_{vc}(h_{cv} z) \quad (2.61)$
        $\tilde{\mathbf{m}}^{(i)}_{cv}(z) = \sum_{v' \in V(c) \backslash v} \tilde{\mathbf{m}}^{(i)}_{v'c}(z) \quad (2.62)$
        $\mathbf{m}^{(i)}_{cv}(z) = \tilde{\mathbf{m}}^{(i)}_{cv}(h^{-1}_{cv} z) \quad (2.63)$
    BN Processing:
        $\mathbf{m}^{(i)}_{vc}(z) = p_v(z) \boxplus_{v' \in C(v) \backslash c} \mathbf{c}^{(i)}_{c'v}(z) \quad (2.64)$
        where $\Lambda_1 \boxplus \Lambda_2 \in (\mathrm{GF}(2) \times [-\infty, 0])^{2^m}$ is defined as:
        $(\lambda_1 \boxplus \lambda_2)(x) = \Gamma\left( \sum_{x_1, x_2 \in \mathrm{GF}(2^m) : x = x_1 + x_2} \Gamma^{-1}(\lambda_1(x_1) + \lambda_2(x_2)) \right) \quad (2.65)$
    *a-posteriori* decoding:
        $\mathbf{m}{*}^{(i)}_v(z) = \mathbf{m}_v(z) \boxplus_{c' \in C(v)} \mathbf{m}^{(i)}_{c'v}(z) \quad (2.66)$
    Hard-decoding:
        $\mu_v(x) = \sum_{z \in \mathrm{GF}(2^m)} \mathbf{m}{*}^{(i)}_v(z)(-1)^{z \cdot x} \quad (2.67)$
        $c_v = \arg\max_x \mu^{(i)}_v(x) \quad (2.68)$
end while
```

and the symmetric permutation occurs for α^{-1}.

We are also able to define a configuration set as follows

$$\mathrm{Conf}_{i_t}(x) = \left\{ \{v_c(x)\}_{c \neq t} : \sum_{c \in C(v)} v_c(x) = 0 \right\}, \tag{2.71}$$

which provides an alternative definition to $p(z_v = 0|\mathbf{v})$ in (2.42) and also defines a particularly powerful tool for arriving at the *extended min-sum* (EMS) definition.

Under this new representation we may write the SPA in Algorithm 2.14. The nomenclature of the tensorial SPA is similar to the original SPA formulation, only that symbols are indexed by their polynomial in GF(2) rather than their values in GF(2^m) and now we account for the permutation and de-permutation operations explicitly, due to the permutation nodes inclusion in the Tanner graph. It is noteworthy that under the SPA definition in Algorithm 2.9, the permutation and de-permutation were not explicitly included. This is due to the $p(z_v = 0|\mathbf{v})$ term in the CN processing that conceals such operations, despite being already implicit in such definition. This transformation in itself, does not account for any simplification of the algorithm nor does it provide a means of reducing its numerical complexity.

Algorithm 2.14 Tensorial SPA for GF(q)

```
while $\mathbf{v} \cdot \mathbf{H}^\top \neq 0$ or $i <$ Iterations$_{\text{MAX}}$ do
    CN Processing$^1$:
        $\mathbf{m}_{cv}^{(i)}[h_{cv} \times (i_{t_1}, \cdots, i_{t_v})] = \sum_{\{i_c(x)\} \in \text{Conf}_{i_t(x)}} \prod_{c=1, m \neq t}^{d_c} \mathbf{m}_{cv}^{(i)}[h_{cv} \times (i_{c_1}, \cdots, i_{c_v})]$    (2.72)
    BN Processing$^2$:
        $\mathbf{m}_{vp}^{(i)}[i_1, \cdots, i_p] = k_{vp}^{(i)} p[i_1, \cdots, i_p] \prod_{v=1, n \neq t}^{d_v} r_{pv}^{(i)}[i_1, \cdots, i_p]$    (2.73)
        $k_{vp}^{(i)} \Leftarrow \sum \mathbf{m}_{vp}^{(i)}[i_1, \cdots, i_p] = 1$    (2.74)
    a-posteriori decoding:
        $\mathbf{m}^{*(i)}_{v}[i_1, \cdots, i_p] = K_v p_v[i_1, \cdots, i_p] \prod_{v=1}^{d_v} \mathbf{m}_{cv}[i_1, \cdots, i_p]$    (2.75)
        $K_v \Leftarrow \sum \mathbf{m}^{*(i)}_{v}[i_1, \cdots, i_p] = 1$    (2.76)
    Hard-decoding:
        $\hat{c}_v = \arg\max_x \mathbf{m}_v[i_1, \cdots, i_z]$
end while
```

Mixed Domain Algorithm

The *mixed-domain algorithm* (MDA)[97] combines the low-complexity of the FFT-SPA with the log-domain approach of the LPFSPA. The rational is to benefit from the advantages presented by both the probability domain and the log domain. Thus, all the operations regarding the FFT are executed in the linear domain and the VN and CN operations in the log domain.

In order to do so, two domain changes must occur

$$p(x) \rightarrow \log\left(p(x)\right), \tag{2.77}$$

and its inverse

$$\log (p(u)) \rightarrow p(u), \tag{2.78}$$

the former from linear to log and the latter from log to linear. Albeit possessing the same numerical complexity as the former two algorithms, the MDA improves in requiring only the computation of logarithms and not of logarithms of sums of exponents.

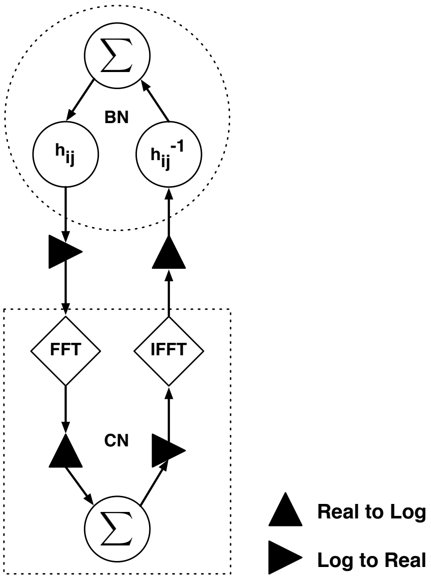


Figure 2.4: Mixed-domain message-passing operations and domain transformations.

The $\mathbf{m}_{vc}$ messages, those leaving the VNs, are permuted before being transformed into the linear domain. The FFT is applied to change the messages to the log domain prior to the convolution being computed. The messages leaving the CNs, i.e. the $\mathbf{m}_{cv}$ messages, are re-changed to the linear domain prior to the *inverse fast Fourier transform* (iFFT) computation and afterwards its output is reconverted to the log domain. The messages leaving the CNs, i.e. $\mathbf{m}_{cv}$ messages, are transformed to the real domain prior to the computation of the iFFT and afterwards converted to the log domain. Normalization is still required, but the log domain allows the normalization step to be defined in terms of subtractions[97]. The process is depicted in Figure 2.4.

2.5.2 Extended Min-Sum Algorithm

This section presents the EMS, a generalization of the MSA for GF(q), and in particular for GF(2^m) by using the tensorial notation previously introduced. In order to arrive at the definition of the MSA, a domain change was required from probability to LLR. However, whereas in the binary case we had only to define $L(u) = \log (p(u = 1)/p(u = 0))$, in the q-ary case there are $q - 1$ LLR messages to account for. Under the tensorial representation we define the LLR of $u[i_1, \cdots, i_m]$ as $L(u[i_1, \cdots, i_m])$

$$L(u[i_1, \cdots, i_z]) = \log \frac{p(u[i_1, \cdots, i_z])}{p(u[0, \cdots, 0])}, \ \forall (i_1, \cdots, i_z) \in \{0,1\}^p, \tag{2.79}$$

and therefore $L(u[0, \cdots, 0]) = 0$. It is crucial for the introduction of the configuration concept that we observe that $L(u)$ is a LLR vector sized $q - 1$.

The purpose of extending the MSA to its general EMS for GF(q) is to reduce the numerical complexity of the CN processing of the SPA. Whereas the FFT-SPA is restricted to GF(2^m) fields, the EMS is no longer constrained to extension fields. The EMS aims at assigning each message received a reliability measure not only by minimizing the number of operations required and avoiding complex operations, but also by utilizing only a limited number of incoming messages.

Configuration Sets and Configurations In order to use only a limited number of the available incoming messages, we need to introduce the configuration concept. In (2.71), the elements belonging to the configuration set are the ones that verify a given parity-check restriction. We can define in $L(\bar{\mathbf{m}}_{pm}^{k_c}),\ k_c = 1, \cdots, n_m$ the n_m largest values in $L(\mathbf{m}_{pm})$. The notation of the associated field elements $\alpha_c^{(k_m)}(x)$ is such that

$$L(\bar{\mathbf{m}}_{pm}^{(k_m)}) = \log \frac{p(h_m(x)i_m(x) = \alpha_c^{k_m})(x)}{p(h_m(x)i_m(x) = 0} = L(q_{pm}[\alpha_{c_1}^{(k_m)}, \cdots, \alpha_{c_z}^{(k_m)}]) \tag{2.80}$$

For simplicity, we will assume that n_m values are selected per node regardless of the LDPC graph characteristics. From those n_m largest values a set of configurations can be defined as

$$\mathrm{Conf}(n_m) = \left\{ \mathbf{ff_k} = \left[\alpha_1^{(k_1)(x)}, \cdots \alpha_{d_c-1}^{(k_{d_c}-1)}(x)\right]^\top \ : \ \forall \mathbf{k} = [k_1, \cdots, k_{d_c-1}]^\top \in \{1, \cdots, n_m\}^{d_c-1} \right\}, \tag{2.81}$$

such that any vector of $d_c - 1$ elements defines a configuration, and its cardinality is

$$|\mathrm{Conf}(n_m)| = n_m^{d_c-1} \tag{2.82}$$

Conf(1) contains only one configuration, thus designated the order-0 configuration. For large values of d_c or n_m, the number of configurations is also large, motivating the consideration of only a subset of Conf(n_m) for $n_c \leq d_c - 1$ such that

$$\mathrm{Conf}(n_m, n_c) = \mathrm{Conf}(n_m)^{(0)} \cup \mathrm{Conf}(n_m)^{(1)} \cup \cdots \cup \mathrm{Conf}(n_m)^{(n_c)}, \tag{2.83}$$

where Conf$(n_m)^{(l)}$ is the subset of configurations differing l entries from the order-0 configuration. Hence, Conf(n_m, n_c) is the subset of configurations differing at most n_c entries from the order-0 configuration. The cardinality of $\mathrm{Conf}(n_m, n_c)$ is

$$|\mathrm{Conf}(n_m, n_c)| = \sum_{k=0}^{n_c} \binom{d_c - 1}{k} (n_m 1 - 1)^k \simeq \binom{d_c - 1}{n_c} n_m^{n_c}, \tag{2.84}$$

and it can be easily seen that it is much smaller than (2.82) and built from configurations with large probabilities. It is also noteworthy that $\mathrm{Conf}(n_m) = \mathrm{Conf}(n_m, d_c - 1)$.

Each configuration is assigned with a reliability value $L(\mathbf{ff_k})$ which is straightforward to compute based on (2.80)

$$L(\mathbf{ff_k}) = \sum_{m=1,\dots,d_c-1} L(\bar{\mathbf{m}}_{pm}^{(k_m)}). \tag{2.85}$$

Let us also define the subset $\mathrm{Conf}_{i_{d_c}(x)}(n_m, n_c)$ defined by the parity-check constraint

$$\mathrm{Conf}_{i_{d_m}(x)}(n_m, n_c) = \left\{ \mathbf{ff_k} \in \mathrm{Conf}(n_m, n_c) : \; h_{d_m}(x) i_{d_m}(x) + \sum_{m=1}^{d_c-1} \alpha_c^{k_m}(x) = 0 \right\}. \tag{2.86}$$

Not every choice of (n_m, n_c) is able to deliver non-empty configurations, which adds convergence issues to the EMS decoding. However, this problem may be overcome if

Algorithm 2.15 EMS for GF(q)

```
Initialization:
            $\mathbf{m}_v[i(x)] = p(c_v = x|y_v)$    (2.87)
            $L(p_n[i(x)]) = L(q_{np}^{(0)}[i(x)]) \log \frac{p_n[i(x)|_{i(x)\neq 0}]}{p_n[i(x) = 0]}$    (2.88)
while $\mathbf{v} \cdot \mathbf{H}^\top \neq 0$ or $i <$ Iterations$_{\mathrm{MAX}}$ do
    CN Processing$^1$:
            from $d_c - 1$ incoming $L(q_{pm}^{(i)}[i_{c_1}, \cdots, i_{c_p}])$ messages build sets:
            $\mathrm{S}_{i_{d_m}(x)}(x) = \mathrm{Conf}_{i_{d_m}(x)}(q, 1) \cup \mathrm{Conf}_{i_{d_m}(x)}(n_m, n_c)$    (2.89)
            $L(r_{d_c p}^{(i)}[i_{d_{c_1}}, \cdots, i_{d_{cm}}]) = \max_{\mathbf{ff_k} \in \mathrm{S}_{d_c}(x)} \{L(\mathbf{ff_k})\}$    (2.90)
            $L(r_{mp}^{(i)}[i_1, \cdots, i_p]) = L(r_{mp}^{(i)}[i_{d_{c_1}}, \cdots, i_{d_{c_p}}]) - L(r_{mp}^{(i)}[0, \cdots, 0])$    (2.91)
    BN Processing$^2$:
            $L(q_{np}^{(i)}[i_1, \cdots, i_z]) = L(p[i_1, \cdots, i_z]) + \sum_{n=1, n\neq t}^{d_b} L(r_{pn}^{(i)}[i_1, \cdots, i_z])$    (2.92)
    a-posteriori decoding:
            $L(Q_n^{(i)}[i_1, \cdots, i_z]) = L(p_n[i_1, \cdots, i_z]) + \sum_{n=1}^{d_b} L(r_{pn}^{(i)}[i_1, \cdots, i_z])$    (2.93)
end while
```

the set $\mathrm{Conf}_{i_{d_m}(x)}(q, 1)$, a non-empty set for all $i_{d_m}(x) \in \mathrm{GF}(q)$ is included along with the remaining considered subsets.

Finally, we are able to present, in Algorithm 2.15, the formal outline of the EMS. The second step in the CN processing (2.91) is a postprocessing step required to prevent the algorithm from diverging. Unless applied, some messages may grow to the highest numerical value. It can also be shown that the EMS converges to the MSA when applied to the GF(2) case[96]. It is noteworthy that the under parameters $(n_m, n_c) = (q, d_c - 1)$, the EMS converges to the LSPA proposed in[88] and thus the EMS follows $\mathrm{O}(M \cdot d_c \cdot (2^m - 1)^2)$. A binary tree representation allows for $\mathrm{O}(M \cdot d_c \cdot (2^m + (n_m - 1)m))$, while the recursion proposed in[46] attains $\mathrm{O}(M \cdot d_c \cdot n_m \cdot 2^m)$. Exposing parallelism to finding the configurations yields a complexity $\mathrm{O}(M \cdot d_c \cdot \binom{d_c - 1}{n_c} n_m^{n_c})$[96]. However, the authors final remark on the CN processing complexity is that, under (n_m, n_c) parameters which approximate the EMS to the SPA, the numerical complexity is bounded by $\mathrm{O}(M \cdot m \cdot d_c \cdot 2^m)$, which requires the same number of operations as the FFT-SPA formulation with a major improvement over the latter as the only numerical operation required is addition[96].

Corrected Extended Min-Sum Algorithm Similar to what is observed the binary case[24,98], the EMS naturally benefits from numerical corrections to it, due to its suboptimal nature caused by the messages overestimation. Simple techniques can be employed[85,96] to correct the sub-optimality shown by the EMS.

Factor EMS a factor $\gamma > 1$ is included in the VN processing.

$$L(\mathbf{m}_{np}^{(i)}[i_1, \cdots, i_z]) = L(p[i_1, \cdots, i_z]) + \frac{1}{\gamma} \sum_{n=1, n \neq t}^{d_v} L(\mathbf{m}_{pn}^{(i)}[i_1, \cdots, i_z]) \tag{2.94}$$

Offset EMS a factor $c > 0$ is included in the VN processing.

$$L(\mathbf{m}_{pn}^{(i)}[i_1, \cdots, i_z]) = \max\left(L(\mathbf{m}_{pn}^{(i)}[i_1, \cdots, i_z]) - c, 0\right) \Leftarrow L(\mathbf{m}_{pn}^{(i)}[i_1, \cdots, i_z]) > 0 \tag{2.95}$$

$$L(\mathbf{m}_{pn}^{(i)}[i_1, \cdots, i_z]) = \min\left(L(\mathbf{m}_{pn}^{(i)}[i_1, \cdots, i_z]) - c, 0\right) \Leftarrow L(\mathbf{m}_{pn}^{(i)}[i_1, \cdots, i_z]) < 0 \tag{2.96}$$

$$L(\mathbf{m}_{np}^{(i)}[i_1, \cdots, i_z]) = L(p[i_1, \cdots, i_z]) + \sum_{n=1, n \neq t}^{d_v} L(\mathbf{m}_{pn}^{(i)}[i_1, \cdots, i_z]) \tag{2.97}$$

Both the correction factor and offset can be computed by minimization of a cost function associated with the LDPC code under study. This may be achieved through DE studies[28,85,96,98].

2.6 Decoding Schedules

The decoding schedule is absent from the formalization of the LDPC decoding algorithms, or is at most implicit. The decoding success does not generally depend on how the processing phases are computed within a decoding iteration[99,100].

2.6.1 Two-phased Message-passing

Two-phased message-passing (TPMP) applies an update rule at a time for all nodes in the Tanner graph for the binary case, or factor graph in the non-binary case[60]. Essentially all nodes within a node type in the graph are updated before proceeding to the next node type. When all CNs and VNs have been updated, a decoding iteration has been executed. This type of scheduling is appealing for its simplicity. Dependencies of operations are not taken into consideration since the update of any given message will not interfere with the update of another message.

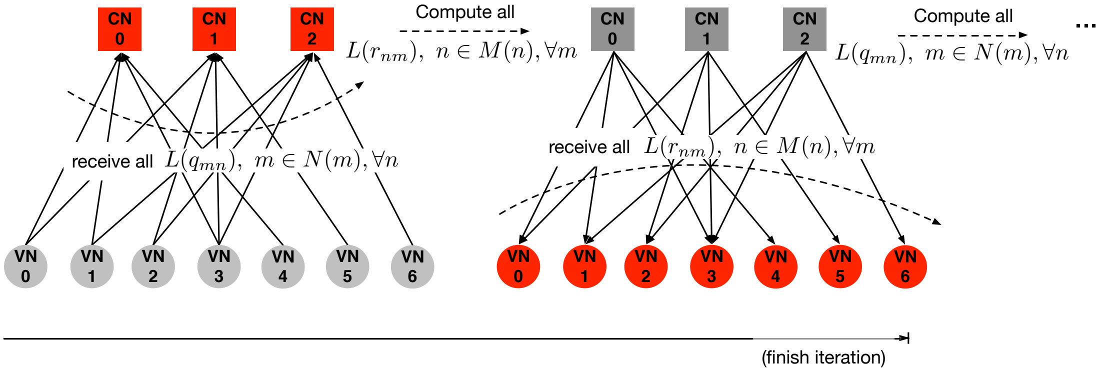

CN 0
CN 1
CN 2
Compute all
$L(r_{nm}),\ n \in M(n), \forall m$
CN 0
CN 1
CN 2
Compute all
$L(q_{mn}),\ m \in N(m), \forall n$
...
receive all $L(q_{mn}),\ m \in N(m), \forall n$
receive all $L(r_{nm}),\ n \in M(n), \forall m$
VN 0
VN 1
VN 2
VN 3
VN 4
VN 5
VN 6
VN 0
VN 1
VN 2
VN 3
VN 4
VN 5
VN 6
(finish iteration)

Figure 2.5: TPMP decoding schedule. Each type of node is updated in the same phase, i.e. all CNs are updated before any subsequent update is made to VNs and vice-versa, allowing all nodes in the CN and VN processing to be updated simultaneously.

2.6.2 Turbo-decoding Message-passing

Turbo-decoding message-passing (TDMP) is inspired in the Turbo-decoding algorithm[101], whereupon a node is made active and the processing phases are computed for the subgraph composed of the active node and adjacent nodes. Decoding can be performed in respect to either CNs or VNs. In Figure 2.6, the TDMP schedule is applied to CNs. At each time a node is active both the VN and the CN processing are computed, as it occurs in the TPMP case. However, there are advantages with the use of the TDMP scheduling in the convergence rate of the decoder, roughly, each iteration of TDMP is as powerful as two TPMP iterations[102]. Naturally, the TDMP exposes a limited level of parallelism as

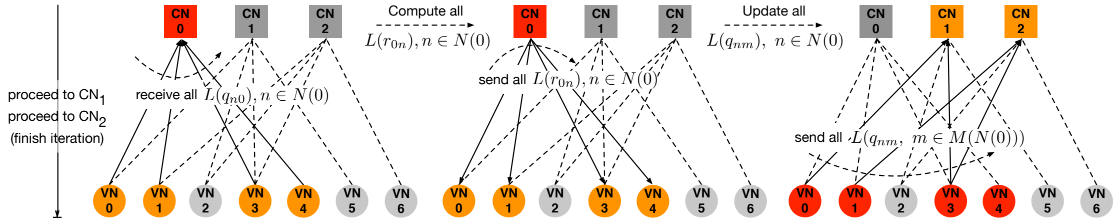


Figure 2.6: TDMP decoding schedule. Nodes are updated in a sequential fashion and also update their adjacent nodes. As seen, the decoding is performed by sweeping all the CNs and sequentially applying 1) first the CN processing to the active CN and 2) then the VN processing is calculated for the VNs adjacent to the active CN.

dependences can arise between message updates if instead of a single CN being active, two or more are. One way to schedule the maximum number of nodes to update simultaneously, while following the TDMP schedule is to find CNs such that the intersection of their $N(m)$ is an empty graph. In other words, so as long as the active CNs are not adjacent to the same VNs, the TDMP allows also for simultaneous update, and not just for the sequential update depicted in Figure 2.6. While finding such CNs can be a problem for unstructured codes, LDPC-IRA and QC-LDPC have been designed for maximizing the parallel potential of a TDMP schedule[61,103,104].

2.7 Overview of the Complexity

Based on the discussion written in this chapter, we are able to determine the suitability of certain decoding algorithms for, the task laying ahead, developing efficient binary and non-binary LDPC decoders.

2.7.1 Binary Decoding Algorithms

Among the binary LDPC decoding algorithms, the MSA is of particular interesting. Despite its sub-optimal decoding capabilities, exhibiting at times a degradation of over 0.5 dB[3], it conveys the least complex LLR-based message-passing algorithm. This is of use for both programmable and reconfigurable computing approaches. On the former, we can guarantee through the use of the MSA algorithm that our proposed decoders will not be instruction bound[105], and on the latter we ensure that reconfigurable computing designs will not consume too many *digital signal processor*s (DSPs), which are a scarce commodity on the *field-programmable gate array* (FPGA) die. Furthermore, it has been shown that the MSA can reach within the LSPA *bit error rate* (BER) performance at the cost of corrections that are numerically less intensive such as the case of the self-correction[3].

2.7.2 Non-binary Decoding Algorithms

The decoding algorithms surveyed in this Chapter are summarized in Table 2.1. In the second column the asymptotic numerical complexity is presented for the CN processing. The SPA and LSPA formulations exhibit the highest numerical complexities. The recursive computation of the former leads to a complexity scaling with the power of two of the field dimension, while the latter relaxes the numerical complexity scaling to the square of the field dimension. Clearly, this is too demanding either for field-decoders or for simulation purposes. The FFT-SPA, the LPFSPA and the MDA follow an equivalent numerical complexity. However, the MDA is more intricate as it requires successive domain changes to enjoy the benefits introduced by the FFT-SPA and the LPFSPA. The linear scaling with the field dimension and the field order presents better opportunities to capitalize in the development of decoders for simulation purposes.

Table 2.1: LDPC over GF(q) Decoding Algorithms Numerical Complexity Overview.

Algorithm	$O_{CN}(\cdot)$	LUT Accesses[a)]	Decoding Loss	Special Requirements
SPA	$M \cdot d_c \cdot (2^{2^m})$	$1536M$	–	GF(2^m) arithmetic
LSPA	$M \cdot d_c \cdot (2^m - 1)^2$	$490M$	$\sim$ 0dB	GF(2^m) arithmetic
FFT-SPA	$M \cdot d_c \cdot m \cdot 2^m$	0	0dB	FWHT
LPFSPA	$M \cdot d_c \cdot m \cdot 2^m$	$21N + 288M$	N/A	FWHT and GF(2^m) arithmetic
MDA	$M \cdot d_c \cdot m \cdot 2^m$	$96N$	$\sim$ 0dB	FWHT, log-*pmf* change
LFSPA[b)]	$M \cdot d_c \cdot 2^m$	0	0dB	FWHT
EMS[c)]	$M \cdot d_c \cdot n_m \cdot 2^m$	0	0.1 $\sim$ 0.2dB	GF(2^m) arithmetic

[a)] As reported for GF(2^3); [b)] Reported for a $(2, d_c)$-LDPC code, but the VN complexity increases to $O_{VN}(M \cdot d_c \cdot 2^m)$.;
[c)] Best reported by Declercq[96].

Furthermore, the CN processing complexity can be further downsized. The LFSPA does so by transferring part of such complexity to the VN processing. By doing so, the linear dependency of the field order is discarded and thus it is the surveyed algorithm presenting the lowest complexity. The EMS profits from the configuration set concept by using only the most significant messages. This ensures that instead of the field order dependency is effectively replaced by the configuration set parameter n_m. This is particularly helpful towards simplifying the decoding process, especially for high-order fields[96]. These two latter algorithms present the best opportunity to not only develop multicore decoders for simulation purposes, but also for the ultimate goal of developing a software decoder for real-time usage. Unlike some of the binary decoding algorithms, which trade decoding performance for lower numerical complexities, the EMS is the only non-binary decoding algorithm doing so, at a reported negligible loss of 0.1 $\sim$ 0.2dB when compared to the SPA.

2.8 Summary

Two class of decoding algorithms stand out in the analysis afore performed, MSA-based decoders for binary LDPC decoding and the FFT-SPA for the non-binary case. In the binary case, typical soft-decoding algorithms are based on a likelihood representation whose domain allows for certain numerical approximations. The MSA is particularly appealing due to its lower numerical complexity. As the LLR-domain is utilized, only additions need to be performed, and to address the BER degradation arising from the sub-optimal approximation of the CN processing (c.f. Algorithm 2.5), corrections can be made that mitigate such degradation at negligible complexity increase

The non-binary case is a particular one, as the generalization of the decoding algorithms to GF(q), and in the special case of GF(2^m) that we assume, allows us to divide non-binary algorithms into two categories. Those that require arithmetic operations over the Galois field, and those that do not. The former introduce a very high overhead, as arithmetic operations in GF(2^m) come hand in hand with very high CN processing complexity. The latter are, needless to say, of greater interest as we can dismiss Galois field operations from adding overhead to the decoding algorithm. Whereas the generality of SPA-based algorithms requires arithmetic operations over GF(2^m) or attempts at reducing the numerical complexity by changing the *pmf*-domain to a log-*pmf*-domain, the algorithms that change the *pmf* domain to a Fourier benefit from two factors 1) all existing convolutions become multiplications and 2) explicit GF(2^m) arithmetic operations can be altogether relished in favor of simpler operations, such as the case of the FFT-SPA decoding algorithm[47].

From the surveyed decoding algorithms it is clear that under binary LDPC decoding the MSA conveys the algorithm of choice. Not only does it provide a good tradeoff between complexity and BER performance, but can be improved with regards to attained BER by applying one of many corrections. Secondly, under non-binary LDPC decoding, the FFT-SPA stands out due to the reduced complexity of its arithmetic operations, forgoing any calculations performed over GF(2^m), and reducing the decoding procedure to Hadamard products applied at the CN and VN update phases.

3

LDPC Decoder Architectures Overview

Contents

In this chapter, the state-of-the-art of *low-density parity-check* (LDPC) decoders on programmable[166,167] and on reconfigurable computing architectures[56] is surveyed. While we can only assume that *central processing unit*s (CPUs) are the first choice for code study and *bit error rate* (BER) performance evaluation through Monte Carlo simulation, the majority of LDPC decoders found in the literature are not based on CPU architectures. With the odd exception, CPUs are mostly the underlying platform conveying proof of LDPC theory and concepts, but the decoder implementation is not the object of study, notwithstanding the fact that the advent of cross-platform parallel programming models and the growth in the register width of *single-instruction multiple-data* (SIMD)-vector units has given CPUs a high level of computational power[165].

On the other hand, most references found in the literature deal with LDPC decoders based on streaming architectures, namely, *graphics processing unit*s (GPUs) and other accelerators such as ARM mobile *systems on a chip* (SoCs)[168], Intel *single-chip cloud computer* (SCC), the Cell B.E.[169] or experimental stream processors—the latter examples are less prevalent than GPU-based LDPC decoders. One of the reasons for the GPUs popularity is due to the compromise between effort put into the development of a parallel algorithm that conveniently exploits the GPU *single-instruction multiple-thread* (SIMT)-architecture, and the corresponding attained performance. For the one, the development time between testing and prototyping, and the final optimized decoder ready for deployment is not as high and does not incur in too high *non-recurring engineering* (NRE) costs, as dedicated solutions do. On the other hand, this flexibility usually means diminished returns in the performance of the decoding solution, due to the fixed instruction set, memory hierarchy and underlying architecture that are not custom-tailored to the developed decoder. Furthermore, the introduction of data parallel programming models such as *Compute Unified Device Architecture* (CUDA)[170] and *Open Computing Language* (OpenCL)[171], timed with the unification of the graphics pipeline into a single programmable processor, meant that high-level productivity scientific languages such as C/C++, Fortran, Python and Ruby could be utilized, instead of protracted graphics languages. The drawback is that only with sufficient knowledge of the underlying GPU architecture will the developed LDPC decoders perform with high decoding throughputs.

Furthermore, as GPUs computing, due to their raw computational power (peaking in the TFLOPs range) and performance-to-watt-ratios orders of magnitude above CPUs, began to dip into the *high performance computing* (HPC) market[172], and to some extent on the datacenter market too, *field-programmable gate array*s (FPGAs) were evolving too. From their primitive "glue logic" status[173], they have since given rise to the very active field of reconfigurable computing[56,174–176]. They bring more throughput per silicon area[175] and less energy is consumed in the process than conventional processors[174].

Furthermore, due to the chip area of FPGAs, they usually accompany Moore's law technology nodes, while improvements on dedicated solutions, more often than not, fail to upgrade to faster, more efficient and smaller nodes in the same time-frame. Thus, the utilization of FPGAs as custom accelerators, usually designated as reconfigurable computing, addresses some of the issues surrounding the development of *application-specific integrated circuit* (ASIC) technology, but also set opened a whole new level of challenges purported by the availability of gate-level optimizations. Of particular interest to the work developed in this Thesis, is the use of *high-level synthesis* (HLS) models, which extend C/C++ and other programming languages[177–179], in a somewhat similar way that CUDA and OpenCL did for GPUs, thereby avoiding the drawbacks of VHDL and Verilog *register-transfer level* (RTL)-descriptions to generate circuits.

In the previous Chapter, the characteristics and complexity of the LDPC decoding algorithms was surveyed for codes defined over binary and non-binary fields. The $O(\cdot)$ numerical complexity presented does not capture the totality of transposing the LDPC decoding algorithm into an efficient LDPC decoder operating on either programmable or reconfigurable architectures. We can enumerate a list of the most important challenges to overcome in the design of efficient and high-performance LDPC decoding solutions as follows.

i) the need to transform the node connections imposed by the Tanner graph into a suitable memory layout and efficient addressing problem, considering that in most cases, irregular access patterns will be imposed by the Tanner graph structure;

ii) the weighing of suitable ratios of arithmetic-to-memory-instructions which maximize the efficiency of the LDPC decoding for the underlying computer architectures;

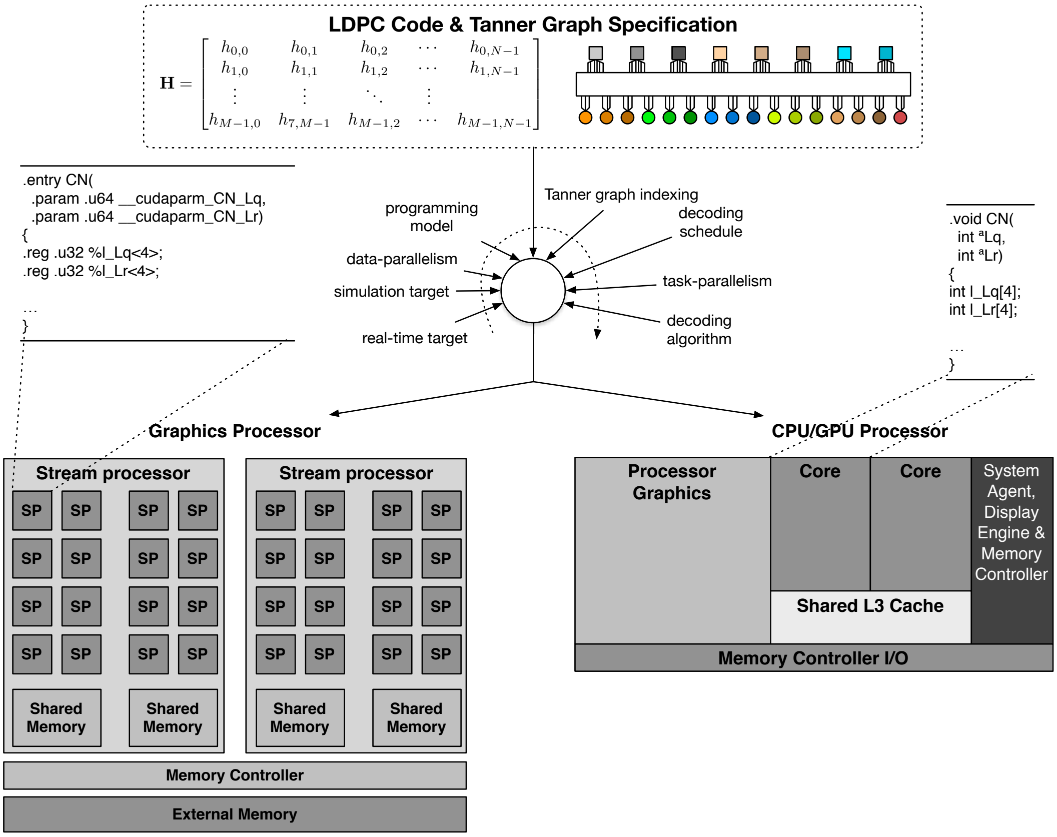
iii) the selected scheduling variants of the algorithm—*two-phased message-passing* (TPMP) or *Turbo-decoding message-passing* (TDMP)—influence on the aforementioned items;

iv) parallelism has to be explored at different levels of complexity depending on the architecture being programmed;

v) the complex exploitation of the memory hierarchy of multicore systems, or the complex problem of defining a suitable memory hierarchy for reconfigurable LDPC decoders.

3.1 Decoding on Programmable Architectures

In this section, the survey is focused on programmable architectures for LDPC decoder solutions. The most prevalent LDPC decoders found are GPU-based, then CPU-based, and finally, those based on streaming accelerators. A comprehensive list of the decoders found is tabulated in Table A.1 in Appendix A, highlighting key characteristics of the LDPC decoders, 1) task- and data-parallelism, 2) data representation, 3) LDPC code type and dimensions, 4) the indexing method of the messages circulating in the Tanner graph; and figures of merit of the LDPC decoder performance with respect to computational power, i.e., 5) decoding latency, 6) decoding throughput, both at a fixed number of iterations (when possible). Finally, the programmable platform is also identified. The following subsections are devoted to discussing the methodologies developed for defining efficient programmable LDPC decoders in light of their characteristics and design space constraints. A representation of the design space exploration characteristics and relations is illustrated in Figure 3.1.



$$\mathbf{H} = \begin{bmatrix} h_{0,0} & h_{0,1} & h_{0,2} & \cdots & h_{0,N-1} \\ h_{1,0} & h_{1,1} & h_{1,2} & \cdots & h_{1,N-1} \\ \vdots & \vdots & \ddots & \vdots & \\ h_{M-1,0} & h_{7,M-1} & h_{M-1,2} & \cdots & h_{M-1,N-1} \end{bmatrix}$$

```
.entry CN(
  .param .u64 __cudaparm_CN_Lq,
  .param .u64 __cudaparm_CN_Lr)
{
.reg .u32 %l_Lq<4>;
.reg .u32 %l_Lr<4>;

...
}
```

```
.void CN(
  int ªLq,
  int ªLr)
{
int l_Lq[4];
int l_Lr[4];

...
}
```

Figure 3.1: Tanner graph isomorphic mapping on programmable architectures. In the example above, the CN and VN functionalities are ultimately mapped as assembly instructions and with some degree of control, CNs and VNs can parallel process the decoding algorithm by utilizing distinct SPs or cores with a certain granularity.

3.1.1 Programmable LDPC Decoder Mapping

Due to programmable nature of the underlying computer architectures, a prototype isomorphic architecture[180] that is a direct mapping of the Tanner graph to *check node*

(CN) units, *variable node* (VN) units and the Tanner graph interconnection network is not truly possible[181]. Instead a programming description is required taking into account that the fixed underlying architecture and instruction set will require the sharing of computational resources. Only clever usage of the instruction set functionality and exploitation of the different regions within the memory hierarchy is guaranteed to optimize the LDPC decoders for performance and efficiency of computation[182]. While the term occupancy usually refers to GPU computing, the concept also extends to CPU architectures. Considering the limited but fixed number of logic arithmetic and memory resources available in programmable architectures, only if occupancy of the resources is high, will the performance of LDPC decoders peak. However, occupancy is a double-edged sword in the sense that it does not take into account over-utilization of resources leading to bottlenecks or deadlocks, that nevertheless keep occupancy high. Moreover, it is difficult to assess how efficiently a hardware resource is being utilized, based solely on the information provided by what authors have made available in their LDPC decoders on programmable architectures publications. Thus, we are left with figures of merit contextualized for the LDPC decoding problem, decoding throughput and decoding latency.

3.1.2 Tanner Graph Indexing Schemes

The LDPC decoder structure plays an important role on how efficiently the Tanner graph connections between nodes can be mapped. While regular codes might seem at first simpler than irregular ones, in practice they are not. Since the majority of the LDPC codes also keeps simplicity of encoding and simplicity of Tanner graph indexing in mind, standardized codes make use of systematic coding schemes, exploring *repeat-accumulate* (RA) parity sub-matrices, mainly for encoding purposes, and structured irregularity in the remaining portion of the parity-check matrix concerning the connectivity of *information node*s (INs)[183]. As discussed in Chapter 2, each edge defines two messages, traversing in opposite directions. When mapping the Tanner graph connections to a programmable processor, we must take into account that the messages have to be laid out in memory and, thus, their location index must be known with respect to both the CN and VN that define the edge. The memory index is usually not the same for messages traversing the same edge but in different directions.

Since the LDPC code parity-check matrix is also the adjacency matrix of the Tanner graph, any given LDPC code can be stored through matrix storage methods. Due to its sparsity, sparse matrix storage methods have lower memory footprints and can be employed for any type of code, and in fact they are (c.f. Tanner column in Table A.1). Regular codes, typically those constructed through *progressive edge growth* (PG) methods and made available in the Encyclopedia of Sparse Graph Codes[184] (Mackay codes), are stored most of the times in *compressed row storage* (CRS)- and *compressed column storage* (CCS)-like formats. This method of indexing the Tanner graph is shown in Figure 3.2. It is readily seen that out of the four memory accesses to q_{nm} and r_{mn} messages, or $\mathbf{m}_{cv}$ and $\mathbf{m}_{vc}$, two can be contiguous, a feature most of the surveyed decoders implement, as this exposes high bandwidth due to coalesced data accesses on GPUs and high cache hit rates on CPUs. The depicted scenario in Figure 3.2 defines reading accesses to be contiguous and writing accesses as non-contiguous. Thus, CNs require indexes relative to their adjacent VNs, and read a memory location offset from a *lookup-table* (LUT) (CN_{idx}), and VNs, likewise, read a memory offset from the other LUT (VN_{idx}). Essentially, CN_{idx} corresponds to the messages positions in memory for a reshaped column-wise parity-check matrix, and the VN_{idx} to its row-wise reshaped. As a consequence, the number of elements required to store the connections of a binary LDPC code Tanner graph is

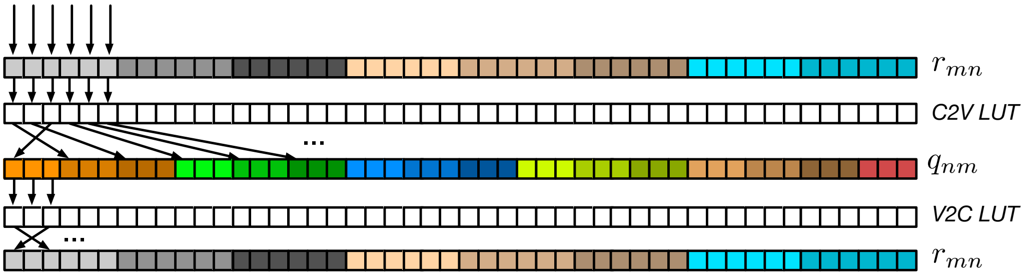


Figure 3.2: Tanner graph indexing based on sparse matrix storage. Different colors represent messages traversing edges connected to the same node.

$$TG_{sparse} = 2\times \sum_{i=0}^{M} \sum_{j=0}^{N} h_{i,j}. \tag{3.1}$$

This indexing method can also be employed for standardized codes[142], though the memory footprint of the Tanner graph mapping can be reduced by orders of magnitude[131,185]. In the particular case of *LDPC Irregular-Repeat-Accumulate* (LDPC-IRA) codes, such as those employed in the *2nd generation DVB* (DVB 2) standards, shown in Figure 3.2, the LDPC code Tanner graph is systematically constructed in a way that permits indexing using a barrel shifter approach[185]. For instance, the number of elements required to index the DVB 2 Tanner graph codes is:

$$TG_{DVB2} = 2\times \hat{d}_c \times r_f,\ r_f \ll N, \tag{3.2}$$

with r_f a code construction regularity parameter[61,185], and $\hat{d}_c$ the arithmetic mean of $d_c(i)$. This allows an on-the-fly computation of the memory locations to where each message reads and writes. In particular, q_{nm} messages will read and write to a contiguously increasing base offset, but writes will be shifted, and r_{mn} messages are read from an indexed base offset and written shifted while maintaining the offset. Hence, an address and a shift LUT, with a much lower size than the CCS and CRS sparse matrix storage (3.2) can be employed. The LDPC decoding methods exploring the Tanner graph structured construction, i.e. using the aforementioned efficient sparse matrix storage methods, are labeled as *structured* (Struct.) in Table A.1 and *sparse* (Sparse) if the first method is used.

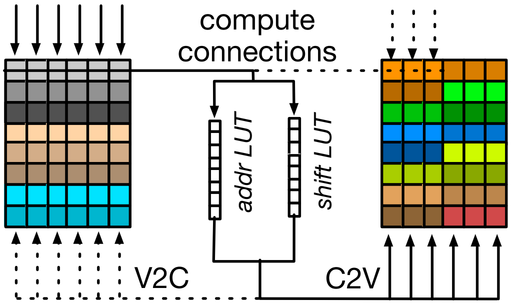


Figure 3.3: LDPC-IRA Tanner graph indexing based on sparse matrix storage.

Likewise, *quasi-cyclic LDPC* (QC-LDPC) codes can also be indexed by small-sized LUTs, as shown in Figure 3.4, by performing an on-the-fly computation of the memory location to where a message is sent after computation. This method in particular[131], defines contiguous reading of messages and indexed writing to memory, with a footprint that is independent of the LDPC code dimensions. In a way, the code protograph is sparsely indexed using the first method, though extra computation of indexes is required. The memory footprint of this method is

$$TG_{QC} = 2\times 3\times \left(\sum_{i=0}^{M} \sum_{j=0}^{N} f_{i,j} \neq \infty \right) + M_f + N_f,\ 0 \leq i < M_f,\ 0 \leq j < N_f, \tag{3.3}$$

with M_f and N_f the dimensions of the protograph matrix that generates the QC-LDPC code. The great advantage to this indexing scheme is that, regardless of the final codeword length which depends on the expansion factor z_f, the indexing LUTs size remain the same[131].

The memory footprint is not the most pressing issue in programmable architectures, as the memory addressing space size of modern CPU and GPU systems can be larger than the Tanner graph indexing memory footprint (3.1) (3.2) (3.3). However, the indexing method becomes a source for memory contention if for every computed message a memory index location requires loading, reducing the overall bandwidth to memory—it contributes to poorer cache hit ratios on CPUs[165], and adds further pressure to GPU memory engines[122]. As observed in Table A.1, the best performing LDPC decoders

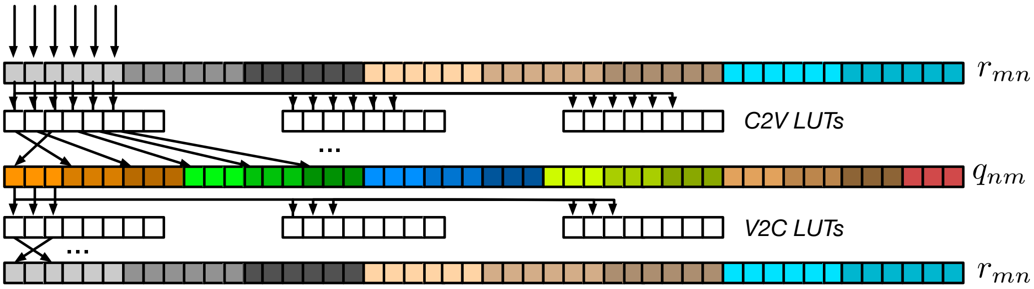


Figure 3.4: QC-LDPC Tanner graph indexing based on sparse matrix storage.

are those exploring structure sparse storage that exploit the Tanner graph structure, as opposed to a generic sparse matrix storage method. For instance, LDPC decoders implementing the former methodology achieve much higher throughputs than those exploring the latter.

Several parameters, other than the Tanner graph indexing, influence the decoding throughput attained, however, a clear trend is observed in this case. For all thread-parallelism techniques employed, only the *thread-per-codeword* (TpC) strategy overcomes the overhead in the Tanner graph indexing scheme[141], since the imposed overhead is negligible with regards to the amount of data moved in the GPU architecture. The remaining decoders see throughputs of a few to a dozen Mbit/s. Observing Table A.1, it is clear that only structured indexing schemes consistently see decoding throughputs in the hundreds of Mbit/s.

3.1.3 Programming Models

A prevalence of C/C++-based families can be observed throughout the surveyed LDPC decoders, adding to the popularity language under an HPC challenge such as the one concerning the development of LDPC decoders churning out very high decoding throughputs.

In particular, parallelism in CPU-based decoders has been exploited using the *Open Multi-Processing* (OpenMP) programming model in a number of decoders[110,111,138,145,146]. Therein, the strategy to extract parallelism is based on the automatic parallelization of computation loops wherein the CN and the VN processing are defined. This is achieved with appropriate OpenMP directives. A minority of LDPC decoders replace the functionality provided by the OpenMP loop parallelization directives with explicit thread-partitioning using POSIX threads[128]. Despite the usage of a lower-level *application programming interface* (API) to perform multithreading, the opportunity to improve on the decoding throughput is not fully captured by this approach. However, POSIX threads are the basis of the Cell B.E. SDK, which is a C-extended programming model[106]. Other authors choose to put upon the OpenCL cross-platform capabilities to use on CPU technology[186] making minor adjustments from the GPU-optimized decoder onto a computing substrate with lower parallel capabilities. Similar to the aforementioned OpenMP and POSIX threads strategies, the delivered throughput is limited by a number of factors, the most important of which is the OpenCL compiler ability to pack data elements within wide registers over which SIMD computation is performed[182].

Under this light, explicit utilization of SIMD-instructions is pursued on both x86 and ARM processors. The first have since evolved from their assembly-accessible 128 bit MMX registers[120,124,151] to the richer extensions provided by *streaming SIMD extensions* (SSE) and *advanced vector extensions* (AVX) at 128, 256 and 512 bits register width, though LDPC decoders in the literature exploit only 256 bit AVX registers[165]. As the instruction-set functionality of the SIMD extensions became richer, so did the abstraction concerning its use. While MMX required explicit assembly coding early at their introduction time, nowadays, MMX, SSE and AVX support high-level C/C++ intrinsics. On the ARM processors, the NEON extensions provide 64 and 128 bit registers, exploited through a set of C/C++ intrinsics[128]. SIMD computation also faces another challenge. The indexing of the Tanner graph connections renders data element packing and unpacking unavoidable[165]. This means that under MMX registers, a non-negligible overhead of data management housekeeping tasks offsets the performance gains enjoyed from SIMD computation. On the other hand, only the increased functionality and width of the SIMD units can guarantee higher performances, on a par with GPUs, due to the increase levels of data-parallelism purported by the data packing for SIMD execution[165].

On the GPU LDPC decoders side, apart from the seminal approaches utilizing streaming models[187] based on graphics programming languages[118–120] and early streaming computing models[187], the majority of LDPC decoders makes use of CUDA and a minority of OpenCL. While in certain aspects both programming models are alike, with many language traits and constructs referring to the same hardware features of the GPU processor, only with different designations and handles[171,188], CUDA popularity overwhelms that of OpenCL. On Nvidia platforms the reason is clear, as OpenCL is mapped on top of CUDA, with the performance of the former only reaching that of the latter at best[189]. Also, despite the similar ways to explore arithmetic instructions, data types and memory addressing spaces of both models, OpenCL is a cross-platform programming model, rendering superfluous management instructions and too verbose coding requirements not really necessary when cross-platform is not truly intended. Nevertheless, the surveyed decoders see no clear performance gap between the use of CUDA or OpenCL LDPC (c.f. Table A.1). CUDA-based LDPC decoders range from the inception of CUDA in 2007 to the latest stable version. However, majority of the LDPC decoders based on it explore only a limited subset of its features. In particular, more advanced features such as kernel self-calling and reconfiguring the GPU *execution grid* without the host CPU intervention (*dynamic parallelism*)[188] and advanced memory synchronization and fencing operations available to the whole thread execution grid are not explored in the surveyed works, though they address limitations identified in some of the works[112–114,149,164]. With this regard, the OpenCL specification suffers from a lower evolution pace, with features sensitive to GPU programming having to be ratified for inclusion in a cross-platform model also supporting CPUs and FPGAs, thus, OpenCL-based decoders are somewhat more limited to some features CUDA addresses[128,140,154,186]. Notwithstanding, as such features are not explored, there is no evident loss in choosing one instead of the other.

With more or less control of the underlying hardware, all of utilized programming models allow the development of parallel LDPC decoding algorithms. Parallelism is naturally exposed in these[99,190], but other parallel features pertain only to a certain algorithm expression, in its turn tightly coupled to an underlying architecture. This concerns parallelism at the task-level and at the data-level that are discussed next.

3.1.4 Thread-parallelism

Several parallelism strategies have been proposed on multicore CPU and GPU architectures that divide LDPC decoding tasks between concurrent execution threads. These strategies entail constraints to other important design features of the LDPC decoders, especially with regards to data-parallelism and also to the decoding schedule of nodes, which are on the following Subsection.

Taxonomy Due to the rich set of parameters explored by researchers in the development of programmable LDPC decoders, an appropriate taxonomy for LDPC decoding on programmable architectures will be introduced herein. First regarding parallelism, whereupon the nodes functionality, as seen in Figure 3.1, is translated onto task- and data-parallelism at certain granularity levels among physical or logical cores or between execution threads. To keep a low simplicity of taxonomy, we define it in terms of thread-parallelism, which is also in accord with the majority of the surveyed programmable LDPC decoders that take in thread-based programs.

***Pixel-per-edge* (PpE) Decoding** PpE is the oldest parallelism strategy dating from the time when *general-purpose GPU* (GPGPU) was at its inception, with graphics languages being the only way available to perform non-graphical computation on GPU processors. Back then, data elements had to be mapped onto “graphical data elements” in order to be processed by the pixel shaders, thereof stemming the designation of this thread-parallelism strategy.

While results seemed highly promising at a time when the only prospective way to achieve high-throughput would be to develop a dedicated hardware accelerator, they were still lacking the performance seen for the LDPC decoders under CUDA and OpenCL, once the graphics pipeline had been unified onto a single processor[170]. Approaches such as those proposed by Falcão *et al.* allowed decoding throughputs of dozens to hundreds of Mbit/s[118–120], combining a graphics language approach with Caravela streams[187].

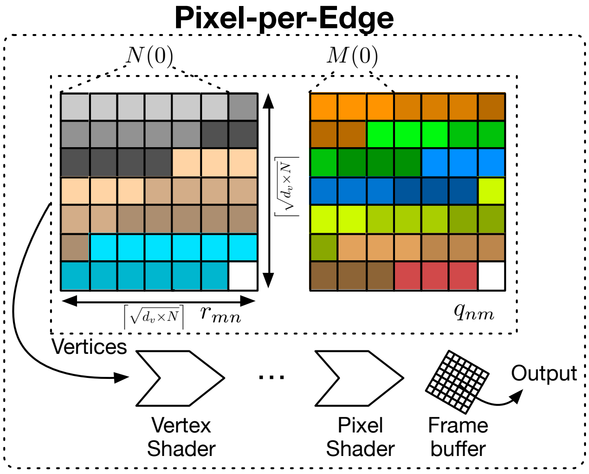


Figure 3.5: Pixel-per-Edge LDPC decoder thread-parallelism.

***Thread-per-edge* (TpE) Decoding** TpE corresponds to the strategy with the finest granularity, which brings upon the LDPC decoder designer a granularity tradeoff. For the one, if consecutive threads deal with the update of messages belonging to consecutive nodes in the Tanner graph, then there is a high exposure of both spatial and temporal data locality. For the other, most GPU-based LDPC decoders that exploit this granularity have been developed for pre-Fermi or Fermi architectures that do not have a caching mechanism available to threads for computation[170]—it exists only for off-chip memory transaction. For instance, locality is automatically explored by the x86 cache system in heavily SIMD-based LDPC decoders[165], leading to over 90% cache hit rates that maximize the LDPC decoder bandwidth. Alas, this is not the case in the multicore GPU-based decoders utilizing this strategy. Under the methodology proposed by Chang *et al.* throughputs peak lower than ∼2 Mbit/s for moderate length codes (816, 4000 and 8000 bits)[112–114]. This approach, implies one the most pressing use of threads within the GPU engine. For a regular LDPC code, the VN processing sees $N \times d_v$ threads spawned, and likewise the CN processing spawns $M \times d_c$ threads, implying the thread-parallelism granularity level putting the most pressure through the generation of thousands of threads, even though computation within each thread is not as heavy as the following thread-parallelism strategies.

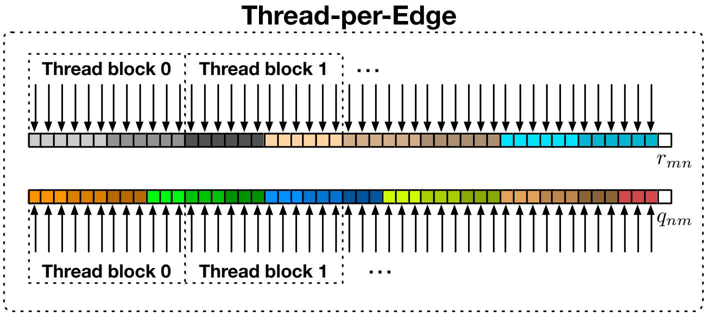


Figure 3.6: Thread-per-Edge LDPC decoder thread-parallelism approach.

***Thread-per-node* (TpN) Decoding** TpN is the most prevalent strategy, with a thread being spawned per node in the LDPC code Tanner graph. While this strategy quickly depletes the number of concurrent threads that can be active simultaneously on multicore GPUs for moderate to long block lengths, this pressure is not as high as in the TpE strategy case for short to moderate codes. One of the reason behind this strategy being by far the most popular strategy is related to its elegance. Each node in the Tanner graph can be assigned to an execution thread in the absence of a *de facto* isomorphic transformation into a *functional unit* (FU)[191].

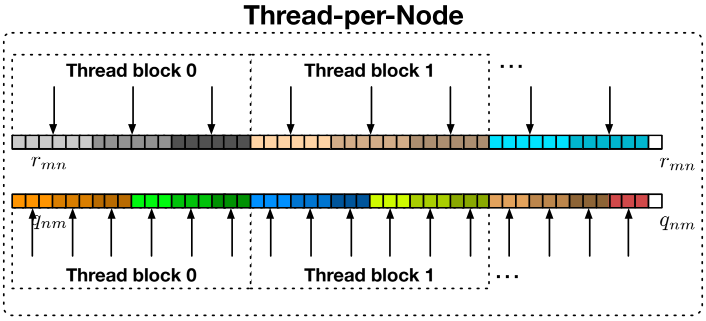


Figure 3.7: Thread-per-Node LDPC decoder thread-parallelism approach.

The first LDPC observed to utilize this thread-granularity was proposed by Falcão *et al.* for short to moderate length Mackay codes[184] (rate 1/2 1024, 4000 and 4896 bits) reaching up to ∼1.63 Mbit/s[123]. The decoder forcibly defined a 2-D texture mapping of the *log-likelihood ratio*s (LLRs) messages that contributes to the poor performance yielded[123]. Under a more general-purpose computing memory mapping, the authors were able to elevate the decoding throughputs to ∼87 Mbit/s for the normal frame

DVB-S2 codes[122], and to ∼40 Mbit/s for rate 1/2 Mackay codes (1024 to 20000 bits)[124]. The difference in the attained performance shows how data-parallelism design decisions and Tanner graph indexing methods are pivotal to elevating the decoding throughputs attained by GPU-based LDPC decoders. This is also verified by the work by Grönroos *et al.*, elevating their initial decoding throughput assessment ($<$ 1.80 Mbit/s)[129] to higher levels (157∼192 Mbit/s) for higher data-parallelism levels. The limits to the scalability of this approach were tested by Zaldivar *et al.*, using a TpN-variant defined over a 3-dimensional execution grid[142], for various d_c and d_v configurations. Chiu *et al.* report high latency times 0.2∼1.8 s for a short length (672 bits) 802.15.3c code[152]. Wang *et al.* propose a TpN decoder for 802.11n and 802.16e codes reaching 40∼52 Mbit/s[155]. However, other TpN approaches achieve only limited decoding throughputs[132,145,146], without a clear pattern to what lead to such low levels of throughput performance. Even more, when some had defined highly efficient Tanner graph indexing methods for cyclic and quasi-cyclic codes[130,131].

While LDPC codes working as the *error-correcting code* (ECC) basis of *forward error correction* (FEC) in communication systems imply an application agnostic operation, i.e., all bits being equally protected, their use for distributed video coding and quality of results is well-known[149]. On their application to video coding benchmarks (Hall Monitor, Foreman, Coastguard and Soccer), excellent frame reconstruction is obtained, though for offline video coding, i.e. it is not applied to real-time decoding[143,144,149]. Another application of LDPC decoding worth mentioning is their use for *quantum-key distribution* (QKD) reconciliation[192]. Mink *et al.* were able to reach high decoding throughputs, though for this purpose a lower number of iterations is required[164].

Finally, non-binary LDPC decoders that implement the TpN strategy have also been proposed[108,154]. Beerman *et al.* define a 3-dimensional grid of computation to properly exploit parallelism exposed at the *binary extension field* (GF(2^m)) dimension and by the scheduling of operations within the processing of the Tanner graph nodes. Under the proposed strategy, equivalent decoding throughput (∼2 Mbit/s) is obtained for GF(2^m) spanning the binary field (GF(2)) to GF(2^8).

Block-per-codeword **(BpC) Decoding** BpC is a strategy available to GPU execution as it is based on the concept of a *thread block*[170], a CUDA concept that finds its equivalent as *workgroup* in OpenCL[171]. The rationale stems from the *execution grid* composed of *threads* and divided onto blocks that permit a suitable exploitation of the memory hierarchy in multicore GPU architectures. Threads within the same block are allowed synchronization and fencing mechanisms for tighter cooperative computation as they can access the *shared memory* space, an addressing block physically unavailable to inter-block communication.

A strategy based on this granularity fails short of utilizing all the GPU *stream multiprocessors* (SMs), as the number of blocks required is independent of the code length. Hence, this strategy is usually accompanied by data-parallelism levels that spawn more blocks to decode more codewords in parallel[a)] throughout the remaining SMs of the architecture[106,133,159]. Notwithstanding, there is a constraint on how many threads can compose a block, that is influenced not only by the capabilities of the underlying hardware—high-end GPUs can execute blocks with a higher thread count than low-end ones—but also by how the developed decoding algorithm consumes registers and shared memory[170]. For QC-LDPC codes, it might make sense to define z_f threads per block as it matches the protograph expansion factor. However, this might be too low of a value, leading to poor resource utilization, or too high, preventing this strategy to be accommodated onto lower-end GPUs[159]. Equivalent tradeoffs can be seen for LDPC-IRA, random LDPC codes and non-binary regular ones[93]. This thread-granularity level is also the *de facto* strategy able to efficiently implement TDMP and TPMP decoding schedules, as opposed to the remaining strategies which are usually limited to the TPMP, as explained next.

First proposed by Abburi[106], this strategy has been applied to *worldwide interoperability for microwave access* (WiMAX) (IEEE 802.16e) codes, and also to *Wi-Fi* (Wi-Fi) codes (IEEE 802.11n)[159], for moderate to high decoding throughputs achieved (24.50∼160 Mbit/s), at relatively low latencies under 12 ms. This method has also been explored for the quick evaluation of a QC-LDPC construction method[133].

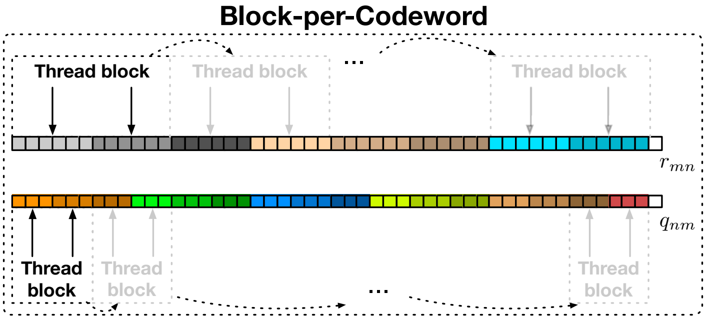


Figure 3.8: Block-per-Codeword LDPC decoder thread-parallelism.

***Block-per-node* (BpN) Decoding** BpN is, in a sense, a particular case of the BpC approach. Non-binary LDPC codes define another dimension exposing parallelism, the GF(2^m) dimension. This strategy is adopted for the particular case of non-binary LDPC decoding, but instead of defining a block of threads depending on the LDPC Tanner

[a)] Note that we adopt the designation codeword in a broader sense. Codeword can be the set of codewords that can fit onto the same data type, thus, it can be a single element or several ones packed into a vector data type[182,188].

graph regular features, it depends of the GF(2^m) dimension[147]. This approach is tested for a GF(2^8) code, yielding throughputs in the Mbit/s range (< 6) under different levels of data-parallelism for a pure sequential decoding schedule. Wang *et al.* also use this approach for a non-binary LDPC decoder for both OpenCL-based execution on CPU and GPU architectures. Though their approach has considerably low latencies (< 5 ms) it achieves only 1.26 Mbit/s at best for GF(2^m) dimensions of 2^2, 2^3 and 2^4.

***Thread-per-codeword* (TpC) Decoding** TpC is in all similar to the former approach, except that in the LDPC program description there is not the concept of a thread executing a codeword. For instance *core-per-codeword* (CpC) in an x86 CPU implies that one thread, corresponding to a logical core will execute the LDPC decoder in some of the physical cores. However, the program can be explicitly defined in terms of an executing thread, which decodes a codeword, hence, the distinction between two approaches would otherwise be blurred. Also, typically multithreading is explored to elevate the parallelism levels and improve the decoding throughput performances by exploiting a higher occupancy of the hardware. Namely, Abburi *et al.* propose a Cell-based LDPC where a thread per *synergistic processing element* (SPE) is assigned with the execution of the longest length rate 1/2 802.16e codewords, peaking at 270 Mbit/s[107].

Furthermore, other authors propose this approach for the multicore GPU architecture[137] and compare the performance of their approach to their previously presented LDPC decoder[160], reducing by one order of magnitude the time required to perform BER Monte Carlo simulation for a Mackay code[184]. Also, Lin *et al.* were able to achieve decoding throughputs in the range 212∼550 Mbit/s, though for high latencies (53∼421 ms) using short to long length codes (204∼20000 bits)[141]. Finally, Wang *et al.*, in order to assess the performance of the construction of a QC-LDPC convolutional code developed a TpC decoder peaking at 15 Mbit/s (using 768 to 1536 rates 1/2 and 2/3 codes)[161].

***Core-per-codeword* (CpC) Decoding** CpC is a thread-parallelism granularity that has no equivalent method in GPU computing, it is only available to CPU architectures. Herein, we consider the logical core definition of "hyperthreaded" processors, which defines a core as equivalent to an execution thread. Thus, a logical core will be responsible for executing a codeword or batch of codewords. However, the scenario under consideration is somewhat vaster here, as several approaches can be taken to implement this task-parallelism strategy.

For the upcoming exascale computing platforms[193], Diavastos *et al.* studied the scalability of LDPC decoders under 1) distributed and 2) shared memory model cooperative execution, and 3) shared memory model but not cooperative[117]. Regarding scal-

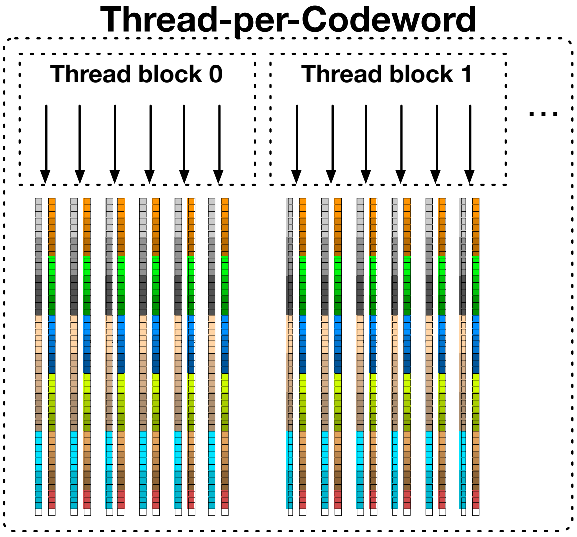


Figure 3.9: Thread-per-Codeword LDPC decoder thread-parallelism approach.

ability, 1) saw a reduction of the throughput to less than 1% of the single core baseline reference when all cores were committed to the computation, mainly due to high communication overheads caused by absence of caching mechanisms, 2) saw a sub-linear scalability of up to 11× when 48 cores were committed to the computation, while 3), saw 41× speedup when compared to the baseline[117]. Other approaches with regards to distributed computing involve the use of streaming accelerators applied to Mackay codes[184] and achieve moderate throughputs ($<$ 79 Mbit/s) for low latencies between 0.69∼1.53 ms[121,124]. 802.16e standard codes can be decoded at throughputs of 72∼80 Mbit/s under this methodology on the Cell B.E. processor[194]. Furthermore, this approach is also explored under mobile SoC platforms, whereupon short and normal frame DVB-T2 codes have been tested, reaching high latencies peaking in the 500∼2592 ms range at throughputs of ∼3 Mbit/s[127].

While some of the aforementioned LDPC decoders do not make use of vector processing[117,127], SIMD processing is a widely employed technique to improve high performance and efficiency in CPU architectures. Namely, the LDPC decoders based on the Cell B.E. make use of extensive SIMD-instructions by increasing the data parallelism within each core[121,124]. The work proposed by Falcão *et al.*[124] for their x86-based decoder is a particular type of CpC strategy. The OpenMP model was used to parallelize the computation inside the CN and the VN processing that were encapsulated by loops. Therefore, their true approach was Processor-per-Codeword, which in a sense is a special case of the CpC strategy[124]. Also, Intel CPU-based LDPC decoders are able to explore SSE and AVX SIMD-extensions to improve the data throughput while keeping latency at bay. Le

Gal *et al.*[165] proposed a CpC approach where several multiple codewords are decoded simultaneously by all logical cores in the processor, using the 128- SSE and the 256-bit AVX registers of the CPU to set the decoding throughput within 250∼560 Mbit/s, for CMMB, 802.11n, 802.16e and DVB-S2 codes. Furthermore, their approach is able to keep latency at bay, by keeping it under 10 ms in the majority of the cases, with 802.11n and 802.16e codes in the hundreds of μs range[165].

3.1.5 Data-parallelism

Data-parallelism expresses how the same operations can be applied to different data elements at the same time. Generally speaking, we can define it, with regards to LDPC decoding, as the number of codewords that are decoded simultaneously. The motivation for exploring data-parallelism is clear since short to moderate length codes cannot utilize all the resources that multicore processor architectures possess. Thus, to avoid wasting logic resources that would otherwise be sitting idle, several codewords are loaded and decoded simultaneously to elevate the decoding throughput. However, herein lies a tradeoff. Not only does the decoding throughput sees diminishing returns as the hardware occupancy is elevated, but decoding latency, a figure of merit of the computational performance that should be kept low, also increases. Therefore, only a handful of data-parallelism strategies elevate the decoding throughput to the desired high levels without sacrificing latency beyond admissible levels for real-time operation[154,165].

Taxonomy Similar to the thread-parallelism case, a proper taxonomy is due for data-parallelism within LDPC decoders on programmable hardware. Moreover, regarding data-parallelism the differences between methods that solely concern one type of processor but not the other do not exist. Each of the presented methods is exploited on both CPUs and GPUs alike. Furthermore, because data representation is tightly coupled to the design decisions regarding data-parallelism, it is herein discussed as well.

Codeword batch CPU and GPU memory engines are optimized for certain alignments. Memory transactions bandwidth can be increased by moving increasingly larger data types until the memory engine saturates at the maximum permitted alignment. Instead of storing an LLR using a `float` data type, a `float4` type can be utilized to store 4 LLR contiguously. In fact, data-parallelism strategies go even further and apply bit slicing operations, usually not natively supported by C/C++ languages, unless by SIMD intrinsics, to pack more data elements into a vector type. Considering the negligible BER performance loss when data is no longer represented using floating-point, but instead low bitwidths fixed-point types are used (typically between 4 and 8 bits[84]), an `int` data type can be employed to store 4 data elements and an `int4` 128 bit vector type[188] can store 16 codewords[182]. Single codeword and codeword batch storage is depicted in Figure 3.10.

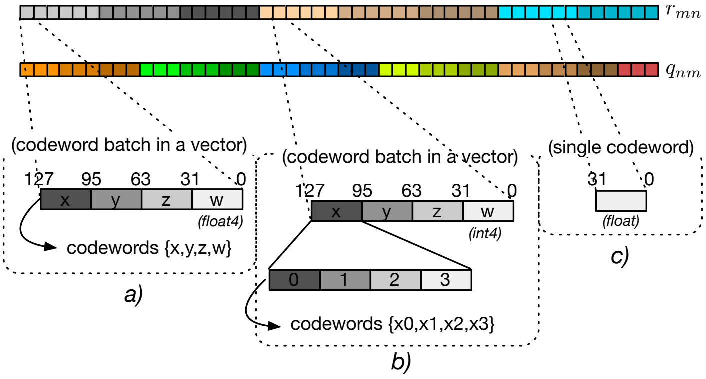


Figure 3.10: Single codeword and codeword batch on a vector. Data elements are stored a) several elements per vector type without resorting to bit slicing operations, b) several elements per vector type but bit slicing enables the packing of more data elements, or c) a single element is stored per data type.

When data-parallelism levels cannot be raised by increasing the number of elements in a data type, reducing each element bitwidth would hurt the BER performance, and going as further as defining a custom data structure will fail short of improving the bandwidth once the maximum alignment permitted is surpassed[182]. At this point, increasing the number of codeword batches must see the replication of the memory layout of the methodology pursued for a single data type, in one of two approaches possible.

***Padded* codeword batches** Under this approach data the basic level of data-parallelism within a batch is replicated as whole with a memory stride equal to the number of data type elements in memory. Thus, codeword batches become padded in memory, as shown in Figure 3.11, with the Tanner graph indexing method applied D times to D different base offsets to memory. This method does not impose any relevant constraint to the BpC, TpC or CpC approaches. In fact, throughput performance can only reach acceptable levels once data-parallelism levels are raised[127,137]. Using a TpN approach this means

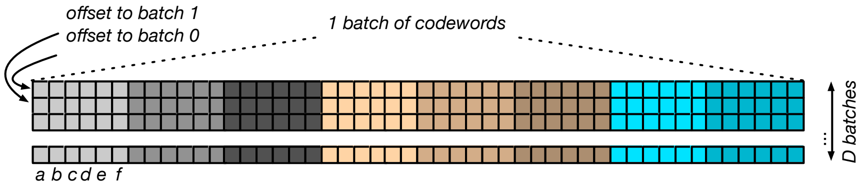


Figure 3.11: Padded data-parallelism approach. The Tanner graph indexing method is replicated for D codeword batches by padding copies of layout consecutively in memory.

the spawning of more threads to deal with the extra batches of codewords in the TpN approach[138–140,153,186].

One of the disadvantages behind this method is the inability to address unbalanced memory transactions that may occur when thread-parallelism does not account for threads having different memory access patterns. More data is accessed for higher CN and VN degrees. Thus, for irregular codes, there can be certain threads computing and moving a higher load of data than others. This problem is addressed by Kang *et al.* by evening out the accesses among threads in the same thread block[134].

***Interleaved* codeword batches** This methods defines D codeword batches as the data-parallelism level and interleaves data elements from different batches at basic data type granularity in memory. The advantage drawn here is that accesses are evened out to large blocks of data moved to and from contiguous locations, regardless of the Tanner graph indexing method. This method is highly suited for SIMD computation in x86 CPUs, with cache hit rations for short to moderate length codes reaching 90%[165]. Furthermore, this method is also highly efficient for GPU architectures, enabling real-time decoding throughputs and simultaneously real-time decoding latencies[154].

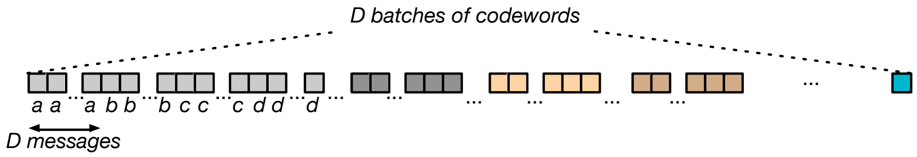


Figure 3.12: Interleaved data-parallelism approach. The Tanner graph indexing method is replicated for D codeword batches by interleaving data elements in memory.

3.1.6 Decoding Algorithms

Among the countless decoding algorithms, by far, the most popular ones are soft-decoding message-passing ones. In particular, LLR-based algorithms are adopted in the majority of decoders on programmable hardware. Floating-point types provide easiness of implementation and better overall BER performance, which is why it has been amply defined for the decoding algorithms discussed next. Even so, fixed-point datatypes are emulated, since they are not natively supported on CPU and GPU instructions sets, in most cases to improve the data-parallelism of the decoders. Exception must be made to the *impl.-efficient reliability ratio-based weighted bit-flipping alg.* (IRRWBF), which performs operations in the bit-state domain. The most favored choice for a decoding algorithm is the MSA in its uncorrected version, offset-corrected OMSA or normalized-corrected NMSA variations. SPA decoders in the probability domain, in the *pmf* Fourier domain

(FFT-SPA), in the LLR domain (LSPA), and in the signed-log Fourier domain (signed-log FFT-SPA) can be found, and also the odd Min-Max and *parity likelihood ratio algorithm* (PLRA) decoder, as tabulated in Table A.1.

The decoding algorithm choice can be tightly coupled to the data type representation chosen for a particular decoding design. Probability domain decoders use floating-point types, as they extensively rely on multiplication, with multiplication not supported natively on programmable hardware in fixed-point types. As a consequence, opportunities to improve the decoding throughput by increasing data-parallelism will be limited by this design decision. GPU hardware, usually aligned for 128 bit data types can pack only 4 floating-point words, while they can pack 16 fixed-point words with a bitwidth of 8 bits[106], and a similar trade-off is expected on CPUs, though they usually implement more sophisticated integer arithmetic than GPUs. As a consequence, all the SPA decoders explore single-precision floating-point (32 bits), though some MSA-based decoders also do so, the majority of them rely on 6$\sim$8 bit fixed-point data representations. This way, parallelism can be raised by increasing the number of words inside a data type defined by the programming model and language.

3.1.7 Decoding Schedules

As discussed in the previous chapter, the decoding of LDPC codes can be scheduled in two approaches, mainly. First approach is the so-called *flooding* or TPMP schedule. In this type of scheduling, the exposed parallelism lies at the complete dimension of the LDPC code, since all nodes can be schedule for processing one type of node at a time. Thus, all CNs can be updated at the same time, and all VNs can also be updated at the same time, provided that the CN and the VN processing is not concurrent. As a consequence, when developing a parallel programmable decoder, a memory fencing mechanism which prevents the scheduling for execution of nodes that are consuming messages from nodes which have not still updated their produced messages is required. Otherwise *write-after-read*s (WARs) hazards unfold. Notwithstanding, this is not particularly challenging to guarantee on either CPU, GPU or other accelerator devices, so as long as CN processing and VN process is defined by different functions or kernels. This way, the function or kernel call implicitly sets a synchronization routine preventing any WAR hazard. LDPC decoders using this decoding schedule (c.f. Table A.1) are among those reaching the highest decoding throughputs, since the TPMP schedule is usually accompanied by a heavily multi-threaded approach, usually TpN.

The TDMP schedule seen in the LDPC decoders that implement it are CN-based, i.e., CNs are scheduled for execution sequentially and after each CN is updated, their adjacent VNs ($N(m)$) are updated on-the-fly as well[195]. As this decoding schedule is applied to LDPC codes designed for the TDMP, such as QC-LDPC codes, this allows the execution of z_f CNs and their adjacent VNs simultaneously as it does not unfold any WAR hazard. The potential for high throughputs for this decoding schedule as been shown for both CUDA-enabled GPUs[106,139,140], the Cell B.E. accelerator[107] and a conventional Intel x86 CPU[165]—decoding throughputs range from 140 to 900 Mbit/s. Other approaches[133,135,136] fail short of such high throughputs, but are still in the same range as those obtained with the TPMP schedule. An interesting result is presented for a non-binary LDPC code case, defined over GF(2^4). The authors[147] study both a sequential and a TPMP schedule, based on the BpN approach. For equivalent BER levels achieved, the authors report lower throughputs (3∼8.5 Mbit/s) for the TPMP than for the sequential approach (5∼12.5) Mbit/s.

The TPMP, or flooding schedule, is the most widely implemented decoding schedule. However, a certain misconception may lie in the heart of this design preference. This type of decoding algorithm is the one permitting highest level of simultaneous scheduling of operations. All CNs can be scheduled for execution at the same time, and the same holds for the VNs, provided the execution of CNs and VNs does not overlap in time. On the contrary the TDMP, despited converging faster and reducing the number fo required decoding iterations to reach the same BER by roughly half, can only schedule a limited number of operations. If the LDPC code design has not been constructed with this scheduling in mind, there can be as little as no opportunity to schedule more than a single node at a time, though in practice this does not happen as the widely standardized quasi-cyclic codes are designed with this in mind. However, the TPMP schedule implies a data consumption/production pattern for each individual node where each message is accessed once per decoding iteration and per processing phase—for instance, a q_{nm} message is produced by VN$_n$ and is consumed by CN$_m$. This type of access pattern benefits little from a cache system. On the other hand, under the TDMP schedule, where data locality can be exploited temporally for short to moderate length codes[165]. For earlier GPU generations this advantage meant little, as there was no caching system, on newer models, L1 caches can exploit this feature of the TDMP schedule. In fact, among the surveyed LDPC decoders, the highest decoding throughputs found for CPU and GPU architecture is the TDMP[139,165].

3.2 Decoding on Reconfigurable Architectures

Efforts to survey the LDPC decoders developed for reconfigurable computing[56] would span out of scope of the work carried out in this Thesis. In particular, we refrain from dwelling into reconfigurable LDPC decoders that are not developed using HLS models, with, by far and large, the great majority of decoders found in the literature for reconfigurable computing developed using traditional RTL approaches, and as a consequence, a limited set of decoders fits in this requirement[196–199].

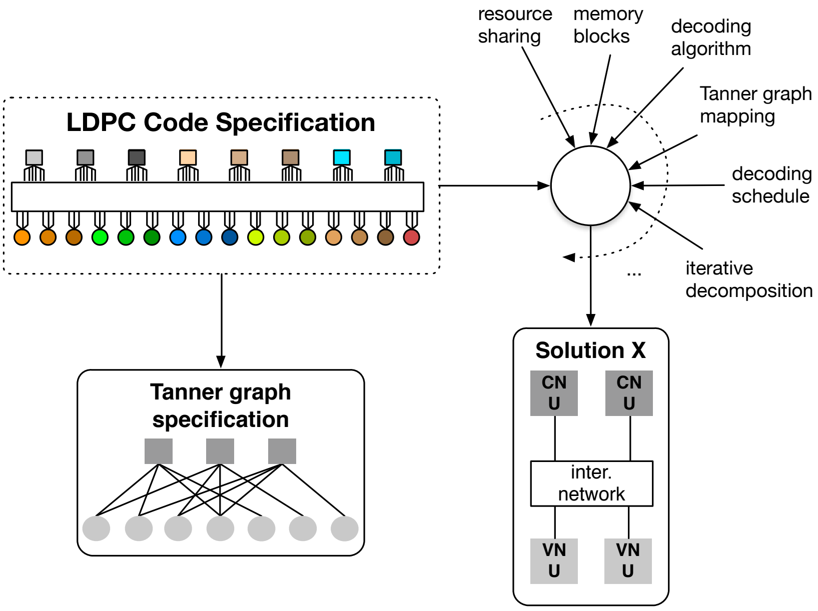


Figure 3.13: Tanner graph isomorphic mapping under a generalized reconfigurable computing approach.

3.2.1 Programming Models

OpenCL has recently become supported by the major FPGA manufacturers[177,200], the OpenCL programming model used for the development of an LDPC decoder[186] is the Silicon-to-OpenCL academic tool[201]. The tool takes in OpenCL kernel C descriptions, though not fully compliant to the OpenCL specification[171], and generates a custom wide-pipeline accelerator.

Moreover, the Vivado HLS[179] defines a comprehensive support for the C/C++ programming languages that get mapped onto circuits on the FPGA board based on a number of HLS directives that instruct how the tool should perform optimizations to different traits of the language. It supports optimizations to 1) memory blocks, 2) arithmetic functions, 3) dataflow directives for loops and functions, through pipeline or unrolling and 4) instantiation of certain IP cores in the C/C++ language for I/O interaction with other logic blocks[200].

3.2.2 Parallelism

Notwithstanding the fact that the OpenCL programming model defines the concept of work-items, a similar concept to execution threads, in the reconfigurable fabric, the generated accelerator defines no such physical nor logical element that is an execution thread. In fact, computation will be defined by the circuits configuration, thus while data-parallelism concepts remain perfectly valid, there is not thread-parallelism equivalent taxonomy to the reconfigurable LDPC decoders case.

Nevertheless, we are able to define the OpenCL LDPC decoder on FPGA, in its inception a TpN decoder, as a wide-pipeline decoder[186], and the Vivado HLS decoder as a wid-epipeline accelerator as well, though, this approach defined the TPMP node processing phases in computation loops[148]. As a consequence, we prefer the designation of loop-annotated decoder since it is through the optimization directives written as annotations (directives) to loops where computation occurs that the hardware generation is guided. Both approaches see modest throughputs of dozens of Mbit/s achieved for short to moderate length codes. The greatest advantage with this approach is the low latency, ranging < 3 ms in the OpenCL decoder case and < 500 μs in the Vivado HLS case.

3.3 Summary

LDPC decoders on programmable hardware can mostly be applied to simulation purposes, due to the methodology pursued in most of the literature be prone to increasing the decoding latency. Notable exceptions to this tradeoff, are the works of Le Gal[165] and Wang[156], which effectively keep latencies at low and real-time compliant levels. Notwithstanding, surveying the decoders in the literature compiled in Table A.1, we observe that the better suited strategies for LDPC decoding are based on LLR-based decoding algorithms, mostly defining fixed-point data representation. This allows for the packing of multiple messages with small bitwidths, usually in the 8 range, to be packed onto wider words. Furthermore, data-parallelism levels are usually pushed beyond the wide word, or vector datatype, granularity, often at the expense of spawning more threads in the decoding underlying architecture. Task-parallelism employed in the literature is explored at all conceived levels, from coarse (CpC) to fine-granularity (TpE), although the strategies attaining the highest performance are mostly fine-grained ones. In particular, the TpN task-parallelism granularity has scored the most prevalent method to expose parallelism for computation.

Regarding LDPC decoders in reconfigurable hardware, the surveyed LDPC decoders on HLS programming models show that this field provides interesting prospects, but remains a larger untapped field. In particular, it remains unclear how to best direct an HLS compiler to generate efficient hardware[202]. The incipient maturity of the tools used in the LDPC decoders[148,186] already attain competitive decoding throughput and latency, as observed during the inception of LDPC decoding on programmable multicore architectures. Furthermore, other programming models such as the Altera OpenCL, more recent versions of the Vivado infrastructure and the Maxeler dataflow decoders[178] promise much lower NRE efforts to target LDPC decoders with high throughputs and higher energy efficiency than programmable computer architectures[175].

4

Programmable LDPC Decoders

Contents

In the previous chapters we analyzed the *low-density parity-check* (LDPC) decoding message-passing algorithms most suitable for the LDPC decoding and surveyed the LDPC decoding solutions found in the literature. In this chapter, we will discuss methodologies for the realization of LDPC decoders on programmable hardware. To this end, we exploit the capabilities of multicore *central processing unit* (CPU) and *graphics processing unit* (GPU) processors, provided by data parallel programming models which help to keep the devised strategies under low *non-recurring engineering* (NRE) development costs.

4.1 Parallel Programming Models and Platforms

One of the reasons behind the slow adoption of LDPC codes had been the too low computation power, at the time of their inception, to realize LDPC decoders on computing fabrics that could tackle the volume of processing required. It is, thus, no wonder, that originally, programmable LDPC decoders have not been considered as viable, even at the time when LDPC codes were rediscovered[203], since general-purpose processors could not deliver more than a few Kbit/s[119], not nearly enough for the data rates then required. Notwithstanding, as the cramming of more components of a chip continued to follow the trend set by Moore's law, a number of key factors lead to the forgoing of the single-core processor in favor of multicore ones. For the one, frequency scaling lead to unbearable power and thermal dissipation levels. Furthermore, *instruction level parallelism* (ILP) and complex cache systems drove the design complexity upwards for diminishing returns that compounded by the power and memory bandwidth walls dictated the end of the single-core processor[204,205]. With it, came the multicore family of processors, attaining more computational power and increased design flexibility—*high performance computing* (HPC)-oriented processors are designed differently than embedded multicore devices—at the cost of increased pressure on the compiler and software development. Exploiting efficiently all the resources provided by multicore hardware is a substantially more complex problem than previously had been attained for single-core processors. multicore technology requires the exploitation of parallelism in order to achieve high performance.

As illustrated in Table 4.1, multicore processors have evolved into systems with dozens to thousands of cores. In addition to supporting multithreading, as a mean to hide memory latency, they also provide large *single-instruction multiple-data* (SIMD) units for vector processing. These units have grown from its vectorized integer 128-bit registers available only through its assembly instructions (MMX), to single- and double-precision floating-point 128-, 256- and 512-bit registers available for manipulation through *streaming SIMD extensions* (SSE) and *advanced vector extensions* (AVX) intrinsics. Also, compilers automat-

Table 4.1: Overview of the performance of recent processors.

	Processor	Specifications[a)]	Performance[b)]	Mem. Band.	Purpose	Year
CPUs	Pentium 4E	1 core @ 2.4GHz	630 MFLOPs	N/A	PC	2002
	Dual Xeon E5 2687v3	20 cores @ 3.1GHz AVX2	788 GFLOPs	68 GB/s	Server	2014
	Xeon Phi 3120A	57 cores @ 1.1GHz 512-bit SIMD	710 GFLOPs	240 GB/s	HPC	2013
	Core i7 5930K	6 cores @ 3.5GHz AVX2	289 GFLOPS	68 GB/s	PC	2014
GPUs	Nvidia Tesla M2050	448 CUDA cores @ 1.15GHz	350 GFLOPs	144 GB/s	HPC	2009
	Tesla K20	2496 CUDA cores @ 806MHz	1125 GFLOPs	208 GB/s	HPC	2012
	Jetson TK1	192 CUDA/4 ARM cores @ 950/2300MHz	720 MFLOPs	N/A	Mobile	2014
	Radeon R9 295X2	2×2816 cores @ 1.01GHz	11264 GFLOPs	640 GB/s	Gaming	2014

[a)] Specifications can be procured on the manufacturers' websites.

[b)] The performance is based on the Linpack benchmark and was accessed in benchmark-specialty websites.

ically try to take advantage of data parallel operations and pack them onto vectorized units when available, but new parallel programming models have truly unleashed the potential behind these computing architectures, and in particular, the GPU architecture.

In just over a decade, the performance of the Pentium 4E has been superseded by today's equivalent i7 5930K with a staggering leap from hundreds of MFLOPs performance to hundreds of GFLOPs. Likewise, server and HPC-oriented CPUs such as the tabulated Xeon E5 and the Xeon Phi have reached within the performance levels of late 2009 GPU architectures. The latter, currently delivering ballpark levels of TFLOPs for both HPC- and gaming-oriented purposes. Furthermore, the Jetson TK1 architecture is playing the level of the Pentium 4E, for much lower power dissipation levels.

4.1.1 Parallel Computing Principles

The concept behind parallelism is not limited to computing solely, it is also part of our daily routines, and simply deals with how to manage the execution of tasks concurrently. Two fundamental types of parallelism can be defined[206]. 1) When multiple tasks apply at the same time, different operations to the same or to distinct data elements, we speak of task-parallelism, e.g., when a task is broken into multiple sub-tasks that can be performed concurrently. 2) When the same operations are applied to distinct sets of data, we refer to data-parallelism, e.g., the batch execution of K FFTs can be divided onto K parallel sub-tasks concurrently executed.

It is critical that task dependency be respected to ensure coherence of computation of the tasks. To this end, a task divided onto multiple sub-tasks and a *data-dependency graph* (DDG) can be drawn, as illustrated in Figure 4.1. Therein, three distinct cases are portrayed. In Figure 4.1a) 1, 2 and 3 are independent but functionally parallel, as well as 5 and 6, but they are data-parallel; in Figure 4.1b) tasks 7, 8 and 9 are dependent, and no dependencies of previous data elements exist, making them suitable for data-parallel pipelined execution; and in Figure 4.1c) tasks 10 and 11 are strictly dependent, with task

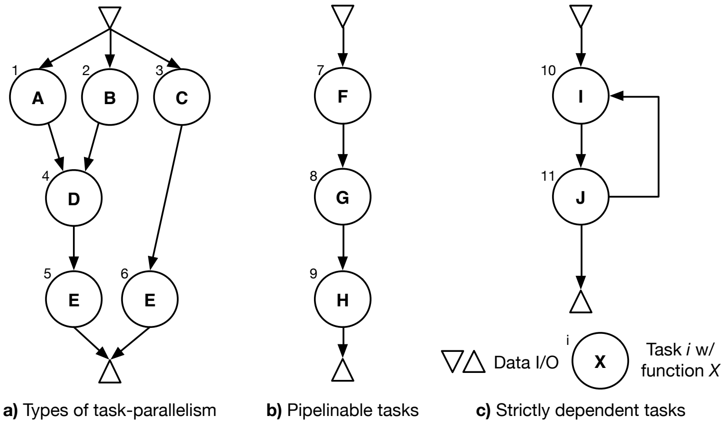


Figure 4.1: Data-dependency graphs.

10 depending for the processing of a new element on task 11. Based on the dependency analysis of the DDG, and on the ordering and function of the tasks, another critical aspect to efficient parallel computing is the frequency of synchronization mechanisms and communication between the different tasks. The communication pattern can define a task classification method into fine-grained, coarse-grained and embarrassingly-parallel. The latter represent the easier applications to parallelize since the workload can be divided onto multiple tasks without the need to modify the algorithmic steering. On the other hand, the true potential of parallel computing architectures lies in fined-grained algorithmic expressions, albeit exposure to synchronization and communication overheads arise with it, it is the most suitable approach to extract high efficiency out of modern parallel computing architectures[207].

While CPUs continue to scale their performance with the purpose of continuing to accelerate the performance of general-purpose systems, to some extent led by the personal computer field, and in the most recent years by the mobile industry field, the target has remained the same. With each new generation, the existing software infrastructure is expected to improve its performance. However, core performance is not significantly improved from one generation to the other with multicore technology. As such, substantial code refactoring and compiler reworking is expected to be made if one is to exploit efficiently all the cores in the processor. GPUs on the other hand, have evolved driven solely by the acceleration the performance of graphics-oriented tasks. Built with ample memory bandwidth and given the parallel features of the great extension of graphics primitives and algorithms, they evolved into highly parallel architectures, capable of processing thousands of vertexes and pixels simultaneously. As researchers began to realize the potential behind GPU acceleration, they soon realized that in order to tame it, more advanced parallel programming models were required, if *general-purpose GPU* (GPGPU) was to see the light of day.

Prior to the unification of the graphics pipeline onto a single scalar core[208,209], researchers exploited GPU programming through streaming models based on DirectX, OpenGL and Cg[187]. Certain authors introduced compiler extensions to the C language so as to better support GPU programming (Brook for GPUs)[210]. The downside of these approaches is the extensive control code required to handle the GPU device as an accelerator for computation. To ameliorate this situation, providing better GPU support by supporting a broadly used programming language with an appropriate *application programming interface* (API) to handle GPU devices and to describe GPU kernels that are able to efficiently exploit the raw performance of GPUs, data parallel programming models such as Nvidia's *Compute Unified Device Architecture* (CUDA)[170] and AMD's *close to the metal* (CTM) have been introduced. The latter has since been deprecated in favor of the *Open Computing Language* (OpenCL) programming model, that despite its cross-platform capabilities, shares many affinities to CUDA constructs and concepts at its inception, as it was mainly driven by GPU manufacturers on board with CPU manufacturers[211]. Since, the OpenCL programming model has seen support growing towards mobile architectures and *field-programmable gate array*s (FPGAs) alike. After the dawn of CUDA, and of the coming of age of the OpenCL[211], GPGPU acceleration through C/C++-based models gained traction, while at the same time, showing that under compilers suitably targeted for general-purpose programming languages, GPUs could unleash hundreds if not thousands of GFLOPs (c.f. Table 4.1).

Moreover, besides the CUDA and the OpenCL programming models, other parallel models exist, but for different purposes or targets. On multicore devices, parallelism, and in particular, multithreading can be manipulated through 1) POSIX `pthreads` at a very low level, explicitly manipulating `mutexes` and `semaphores` to guarantee coherent execution of threads[212], 2) Intel *thread building blocks* (TBB) allowing a higher level expression of parallel kernels through API calls[213] or 3) *Open Multi-Processing* (OpenMP) permitting a directive-based approach to exploiting parallelism with shared memory CPUs[214]. Under similar principles to that of OpenMP, *Open Accelerators* (OpenACC) provides a directive-based approach for GPU computing forgoing any API requirement. Also, *message-passing interface* (MPI) permits exploring multithreaded execution of a parallel kernel by spawning multiple processes on a set of computation resources[215]. While not particularly useful under single-processor machines, MPI, under one of its many implementations (MPICH, CrayMPI or OpenMPI) is an underlying backbone to computation within a distributed environment[172]. Furthermore, the Khronos Group has not remained with their hands tied to the OpenCL standard. Since its dawn, we have seen the proposal of *Standard Portable Intermediate Representation* (SPIR) aiming at a standardized language for the intermediate representation of parallel kernels and graphics[216], and also of SYCL, to go hand in hand with SPIR, providing a C++ interface to parallel programming with the intent of object containing both host and device code[217]. Finally, *Web computing language* (WebCL) has also been put forward as a model to explore parallel computing within the HTML5 capabilities of modern Internet browsers through CPU/GPU direct access via Javascript[218].

Clearly, a challenge must be overcome *a-priori* any realization of efficient LDPC decoders. With the advantage of hindsight, and forgoing the fact that some of these models have been launched during the execution of the work within this Thesis, we must make an informed decision regarding which models are most suitable for exploring efficient LDPC decoders on programmable hardware. Also, a remark must be made regarding the time of launch of a new programming model or specification, its support by a manufacturer, when models are not developed by the programmable hardware manufacturers, and the time to market of the products supporting it. Notwithstanding how promising the capabilities of a new programming model may be, considering that programmable architectures such as CPUs and GPUs have been around for quite some time, should be targeted using sound methods provided by programming models with broad support and that have proven their worth into reaching high efficiency of computation[209]. Hence, CUDA stands out for Nvidia GPUs in particular, with OpenCL just as worthy allowing also the targeting of AMD GPU devices, and also CPUs devices. Furthermore, to allow computation to be performed in a distributed computing system environment, we rely on the MPI standard. We discuss each architecture in conjunction with the programming models explored next. Naturally, with Moore's law incessant pace, and weighing constraints such as the execution time of this Thesis and the hardware availability, certain discussed architectures have since been superseded by newer generations.

4.1.2 General-purpose x86 multicore CPU

The current families of x86 multicore systems have begun to incorporate a graphics processor in its die in order to replace entry-level discrete GPUs. The number of CPU cores has been increasing from dual- to 10-core designs (from a mobile to a server environment), with HPC-oriented server on a chip devices such as the Xeon Phi containing up to 50 cores in a distributed system[193]. Additionally, several memory caching mechanisms can be found to address the widening gap between memory bandwidth and computational power[166]. In particular, at a certain level of the cache mechanism data can be shared among all cores and also between the CPU and the GPU devices. The Intel Ivy Bridge architecture illustrated in Figure 4.2 took this principle to a new level[b)]. As

[a)] Abbreviations in the graphics pipeline stand for: *command streamer* (CS); *vertex fetcher* (VF); *vertex shader* (VS); *hull shader* (HS); *domain shader* (DS); *geometry shader* (GS); *stream-out* (SOL)[219].

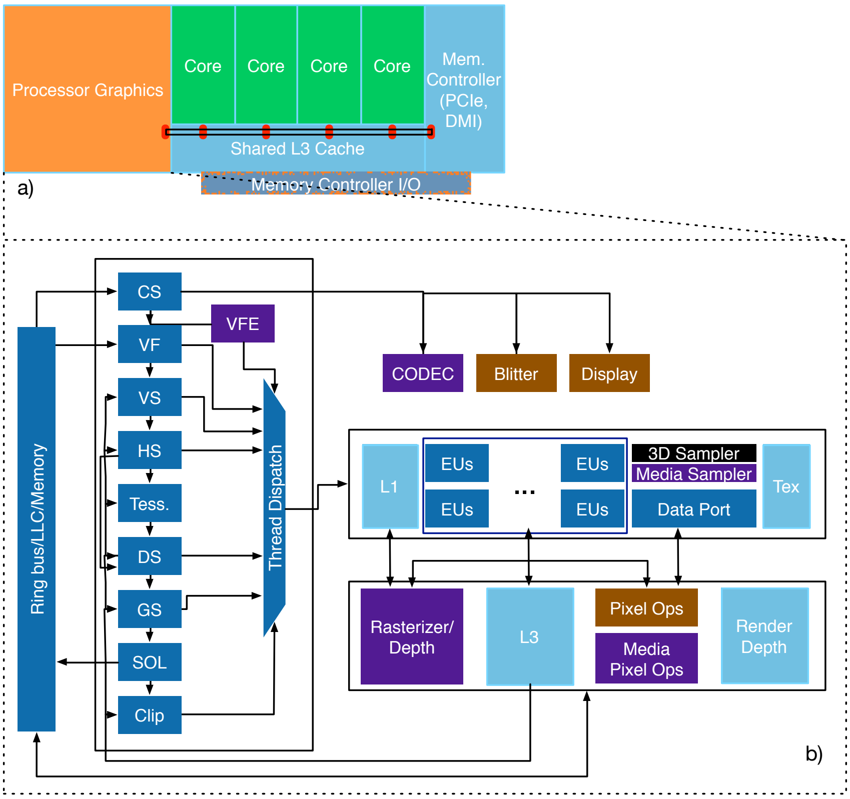


Figure 4.2: Ivy bridge CPU/GPU hybrid architecture: a) hybrid CPU/GPU system level diagram showing the shared access of the L3 cache by the CPU and the GPU; b) detailed GPU pipeline composed of multiple graphics stages$^{a)}$ which trigger the thread dispatch to enqueue every stream of computation on the EUs.

observed, both CPU and GPU can access data through the *last level cache* (LLC) (L3 in this case) allowing for joint execution of algorithms under appropriate programming models that define and allow synchronization events between both devices, to allow the expression of the types of parallelism overviewed by the DDGs in Figure 4.1. While *pthreads* and OpenMP can take advantage of the shared-memory architecture, they can only do so for multithreaded execution performed by the CPU cores. General-purpose computation on both devices, while not restricted to the use of OpenCL, since a separate programming model could be employed for the computation on the GPU chip, maintains a unified model to express the computation applied by the LDPC decoding kernels. Consequently, it is the programming model of choice to explore parallelism using both devices on the CPU/GPU hybrid die[219]. The literature overview in Chapter 3 reveals that the majority of LDPC decoders considers CPUs for data management housekeeping tasks, such as handling GPUs or other accelerators where computation occurs, or uses them as accelerators, fundamentally, through the OpenMP programming model. No references are made to cooperative execution using a CPU/GPU hybrid device.

4.1.3 General-purpose Computing on CUDA and OpenCL GPUs

The unifying of the graphics pipeline such that the pixel, vertex and geometry shaders share common resources in the GPU engine in combination with the introduction of an *instruction set architecture* (ISA) targeted also at general-purpose computation has made GPGPU truly possible[209]. In particular, we propose a methodology set in this chapter that is based on CUDA-enabled GPUs of the Fermi architecture generation[a] as illustrated in Figure 4.3. This GPU architecture is composed of up to 16 *stream multiprocessors* (SMs) each with 32 *scalar processorss* (SPs), also designated as CUDA cores[188]. Each SM has access to a *register space* composed of 32768 registers, to a *shared memory space* of 16 or 48 KB, depending on the L1 cache configuration of its polymorphic engine, to a 64 KB L1-cached read-only memory, to the L2-cached *texture memory* space, and finally, to the *global memory* space, typically ranging in the hundreds of MB to a few GBs. The detailed composition of each SM can be seen in Figure 4.4a) and its CUDA core decomposition in Figure 4.4b). Each of these SMs are equipped with their own *warp dispatch* units which schedule threads for execution on the GPU. Essentially, data-parallel processing is exploited by the execution of multiple concurrent threads throughout the available CUDA cores. The necessary control mechanisms for the correct and coherent execution of parallel kernels are available by the different settings allowed for the execution grid, i.e., the

[b] Since the introduction of the Ivy Bridge, Intel has released a tock (microarchitecture) upgrade to the Haswell and a tick (technology node) to the Broadwell family[220].

[a] Since Fermi, Nvidia has introduced the Kepler family (microarchitecture GK110) with its improved SMx, and also the Maxwell family (microarchitecture GM204) with the updated SMM.

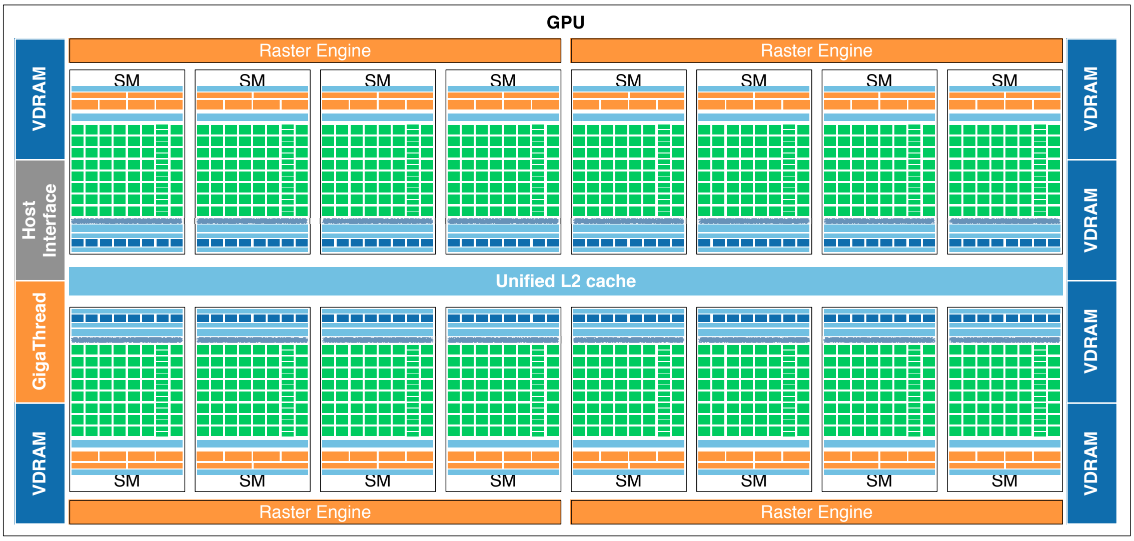
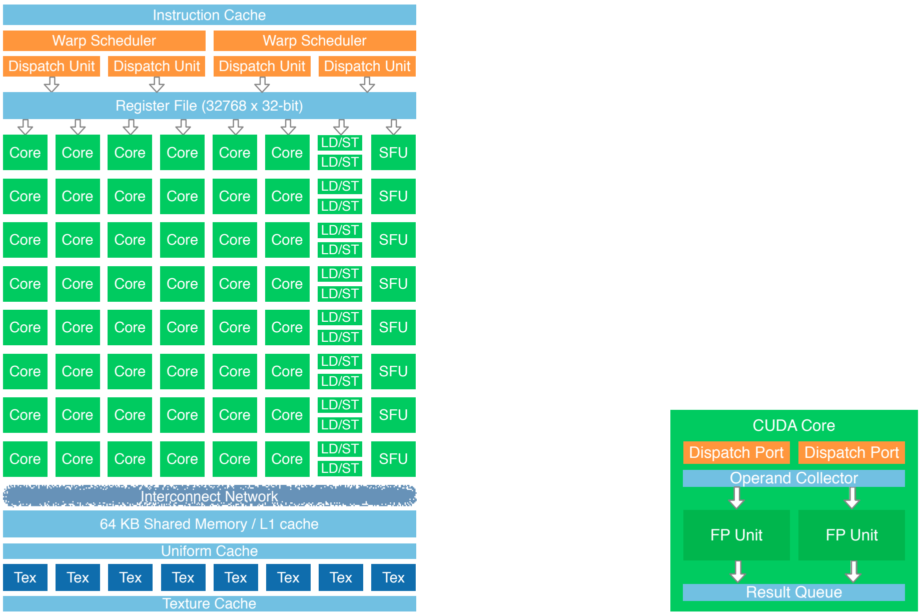


Figure 4.3: Fermi GPU overview

set of threads running a kernel, and through appropriate synchronization and fencing routines. The former, is divided onto multiple blocks of threads, each block executing independently of one another, thus allowing the scalability of kernels throughput the successive generations of CUDA GPUs—the GPU dispatches blocks of threads for execution, in *just-in-time* fashion, for the available SM[170]. The Fermi architecture is limited



(a) Fermi Stream Multiprocessor

(b) CUDA core (Scalar Processor)

Figure 4.4: SIMT GPU SM and CUDA core in detail: a) Fermi SM and b) CUDA core or SP.

to a maximum of $(1024, 1024, 64)$ threads per block across three dimensions (x, y, z), limits that are dynamically adjusted to the logic resources consumed by each thread in a certain kernel description[188]. Furthermore, the engine is restricted to spawning a maximum number of $(2^{16}, 2^{16}, 2^{16})$ threads per execution grid. The concept of the execution grid is standing above the physical scheduling of execution threads which is performed on a *warp* basis. Inside the GPU SM, threads are not executed loose from one another, nor are they subject to packing of data for the execution of SIMD instructions. Instead, they are subject to *single-instruction multiple-thread* (SIMT) execution whereupon a group of 32 threads are grouped in a *warp*. The warp is scheduled for parallel execution, with threads inside it completely synchronized. The downside to this approach is that divergent datapaths inside the *warp* are unraveled by the serial execution of every divergent branch in the datapath[188].

Furthermore, as illustrated in Figure 4.3, the CUDA-enable GPU device communicates with the host through a PCIe interface. The host is entitled to access the GPU global memory either through memory copies or by *direct memory access* (DMA) transactions if permitted. It controls the execution of kernels and spins upon their completion in a synchronous manner, or proceeds its execution flow if asynchronous calls are made. Thus, the global memory (VDRAM) is the memory space to which and from the host writes data, also to the read-only constant memory, and it requires meeting certain constraints for maximum bandwidth delivered. This leads to the introduction of the coalesced memory access concept. Since several threads are being served simultaneously by the GPU memory engine, when all the required data is packed onto the minimum number of transactions, the memory request is said to be coalesced. To put it simply, consecutive or strided accesses that are properly aligned allow for coalescing of memory requests, an important challenge to be overcome, since non-coalesced accesses typically incur in poor bandwidth due to the extra memory transactions needed to move the required data and multiple requests had several times the access latency (400∼600 clock cycles). Since general-purpose algorithms tend to have more complex memory access patterns than graphics-oriented algorithms, some of which may experience dynamic access patterns, the Fermi architecture is fitted with an L2 cache to improve the eligibility of memory access patterns for coalesced transaction[188]. It is clear that the aforementioned hardware details must be met by appropriate CUDA constructs that extend the C/C++ programming language or by appropriate functions that compose the CUDA Runtime API[188]. While the majority of the surveyed LDPC decoders in Chapter 3 is based on CUDA devices, the raw performance of AMD GPUs under OpenCL is similar to that of Nvidia cards, and these devices are discussed next.

The AMD GPU architecture illustrated in Figure 4.5 shows the Cypress microarchitecture of the Evergreen family[a)]. In a way, the Cypress and the Fermi architecture share similar ground for parallel processing, although, while the Nvidia is based on scalar proces-

[a)] The Evergreen family has been followed by the Graphics Core Next during the execution of the Thesis.

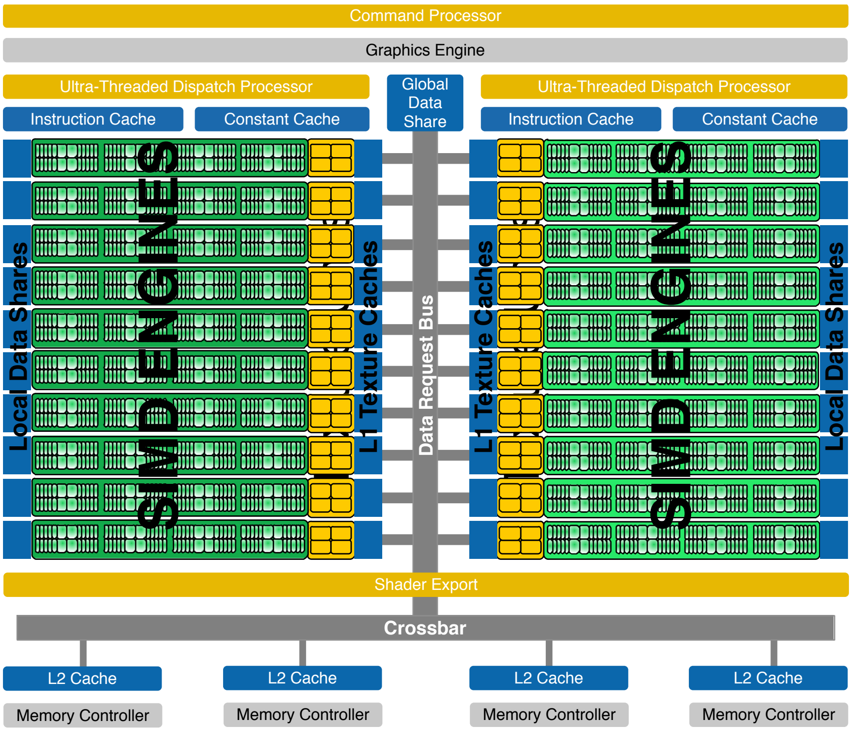


Figure 4.5: SIMD GPU architecture in detail: Cypress SIMD architecture with 4-way vectorized units.

sors, and thus implements a SIMT architecture, the Cypress architecture is SIMD-based. Instead of CUDA cores, *very long instruction word* (VLIW) processors are available to perform computation in a vectorized fashion. Each n-VLIW processors offers n slots for up to n data elements be issued with the same instruction concurrently (SIMD). Clearly, the VLIW is capable of providing more computational power than its scalar counterpart CUDA core. However, it will only do so if packing ratios close to 100% are achieved. In essence, instead of just leading to serialization within the *warp* execution, divergence, and also data dependencies, can lead to less than n slots filled at any given time. For an average m instructions packed for a parallel kernel, the equivalent packing ratio is then m/n, which reaches its sweet spot for optimal GPU efficiency nearing the 90%[221]. Each *thread processor* (TP) is a 5-way VLIW, providing 4 *arithmetic and logic unit*s (ALUs) and a single special function unit for more advanced operations. As a consequence, a 100% packing ratio can only be met if 5 data-independent instructions are issued at every available cycle. The depicted Cypress GPU is composed of up to 20 SIMD *computation engines*, each composed of 16 TPs and its own 32 KB shared memory and register space (omitting from the system level representation). Also omitted is the PCIe and global memory space, although they are present in a similar way as they are under the Fermi architecture. The main difference being that instead of DDR3, the Cypress family comes with DDR5 VDRAM for improved memory bandwidth[221]. Similar to the CPU/GPU hybrid processor case, the Cypress GPUs can target general-purpose computation through the OpenCL programming model.

4.1.4 Distributed Computing on multicore Fermi Dual-GPU Clusters

The rise of GPUs has also produced notable changes to the HPC market, not only did it inspire the development of architectures on the race to exascale computing[193], but they have also seen wide adoption across HPC cluster systems. As written in Table 4.2, GPUs-based cluster systems not only lead to extremely high-performances, but also to promising energy efficiencies, as attested both by the Titan and the ASUS ESC4000 systems[172,222].

Table 4.2: Top500[172] and Green500[222] first tier systems (June 2015).

	Rank	System	Configuration	No. Cores	RPeak[a)]	Power	Power Eff.
					(TFLOPs)	(KW)	(MFLOPs/W)
Top500	1	Tianhe-2	CPU: Xeon Phi	3,120,000	55,902.4	18,808	N/A
	2	Titan	CPU: Opteron, GPU: K20x	560,640	27,115.5	8,209	
	3	Sequoia	CPU: Power BQC	1,572,864	20,132.7	7,890	
Green500	1	Shoubou	CPU: Xeon E5	787,968	843.0	50	7,031.58
	2	Suiren Blue		263,168	384.8	28	6,842.31
	4	ASUS ESC4000	CPU: Xeon E5, GPU: K80	10,976	593.6	57	5,271.81

[a)] Peak performance using the Linpack benchmark[223].

Cluster systems are composed of multiple nodes, the majority of which devoted to computation, designated as *compute nodes*, and *control nodes* that control, administrate and perform data management housekeeping tasks. While each node can work independently of others as it contains all the necessary hardware and runs an *operating system* (OS), it is common that a single *master node* is used to submit execution jobs to the cluster system. In essence, execution is then triggered by the master through an appropriate interconnection network. Due to the HPC requirements, interconnection networks, e.g. *InfiniBand* (IB)[224] or Gemini[225], provide low-latency, high-speed optical connections to physically connect all nodes in the cluster. Then at the logical level an appropriate communication standard must be employed that allows the master node to access the compute resources at its disposal.

The MPI[215] standard defines a set of functions and directives for Fortran and C/C++ programming languages, that are available through appropriate MPI API calls in its multiple open-source and commercial implementations, e.g., OpenMPI, MPICH, CrayMPI. Not only does it provide the ability to run processes across any distributed compute resources that are connected, but it also provides synchronization and fencing instruc-

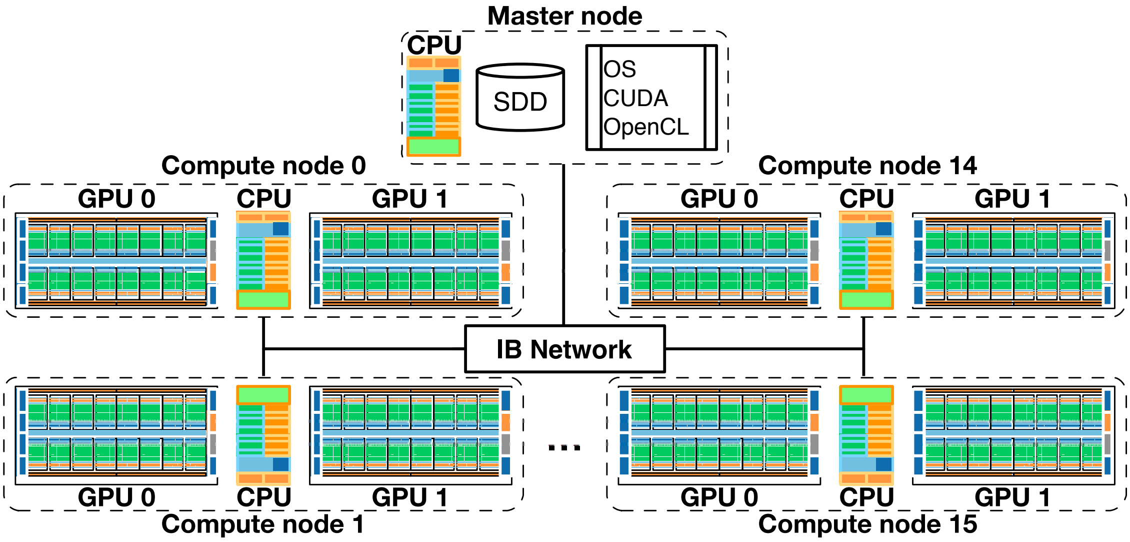


Figure 4.6: Dual-GPU cluster topology: a master node manages the 16 compute nodes, each composed of a single-CPU dual-GPU configuration. All disk I/O emanates from the master node through and the nodes are connected via an IB QDR network.

tions, in addition to common reduction, scatter and gather routines, commonly used in a distributed computing environment[215]. Since MPI does not define how computation is performed, it only provides a set of functions that allow advanced managing of distributed compute resources, the parallel kernels computation is expressed via other APIs, or even programming languages, called from a Fortran or C/C++ program. That said, considering an algorithm exposure to parallel computation many combinations of how parallelism is expressed at the compute node level and at the cluster level can be defined. For certain cases, a bottom-up approach is preferred, the parallel kernels are optimized for execution at the compute nodes and then data-parallelism is scaled by using multiple nodes in the cluster system.

The work performed using distributed computing systems in this Thesis was targeted at dual-GPU (Fermi) cluster systems, using a combined approach of CUDA for expressing the computation at the GPU-level, then computation was defined in the compute node for both GPUs, and finally wrapped in suitable MPI calls to perform computation across the cluster nodes. The topology of this system is illustrated in Figure 4.6. It comprises a master node that performs all management housekeeping tasks. The compute nodes are headless, thus the running OS is loaded from the master node. A considerable data I/O is generated by the parallel execution of processes due to the data lying at the master. Each compute node is equipped with a dual-Fermi GPU in addition to its own CPU.

4.1.5 CUDA Programming Model

Launched by Nvidia with the unification of the vertex, pixel and geometry shader onto common logic resources[170], CUDA permits the developer to describe parallel kernels that exploit fined-grained expression of algorithms using the GPU multithreaded execution. As previously said, many of the physical spaces and logic resources in the GPU engine see an equivalent logic element defined as an extension to the C/C++ programming language, either via appropriate qualifiers or added syntax, or through the CUDA Runtime API[188]. A kernel is defined using an extended version of C/C++ programming language, with limited support on the latter, using the appropriate qualifiers and is compiled offline, i.e., the CUDA executable will bundle the kernel binary with it. CUDA defines three key abstractions exposed to the programmer for GPGPU computing: 1) a hierarchy of thread groups in the execution grid, 2) a hierarchy of shared memories, and 3) mechanisms for synchronization.

Thread hierarchy In order to express parallelism, computation in a kernel is applied a number of times by a certain number of CUDA threads, that form a block of threads with a dimension defined by the programmer and a given number of blocks forms the execution grid, again, defined by the programmer. Hence, the block defines a coarse expression of parallelism, defining a fined-grained expression within. Each block is scheduled for execution inside a SM, and thus, threads in the same block have access to the shared memory space in the SM and to synchronization functions to exploit it. Outside the thread block level, inter-block cooperation is a feature not intended by the CUDA programming model[188], since blocks execute independently in order to ensure scalability across any CUDA device.

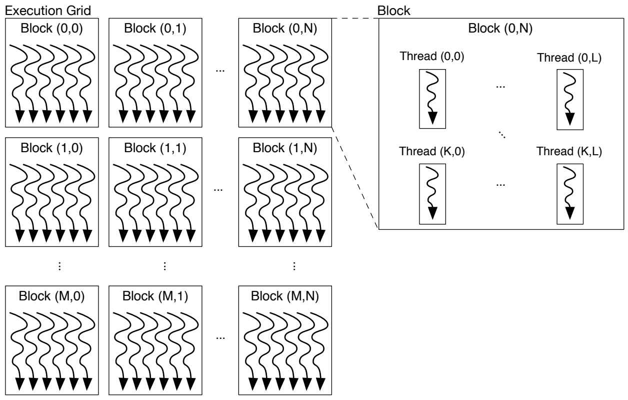


Figure 4.7: Execution grid: CUDA execution grid follows this nomenclature, OpenCL replaces thread with *work-item* and block with *workgroup*.

Memory Hierarchy A CUDA kernel has access to a number of memory spaces throughout its execution. Not only do they vary in memory size, but also in availability of access and allocated lifetime. They can lie off-chip: global memory, texture memory, constant memory; or on-chip: shared memory and register space. As illustrated in Figure 4.8, the threads in the execution grid can access the all the memory spaces with certain restrictions. Texture and constant memory are read-only memory spaces that are cached, and thus, provide fast access. All threads in the execution grid have read-write access to the global memory, as well as the host that can read and write data to the GPU via the global memory space, and can write data to the texture and constant memory spaces. The memory spaces lying within the SM at the physical level, which is to say the block at the logical level, show higher restrictions. Only threads within the same block can shared data through the shared memory space and the lifetime of the variables therein allocated is the lifetime of a block. The register space has a similar lifetime, only this space is private to each thread.

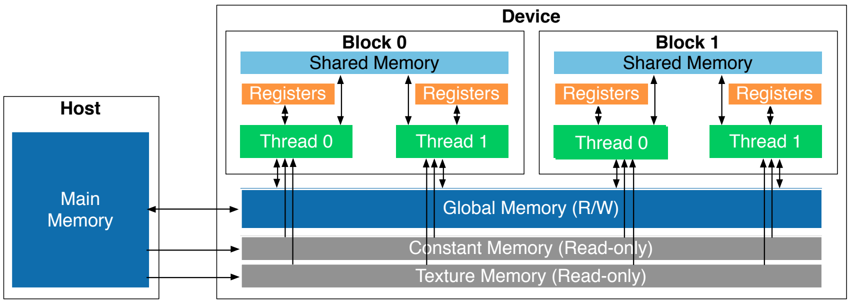


Figure 4.8: Execution grid: CUDA execution grid follows this nomenclature, OpenCL replaces block with *workgroup*, shared memory with *local memory*.

Synchronization Mechanisms CUDA provides a number of synchronization mechanisms at the thread-, block- and grid-level. Some of which are implicit, such as the aforementioned scheduling of threads within a warp, or the implicit barrier at the host side upon a synchronous kernel call. Others are explicitly defined by the programmer pertaining to synchronization or memory fencing instructions within a block. While the newer Kepler devices permit inter-block synchronization, the Fermi generation, employed in this Thesis, does not, and thus, we do not consider it in the remaining discussion of GPU-based LDPC decoders.

4.1.6 OpenCL Programming Model

Despite its cross-platform capabilities, the OpenCL data parallel programming model shares many resemblances with that of CUDA, in particular of its memory hierarchy. Due to its ability to target different types of devices, it provides an API that leads to much higher code verbosity due to the need to handle multiple kinds of devices. Furthermore, since it does not provide a compiler, but rather a compiler is provided by different manufacturers externally, compilation occurs online, during the host program execution. An increased layer of abstraction is provided, defining *platform*, *device*, *context* models, in addition to is *memory*, *programming* and *execution* models as shown in Figure 4.9.

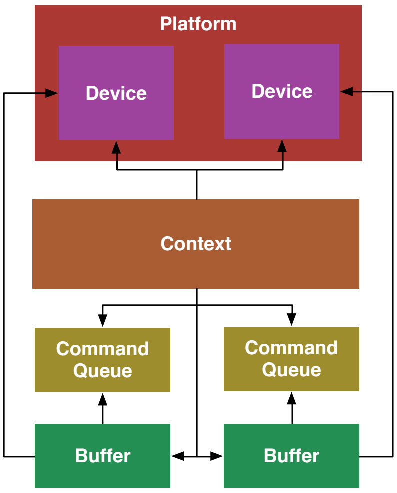


Figure 4.9: OpenCL abstraction model showing the logical dependencies and interaction between *platform*, *device*, *context*, *command queue* and *buffers*.

Platform, Device and Context models A platform is defined as a host system where multiple OpenCL devices lie. Although devices of different manufacturers are handled by separate platforms, devices of the same can be handled through a single-platform. Within the platform, devices are then handled by appropriate abstraction functions. The combination of platform and device are then grouped within a computation context that configures the OpenCL for the appropriate compiler call that generates the OpenCL kernel binaries. The ability to load pre-compiled binaries ensures that devices such as FPGAs kernels that undergo long synthesis, and placing and routing times can be loaded.

Memory Model The underlying OpenCL memory model is similar to that of CUDA, seen in Figure 4.8. In fact, this model is exported to all the devices, regardless of a physical resemblance of the architecture with the logical memory hierarchy. Thus, GPUs are closely matched, while CPUs see an elaborate translation of logical to physical spaces and FPGAs follow a completely different philosophy, as detailed in Chapter 5. The same restrictions previously described for CUDA apply, with the appropriate naming convention.

Programming and Execution Models The parallel expression of kernels is performed also using an execution grid that sees the issuing of a parallel kernel a given number of times across a certain number of *work-items*. These work-items are grouped in *workgroups* to compose the execution grid. Whereas in the GPU engine there is a one-to-one correspondence of work-item to thread, on CPUs each workgroup is assigned to a thread, again with a different philosophy for FPGA devices, as explained in Chapter 5. In addition, to the data-parallel expression allowed, it is also possible to define task-parallelism, though this renders each task-parallel kernel to be solely executed by a single work-item. Furthermore, the manipulation of the OpenCL device memory and execution is performed by the appropriate allocation of *buffers* to which a *command queue* is necessary to issue data transactions and to enqueue parallel OpenCL kernels.

4.1.7 MPI Programming Model

MPI defines a set of library and functions for use with Fortran and C/C++ programming languages that allow proficient handling of compute resources across distributed computation environments. As it often happens with general-purpose programming models that offer a wide spectrum of functionality to handle all the corner cases, we use a narrow segment of MPI functions. Thus, we can summarize the MPI model as illustrated in Figure 4.10. Therein, the master node is responsible for data management housekeeping tasks for the cluster that do not pertain with the acceleration of parallel algorithms. Thus, the master does not perform computation, it only instructs and handles how many MPI processes, or `ranks` exist in the MPI communicator `MPI_COMM_WORLD` that defines the pool of MPI ranks launched. To this end, it issues an `mpirun` command with the appropriate command-line arguments to spawn K processes across the cluster to specific compute

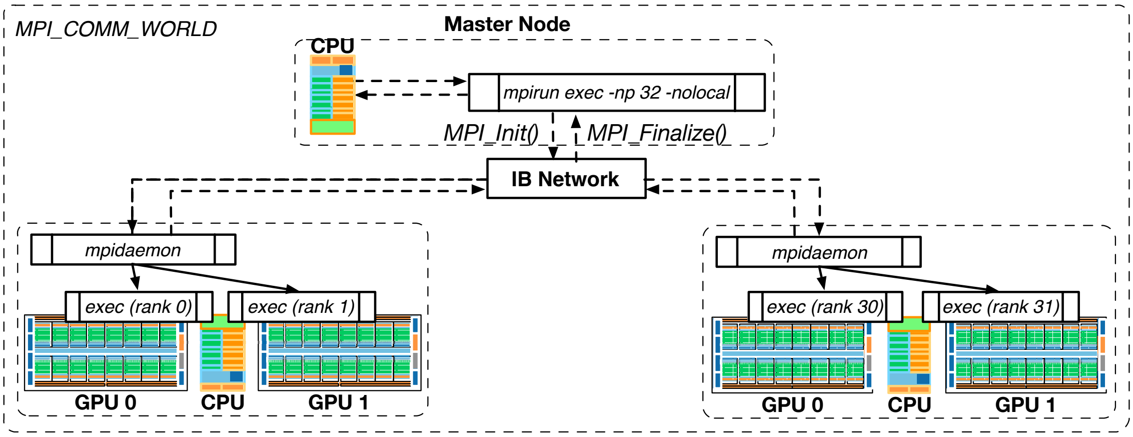


Figure 4.10: MPI basic hierarchy running on the dual-GPU cluster: `mpirun` spawns the MPI job `exec` with 32 jobs through the interconnection network; in the compute nodes, the executable is launched by an `mpidaemon` with a certain `rank` in the communicator `MPI_COMM_WORLD`.

```
int main(int argc, char** argv) {
  int num_elements = atoi(argv[1]);
  int num_trials = atoi(argv[2]);

  MPI_Init(argc, argv);

  int world_rank, world_rank_size;
  MPI_Comm_size(MPI_COMM_WORLD, &world_rank_size);       //Query number of ranks
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);            //Query this rank ID
  double total_my_bcast_time = 0.0;
  double total_mpi_bcast_time = 0.0;
  int *data;

  if(world_rank == 0)
    init_data(data);                                     //Initialize data

  //Broadcast data from rank 0 to all other ranks
  MPI_Bcast(data, num_elements/world_rank_size, MPI_INT, 0, MPI_COMM_WORLD);

  if(world_rank != 0)
    process_data(world_rank,data);                       //SPMD execution

  MPI_Finalize();
}
```

Listing 4.1: MPI program execution flow: the communicator `MPI_COMM_WORLD` is queried and, accordingly, the data workload is partitioned and broadcast from rank 0 for execution among the other ranks (running processes)[226].

nodes. In each compute node an `mpidaemon` that listens on the interconnection network launches the MPI ranks locally.

The control-flow is handled by the programmer at the `mpirun`-level and at the computation performed by each rank in the communicator, similar to what happens in CUDA and OpenCL. In this particular case, each two processes are given to a single compute node and each rank in the same node controls a GPU exclusively (under the dual-Fermi GPU cluster case). As previously discussed, the MPI controls the LDPC decoder at the distributed environment level, while the inner kernels performed by each rank will be the single-GPU decoders, that can be developed under any given programming model, but in this particular case CUDA has been chosen, As a consequence, the GPU-cluster is manipulated in such a way that *single-program multiple-data* (SPMD) computation is performed across its compute resources. In fact, as discussed in the next section, its utilization can be seen as a particular case of parameter sweep when Monte Carlo *bit error rate* (BER) simulation is therein performed[1,2].

4.2 Programmed LDPC Decoder Accelerators

To assess the performance obtained with the programmable LDPC decoders we utilize different LDPC codes and algorithms. The LDPC codes employed are characterized by different features that stress certain design choices made in the development of the

LDPC decoders 1) Tanner graph construction and 2) code length. They are tabulated in Table 4.3. The main figure of merit used in the LDPC decoders discussion is the decoding throughput that can be computed as

$$T_{dec} = \frac{No._{codewords} \times N}{t_{mem} + t_{kernel}}, \tag{4.1}$$

with t_{mem} the time taken to transfer data to and from the device where computation occurs, if applicable, and t_{kernel} the execution time of the LDPC decoding kernels.

Table 4.3: Experimental dataset utilized for the programmable LDPC decoders.

Dataset/scenario	LDPC Code							
	N (symbols)	N (bits)	Rate	m	d_c	d_v	z_f	Standard
I-a)	64800		1/2	1	{6,7}	{2,3,8}	N/A	DVB-S2
I-b)	64800		1/3	1	{4,5}	{2,3,12}	N/A	DVB-S2
II	1944		1/2	1	{7,8}	{2,3,4,11}	81	WiFI
III-a)	768		1/2	1	{6,7}	{2,3,6}	32	WiMAX
III-b)	1152						48	
III-c)	1536						64	
III-d)	1920						80	
IV	8000		1/2	1	6	3	N/A	Mackay[184]
V-a)	384	768	1/3	2	3	2	N/A	Non-binary[192]
V-b)		1152		3				
V-c)		1536		4				
V-d)		1920		5				
V-e)		2304		6				
V-f)		2688		7				
V-g)		3072		8				

4.3 Single-GPU Decoders

The potential for single-GPU devices to excel under the task of LDPC decoding has been identified in early 2008[227], and has since, continued to bear fruits[4,124,156]. While the target of the real-time decoding is still eluded in the majority of the LDPC decoder realizations using a single processor[142,156,165], the GPU remains an extremely useful platform for prototyping and validation of decoding algorithms and decoding schedules through BER simulation. Due to the GPU highly multithreaded execution we need to attend to expressing the LDPC decoding algorithms under the appropriate data- and thread-parallelism levels. In order to do so efficiently, the memory hierarchy of the GPU engine must be efficiently exploited so that high bandwidth of decoding is achieved. The following sections deal with the discussion concerning the many design features that compose the LDPC decoder space exploration under single-GPU execution.

4.3.1 Data-parallelism

Among the different types of parallelism exploitable, data-parallelism is the most straightforward to realize, but its potential is more limited than exploring thread-parallelism. Moreover, efficiently exploring data-parallelism is not a mere matter of pushing a very high workload onto the processor in order to try and maximize the occupancy of the logic resources provided. We need to keep in mind that there constraints posed by the memory hierarchy of the GPU engine. In particular, we should note that the memory hierarchy is composed of multiple addressing spaces, lying both on- and off-chip, with data flowing from off-chip regions to on-chip memory that is substantially faster. As previously said, the model entailed by GPU programming is of hiding memory transfers behind computation, and thus, increased data-parallelism should work towards this goal.

The first and foremost constraint of the LDPC decoder is to define a data-parallelism level which is compatible with the maximum alignment provided by the global memory that lies off-chip. Since a high access latency is always introduced, the memory requests should be maximally aligned so as to maximize the memory line access width. For instance, CUDA GPU devices maximum alignment is set at 128 bits, while for AMD devices this is set at 512-bit words. Under the CUDA and the OpenCL programming models, there is no support for arbitrary precision variables, which means that these large words are presented as vector types—essentially, they are aligned structures of C/C++-supported datatypes. Hence, an appropriate bitwidth for the *log-likelihood ratio* (LLR) messages must be found first. Then the messages need to be grouped into a vector datatype that maximizes the extracted memory engine bandwidth. In this case, this means pushing for the widest word that meets the maximum alignment criteria. Considering that CUDA devices are limited to 128-bits and AMD to 512-bits, the only supported vector datatypes that meet this width are `intn` and `floatn` (c.f. Listing 4.2). Since the majority of LDPC decoders in *application-specific integrated circuit* (ASIC)[228,229] and FPGA[185] are performed in fixed-point arithmetic at 5∼8-bit bitwidths, we define the basic LLR element at 8 bits. Then, since the maximum vector element is 32-bit wide

```
char2 vector_char;       //A vector with 2 char words
short4 vector_short;     //A vector with 4 short words
int3 vector_int;         //A vector with 3 int words
floatn vector_float;     //A vector with n float words
```

Listing 4.2: Supported CUDA and OpenCL vector datatypes: CUDA is limited to $n \in \{2, 3, 4\}$, while OpenCL defines $n \in \{2, 3, 4, 8, 16\}$.

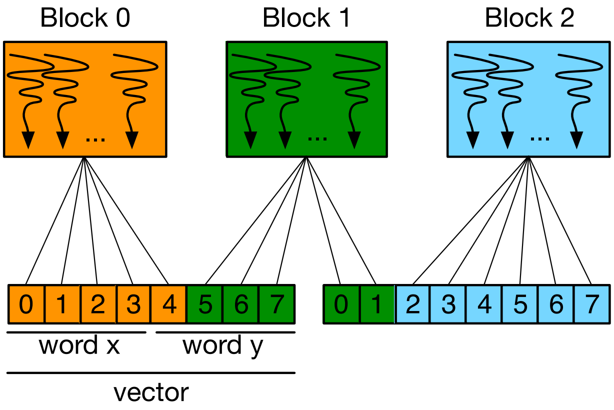
(it is an `int` or a `float`), we pack 4 LLRs into each vector element. This process is then repeated for how many more LLRs are required to meet the alignment. Words can be packed and unpacked cyclically, as shown in Listing 4.3, in a scalable strategy. For instance, it can be used for a single `int` type packing four 8-bit LLRs, or scaled to a `int16` wide word packing 64 8-bit words[182].

```
//A vector with 16 LLRs with int words {x,y,z,w}
int4 vector_llr;

//Fetch word x from vector_llr
int llr_x = vector_llr.x;

//Unpack LLRs cyclically from the word x
char llr_x_0 = llr_x >> 24 & 0x000000FF;

//Perform some manipulation with the data

//Pack LLRs cyclically to word x
llr_x = (llr_x <<  8 & 0x000000FF) | llr_x_0;
```

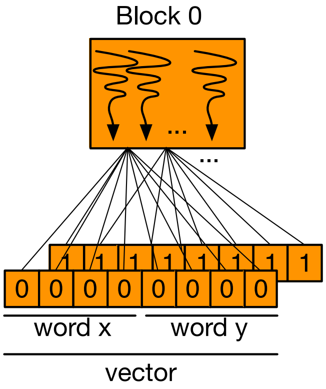
Listing 4.3: Unpacking and packing LLRs from vector types. The strategy of packing and unpacking is of cyclic removal and cyclic introduction to the vector datatype element.

Considering this method of storing low width LLR words onto wide words raises the question whether or not it would be more suitable to pack LLRs messages from the same dataset using this strategy. However, it is easily observed that this raises divergence issues and redundancy of memory accesses. We should consider the limited inter-block communication of the GPU execution model. This is particularly relevant when a two threads in two distinct blocks need to access data elements packed inside the same wide word. Since the memory engine deals with this redundant accesses in runtime it is not clear if there is a broadcast to both threads, since there is no guarantee that both blocks are active simultaneously. Thus, in the better case, when the memory engine is cached, there will be a second access to the vector element that will entail a cache hit. In the worse case, there will be two full accesses to the global memory. Naturally, there is also the question of threads in the same block accessing the same vector element. But in this case the shared memory addressing space can be used to save bandwidth. This propels us to exploit this data representation method for data-parallelism instead of packing consecutive LLR messages together. The aforementioned cases are illustrated in Figure 4.11.

In Figure 4.11 b), the data-parallelism level is raised to 8 codewords. Therefore, there are 8 LLRs corresponding to 8 different codewords, as opposed to interleaving them between multiple vector words. The advantage is clear, since accesses will always be performed at the granularity of the vector words, which can be optimized to meet the maximum alignment criteria, and prevent any redundant access of data, that would be entailed by packing consecutive LLRs in vectors, as seen in Figure 4.11 a).



(a) Contiguous LLRs



(b) Contiguous codewords

Figure 4.11: Memory access to packed LLRs: a) different blocks need to access different vector words, leading to poor bandwidth and redundant memory accesses, due to individual LLR misalignment; while in b) accesses are within the vector type alignment.

The lowest data-parallelism granularity is then defined by the number of the same LLRs corresponding to different codewords that are packed in vector words. Independently of the thread-parallelism level defined, the data-parallelism level can then be increased by expanding the execution grid configuration in multiples of the minimum execution grid configuration that performs the LDPC decoding for the data-parallelism set at the vector granularity, e.g., with 8-bit LLRs packed into `int16` vectors, 64 codewords are defined as the minimum data-parallelism granularity and spawning a twice as large execution grid configuration can be used to decode 128 codewords concurrently. This ability to configure different data-parallelism levels is captured in Figure 4.12.

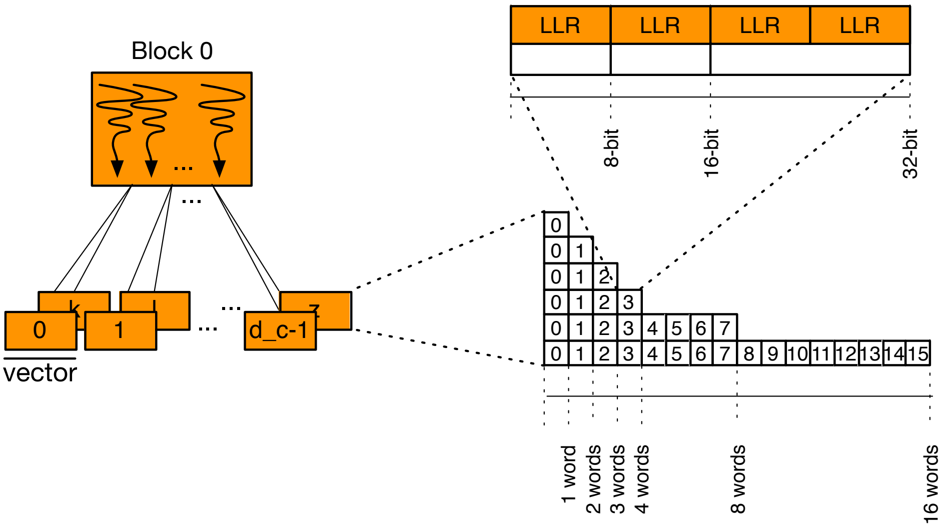


Figure 4.12: Data-parallelism levels at the vector and execution grid granularities.

4.3.2 Thread-parallelism

The partitioning of the LDPC decoding algorithm among threads, or *work-items*, defines how they cooperate towards the completion of the problem. As introduced in Chapter 3, there are several partitioning methods, such as *thread-per-node* (TpN), *thread-per-codeword* (TpC), *thread-per-edge* (TpE) and *block-per-codeword* (BpC), to name some, that can be employed with certain success degrees under the LDPC decoding problem.

Thread-per-Node The closest isomorphic transformation of the Tanner graph onto processing elements on the GPU device is the TpN strategy that applies one thread per *variable node* (VN) and per *check node* (CN), and performs computation based on that logic. The greatest benefit with this approach, is the ability to avoid redundant memory accesses, since threads load only the required messages. On the other hand, there is no data reuse pattern that can be explored due to limitations to the decoding schedule that are possible under this approach. Essentially, despite the fact that threads are not guaranteed to be active simultaneously, the limited support for inter-block synchronization means that only but the *two-phased message-passing* (TPMP) schedule can be deployed.

Moreover, the TpN granularity allows for single-GPU LDPC decoders to peak their performance faster than with coarser granularities. It is simple to observe that, the GPU engine possessing only limited thread spawning capabilities, will have its resources fully occupied faster than if a more limited number of threads is launched, as happens with finer granularities (assuming a fixed LDPC code block length).

Thread-per-Edge The TpE thread-parallelism strategy was presented in one of the seminal works in GPU programming[119]. At the time, the advantage behind this approach was due to the computation having to be defined through the graphics pipeline. It purported greater flexibility to capture the LDPC decoding problem on a 2-D texture structure using a thread granularity based on the edge- than on the node-dimension. The motivation to perform TpE decoding on modern GPU systems is not so strong. In fact, for CUDA and OpenCL enabled devices, it leads to redundant memory accesses that for memory engines without cache lead to full access to the global memory.

The TpE granularity is the most fine-grained strategy. While one can argue that it makes little sense to use it on binary LDPC codes, for non-binary codes, those defined over *binary extension field* (GF(2^m)) in particular, the rationale behind it is to have a maximum number of threads active in the hardware, so as to keep the GPU logic resources committed to the decoding of more complex decoding algorithms. Thus, it may not be necessary to push the data-parallelism levels beyond the granularity of the vector datatype to reach within the peak GPU performance.

Block-per-Codeword The coarser BpC strategy developed commits a thread block, or workgroup, to the decoding of a single codeword, or to a batch of codewords packed onto a vector datatype. This type of thread-parallelism is inspired in the partial-parallel decoders[185,230], which devote a fixed number of *functional unit*s (FUs) to the processing of the LDPC code nodes, in a semi-parallel fashion, since the number of FUs is lower than the number of nodes that are swept. Under the GPU hardware, the FU concept finds its expression in a thread, or workgroup, and for simplicity of synchronization, a block, or workgroup, is devoted to the decoding of the LDPC code. This way, higher flexibility exists to synchronize and apply memory fencing operations.

The drawbacks concerning each approach are mostly related to how efficiently the decoding algorithms are mapped to the GPU engine, but it is also important to notice that two other drawbacks arise with the TpN and the TpE approaches which limit the covering of the design space of LDPC decoders with them. In particular, only the TPMP decoding schedule is guaranteed to execute properly with these approaches. The current generation of CUDA and OpenCL GPUs provide global memory fencing instructions. However, since threads cannot be switched off, if the GPU engine has threads waiting to be executed, the latter will never start, since the former are waiting for the inactive threads to reach the barrier instruction[207].

4.3.3 Optimized Tanner Graph Indexing

The Tanner graph connections define how nodes interact with one another. Under a programming representation, since we are dealing with a software expression, there is no physical entity in the form of a FU that can be hardwired to the other its adjacent nodes' FUs[231], since there the closest isomorphic mapping possible is having a thread expressing a node[10,156]. As a consequence, exchange of messages between nodes must occur through the memory, and in the particular case of the GPU engine, due to the properties of the memory hierarchy, the bulk of exchanges must flow through the global memory. Naturally, since the Tanner graph is sparsely connected, due to the sparse nature of the parity-check matrix, full matrix storage schemes make little sense to apply. Therefore, we discuss suitable methods for the efficient storing of the Tanner graph nodes' adjacencies. Considering the bulk of work found in the literature for *progressive edge growth* (PG) codes, and the prevalence of *LDPC Irregular-Repeat-Accumulate* (LDPC-IRA) and *quasi-cyclic LDPC* (QC-LDPC) codes on wireless communication standards, we discuss suitable methods for each type of code next.

Compressed Sparse Storage Sparse matrix methods can be applied to any LDPC code. However, for LDPC codes whose structure exposes limited to no regularity that can be exploited to devise finely tuned compressed storage methods, more generalist methods need to be applied that are similar to *compressed sparse row* (CSR) and *compressed sparse column* (CSC)[124]. Since the messages exchanged between nodes are accessed by CNs during the CN processing and by VN in the VN processing, the optimized compressed storage for any LDPC code is a combination of both CSR and CSC. Essentially, an ascending index is assigned to each edge, whenever there is a non-null element in the equivalent parity-check matrix position. CNs assign this index looking at the parity-check matrix from a column-wise perspective, while VN do the same, but from a row-wise perspective. The generation of memory indexes from the parity-check matrix is formalized in Algorithm 4.17. Therein, the $\mathbf{H}_{CN}$ and $\mathbf{H}_{VN}$ refer to the parity-check matrix $\mathbf{H}$ rearranged so that only the non-null elements are stored, the former in row-wise fashion and the latter column-wise.

Algorithm 4.16 Construction of the general compressed indexing, suitable for any LDPC code: a) CN indexing and b) VN indexing.

(a) CN indexing

```
$k = 0$
for $m = 0$ to $M-1$ do
  if m=0 then
    $\mathbf{cumCN}(m) = 0$
  else
    $\mathbf{cumCN}(m) \leftarrow \mathbf{cumCN}(m-1)$
  end if
  for $j = 0$ to $edges - 1$ do
    if $\mathbf{H}_{VN}[j] = m$ then
      $\mathbf{CN}_{idx}[j] \leftarrow k$
      $\mathbf{cumCN}(m) \leftarrow \mathbf{cumCN}(m) + 1$
      $k \leftarrow k + 1$
    end if
  end for
end for
```

(b) VN indexing

```
$k = 0$
for $n = 0$ to $N-1$ do
  if n=0 then
    $\mathbf{cumVN}(n) = 0$
  else
    $\mathbf{cumVN}(n) \leftarrow \mathbf{cumVN}(n-1)$
  end if
  for $j = 0$ to $edges - 1$ do
    if $\mathbf{H}_{CN}[j] = n$ then
      $\mathbf{VN}_{idx}[j] \leftarrow k$
      $\mathbf{cumVN}(n) \leftarrow \mathbf{cumVN}(n) + 1$
      $k \leftarrow k + 1$
    end if
  end for
end for
```

This indexing scheme can then be employed in a similar way to that described in Chapter 3. Between the VN and the CN processing phases there are two load and two store operations, and the method described assumes contiguous loading accesses, which need not be indexed, and storing accesses to be indexed by the computed index *lookup-table*s (LUTs), as defined in Algorithm 4.16. Notwithstanding, any other combination could be employed, for instance contiguous storing and indexed loading can be employed just as well, although weighing in how contiguous and indexes accesses are configured yields no significant differences, as determined by the literature survey (c.f. Table A.1). However, indexing with a more efficient method, where efficiency is measured by the number of index elements required is possible by exploring the regular features of the Tanner graph whenever they exist.

Algorithm 4.17 Accessing messages with the general compressed indexing: a) CN access and b) VN access.

```
(a) CN access
for $m = 0$ to $M - 1$ do
  for $j = 0$ to $d_{cm} - 1$ do
    $load_{idx} \leftarrow \mathbf{cumCN}(m) + j$
    $store_{idx} \leftarrow \mathbf{CN}_{idx}(\mathbf{cumCN}(m) + j)$
  end for
end for

(b) VN indexing
for $n = 0$ to $N - 1$ do
  for $j = 0$ to $d_{vn} - 1$ do
    $load_{idx} \leftarrow \mathbf{cumVN}(n) + j$
    $store_{idx} \leftarrow \mathbf{VN}_{idx}(\mathbf{cumVN}(n) + j)$
  end for
end for
```

LDPC-IRA Optimized The generation of LDPC-IRA codes is made by consecutively permuting independent column in the parity-check matrix to generate $r_f - 1$ columns[61]. Since the standardized adoption of these types of codes sees long block lengths, at $N = 16200$ or $N = 64800$ bits in the ETSI standards for 2nd *generation DVB* (DVB 2), the general compressed sparse storage would require LUTs with a very high number of elements. Due to the construction methods of the Tanner graph, it contains structured properties that can be exploited for regularity of memory accesses in multiples or sub-multiples of r_f [185,228,232,233].

Algorithm 4.18 Accessing messages with the DVB 2 LDPC-IRA codes indexing:
a) CN access and b) VN access.

```
(a) CN access
for $m = 0$ to $M - 1$ do
  $\{line, bank\} \leftarrow \{\bmod(m, q), m/q\}$
  for $j = 0$ to $d_{cm} - 1$ do
    $load_{idx} \leftarrow \mathbf{addr}_{idx}(line \times (d_c - 2) + j)$
    $p_{idx} \leftarrow bank - shift_{idx}(addr_{idx}(line \times (d_c - 2) + j)$
    if $p_{idx} > shift_{idx}(addr_{idx}(line \times (d_c - 2) + j)$ then
      $p_{idx} \leftarrow p_{idx} + r_f$
    end if
    $store_{idx} \leftarrow \mathbf{addr}_{idx}(line \times (d_c - 2) + j) + p_{idx}$
  end for
end for
(b) VN access
for $n = 0$ to $N - 1$ do
  $\{line, bank\} \leftarrow \{n/r_f, \bmod(n, r_f)\}$
  for $j = 0$ to $d_{vn} - 1$ do
    $load_{idx} \leftarrow j \times r_f + \sum_{i=0}^{n-1} d_{vi}$
    $store_{idx} \leftarrow load_{idx} + j \times r_f + \bmod\left(\mathbf{shift}_{idx}\left(j + \sum_{i=0}^{line-1} d_{vi}\right), r_f\right)$
  end for
end for
```

Given that consecutive *information node*s (INs), that were generated from the same independent column, have their connections to CNs permuted by a value q, they connect to the same block of CNs in blocks with width r_f with permuted q positions. This allows for a memory layout in two-dimensions that is interpreted when accessed during the VN and the CN processing phase. Since programmable architectures are logically organized into a single dimension, the two-dimensions are linearized with a width of r_f memory banks. In its turn, this entails that VN accesses are contiguous in blocks of r_f data elements, while CN accesses are also contiguous in blocks of r_f elements, although they do not access the memory lines sequentially. The procedure for computing the indexes on-the-fly is formalized in Algorithm 4.18. This indexing scheme is not limited to an organization of the data elements into r_f memory banks and can also be re-expressed into sub-multiples and multiples of it[6,232]. The remapping of data elements in a 2-dimensional layout in a 1-dimension memory addressing space is shown in Figure 4.13.

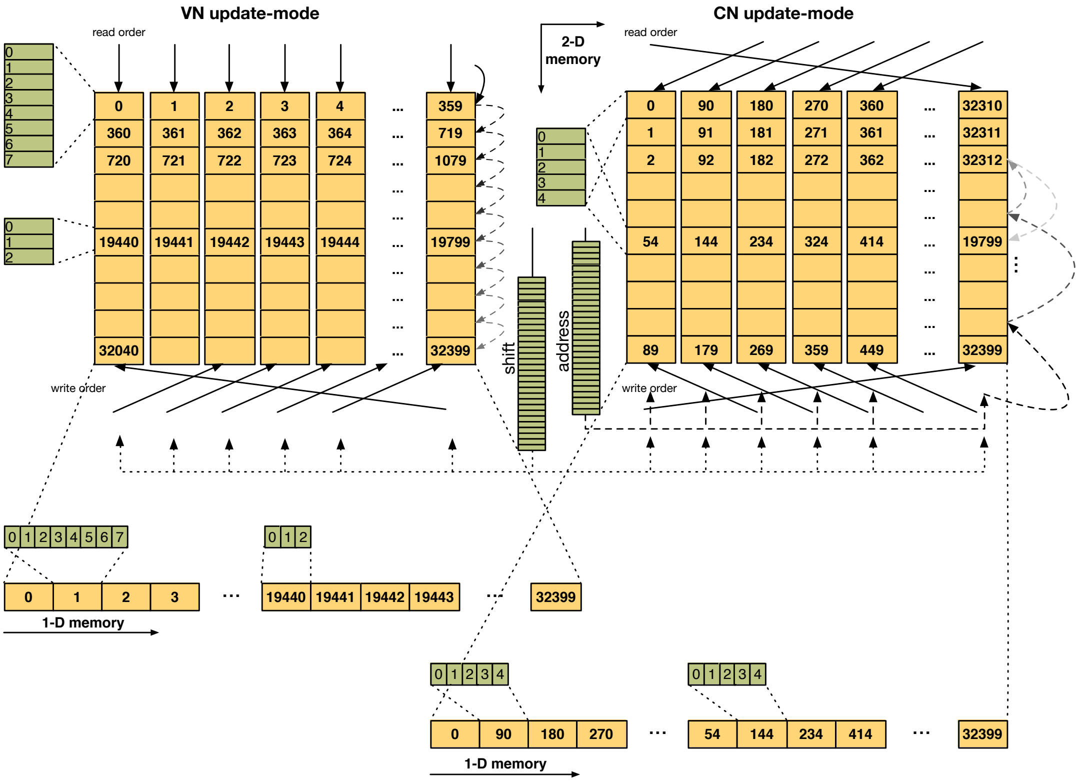


Figure 4.13: Memory layout of the DVB-S2 LDPC-IRA codes in a 1-dimensional memory. As the SIMT execution model will not necessarily guarantee coherence of memory accesses the 2-dimensional memory which had two logical interpretations but was a single physical addressing space is split into two physically allocated arrays that are linearized as shown. The indexes procured in the LUTs generate a valid index in the new layout through on-the-fly index computation (Algorithm 4.18).

QC-LDPC Optimized Considering that QC-LDPC are generated by the expansion of a base matrix $\mathbf{H_f}$ into the parity-check matrix $\mathbf{H}$ by the insertion of permuted identity matrices with dimensions $z_f{\times}z_f$, it is clear that the indexing of $\mathbf{H}$ generated by an arbitrary z_f can be made from the same number of LUT index elements. The scheme devised for indexing QC-LDPC assigns a memory indexing space for messages traversing each direction of the Tanner graph edges based on the maximum degree of VNs and CNs, $\max(d_v)$ and $\max(d_c)$, respectively. Then each non-infinity element in $\mathbf{H_f}$ is assigned to its corresponding position in a column- and a row-wise manner, respectively[131].

Algorithm 4.19 Construction of the QC-LDPC compressed indexing:
a) CN indexing and b) VN indexing.

```
(a) CN indexing
$k = 0$
for $m = 0$ to $M_f - 1$ do
  $d_{cm} \leftarrow 0$
  for $n = 0$ to $N_f - 1$ do
    if $h_{f_{m,n}} \neq \infty$ then
      $\mathbf{CN}_{idx}(k) \leftarrow \{m, n, h_{f_{m,n}}\}$
      $k \leftarrow k + 1$
      $d_{cm} \leftarrow d_{cm} + 1$
    end if
  end for
  $\mathbf{cumCN}(m) = d_{cm}$
end for
```

```
(b) VN indexing
$k = 0$
for $n = 0$ to $N_f - 1$ do
  $d_{cm} \leftarrow 0$
  for $m = 0$ to $M_f - 1$ do
    if $h_{f_{m,n}} \neq \infty$ then
      $\mathbf{VN}_{idx}(k) \leftarrow \{n, m, h_{f_{m,n}}\}$
      $k \leftarrow k + 1$
      $d_{vn} \leftarrow d_{vn} + 1$
    end if
  end for
  $\mathbf{cumVN}(m) = d_{cm}$
end for
```

While regular codes can be indexed with a uniform stride, irregular codes see also a changing stride for their correct position. Thus,exclusive prefix-sums of the nodes degree per row and column in $\mathbf{H_f}$ are calculated for using as base offset for each position in $\mathbf{H_f}$, also designated as layer. Then, the additional LUTs store the layer to which the node is connected in its dimension and on the opposite one are retained, as well as the corresponding $h_{f_{m,n}}$ element, i.e., the permutation with which that particular expansion was made to generate $\mathbf{H}$. A method to generate the index triplets required to index all the messages in the Tanner graph is shown in Algorithm 4.19. The procedure for CN and VN is essentially the same, although the former cycles through the row dimension first, while the latter cycles through the column.

Algorithm 4.20 Accessing messages with the QC-LDPC compressed indexing:
a) CN access and b) VN access.

```
(a) CN access
for $m = 0$ to $M_f - 1$ do
  for $z = 0$ to $z_f - 1$ do
    $load_{idx} \leftarrow \mathbf{cumCN}(m) + j$
    $\{l, n, p\}_{idx} \leftarrow \mathbf{CN}_{idx}(\mathbf{cumCN}(m) + z)$
    if $p_{idx} > z$ then
      $p_{idx} \leftarrow z_f + p_{idx}$
    end if
    $store_{idx} = l_{idx} \times z_f + n_{idx} \times N + p_{idx}$
  end for
end for
```

```
(a) VN access
for $n = 0$ to $N_f - 1$ do
  for $z = 0$ to $z_f - 1$ do
    $load_{idx} \leftarrow \mathbf{cumVN}(m) + j$
    $\{l, m, p\}_{idx} \leftarrow \mathbf{VN}_{idx}(\mathbf{cumVN}(m) + z)$
    if $p_{idx} > z$ then
      $p_{idx} \leftarrow z_f + p_{idx}$
    end if
    $store_{idx} = l_{idx} \times z_f + m_{idx} \times M + p_{idx}$
  end for
end for
```

We should note that the QC-LDPC access scheme is affected by the maximum VN and CN degree. The messages are stored in blocks of z_f contiguous indexes, permuted by the corresponding $\mathbf{H_f}$ entry, and the stride of access between nodes of different layers is related to the maximum VN and CN degrees d_v and d_c. The consequence for irregular codes is different memory sizes required to index $L(r_{mn})$ and $L(q_{nm})$ messages. E.g., the *Wi-Fi* (Wi-Fi) rate 1/2 code for a code length of 1944 bits, to which it corresponds an expansion factor of z_f=81, exchanges 2×6966 messages through the Tanner graph. However, because the maximum degree of VNs for this particular code is 11, but for CNs it is 7, there are more memory elements that do not index any message in the $L(q_{nm})$ memory array than in the $L(r_{mn})$ array.

The relative difference between employing the general compressed indexing for an LDPC-IRA code, and its optimized indexing, has been established[122], where up to 18% of the decoding throughput can be improved for shorter rate codes, as shown in Table 4.4. The diminishing return observed with the higher DVB-S2 index codes is due to the increasing coding rate, which greatly reduces the number of CNs, offsetting the gains observed for the lower coding rate codes[234].

Table 4.4: Comparison of general compressed and DVB-S2 LDPC-IRA indexing methods: decoding throughput, and relative speedup, is presented for 10 decoding iterations[122].

DVB-S2 Codes	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11
Compressed indexing	79	69	61	55	41	55	47	40	35	36	35
LDPC-IRA optimized	87	75	65	65	43	57	49	41	36	36	36
Speedup	1.10	1.09	1.07	1.18	1.05	1.04	1.07	1.03	1.03	1.00	1.00

4.3.4 Binary LDPC Decoding

Binary LDPC decoding on single GPU devices is discussed in this section. The decoders were developed using the CUDA and the OpenCL data-parallel programming models. Binary LDPC decoding on the GPU has been performed in the past, as discussed in Chapter 3, with promising results to the obtaining of high decoding throughputs, though low latency has been largely eluded in the proposed decoders with the few odd exceptions[156,165]. To leverage the GPU device computational power, certain features must be taken into account which are discussed next.

MSA Decoding on GPU Among the message-passing algorithms for binary LDPC codes, the *min-sum algorithm* (MSA) stands out due to its lower numerical complexity. Whereas this, in its turn is made at the cost of sub-optimality, thereby introducing an error performance degradation that can be corrected by scaling, offset or self-correction methods. The uncorrected version, however, can serve as an upper bound to the performance that can be attained by the more complex, and necessarily, slower corrected MSA versions[99].

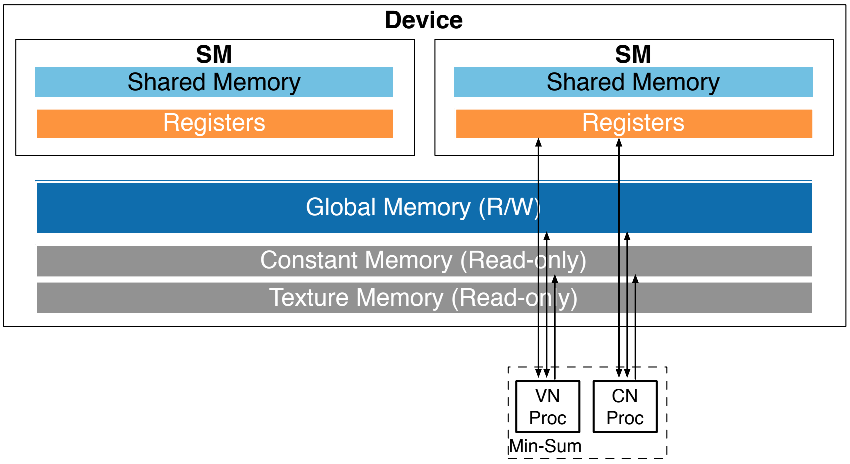
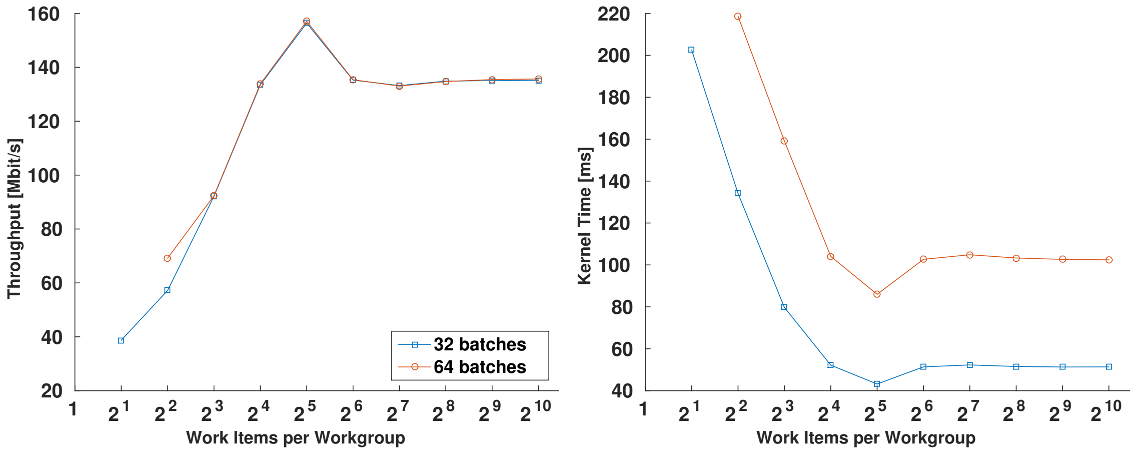


Figure 4.14: MSA employment of the GPU memory hierarchy.

The MSA can be broken down into two distinct kernels, regardless of the decoding schedule selected, corresponding to the CN processing and to the VN processing. The proposed exploitation of the GPU memory hierarchy by the MSA decoder can be seen in Figure 4.14. Therein, threads, or work-items, executing the decoding algorithm will load data, streamed by the host to the global memory, to registers, perform the required computation and proceed to store them back to global memory. The rationale was to make the decoding of data independent of the thread block, or workgroup, dimension. If, on the other hand, shared memory is employed, this introduces a restriction to the maximum number of threads allowed per block, and also, to the number of threads that can be active on the GPU cores. As a consequence, greater flexibility to find the optimal number of threads per block exists under the proposed methodology. Naturally, this poses an unsurmountable constraint to the realization of *Turbo-decoding message-passing* (TDMP) scheduled decoders. However, considering the inability to fully exploit the parallelism, exposed by the Tanner graph under a TPMP expression, of the TDMP decoding schedule, even when the number of decoding iterations is roughly halved, the decoding throughput is as good in one approach as the other[107,156,165]. Consequently, and since the GPU architecture cannot be trimmed down to remove excess silicon, we deemed reasonable to explore the available computing cores to the computation of the same batch of codewords, *intra-codeword* and then scale the procedure to several batches, i.e., *inter-codeword*[182].



(a) Throughput vs. workgroup size

(b) Latency vs. workgroup size

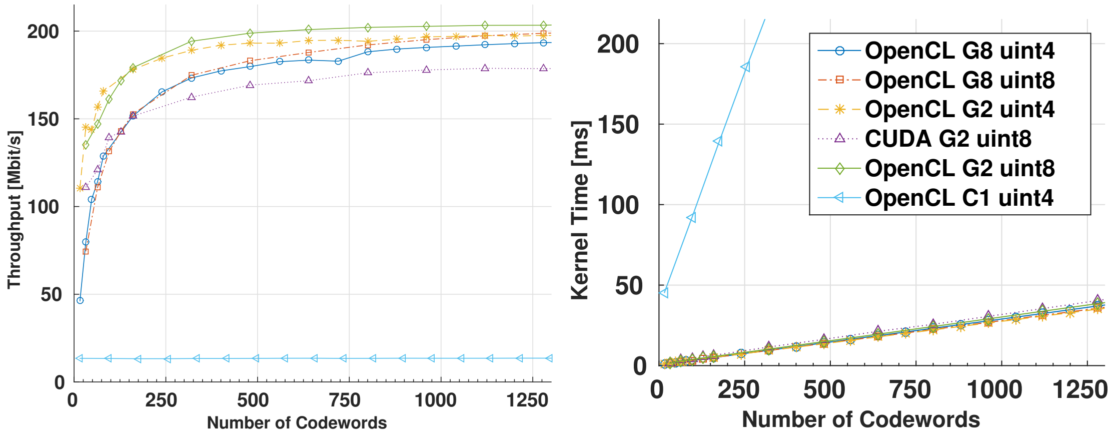
Figure 4.15: Decoder performance vs. workgroup size: a)–b) dataset IV on the GPU architecture G1.

The performance of the GPU, for any given kernel, depends on the thread block dimension. The first limitation is the warp or wavefront dimension. If the block dimension is not a multiple of it, then there will be warps or wavefronts with threads that have no work to perform[170]. To a certain extent, it makes little sense to define *execution grids* that define block dimensions under the warp size or the wavefront size. The illustration in Figure 4.15 presents two cases, when blocks are smaller than those dimensions and when it expands beyond it in powers of 2. In this particular case, the warp size is 32, but a similar behavior would be observed for GPUs with wavefronts of 64 threads[221]. As observed, the performance will steadily increase until one of two cases unfolds. Either the performance will see its peak in a monotonic behavior, and thus, the performance will not improve over a certain block size. At this stage, the GPU cannot provide more active threads. On the other hand, for certain cases, the number of active threads in the GPU engine sees a peak for a certain dimension of the thread block, after which the number of active threads decreases and so does the computational performance.

Considering the decoder topology of Figure 4.14, there is no dependency of the underlying kernels with the block dimension. Hence, with this methodology, the best block size dimension can be known beforehand, using the CUDA occupancy calculator[235], or by empirical evaluation of the best block dimension by performing a sweep of the possible configurations.

Data-Parallelism, Layout and Representation Data-parallelism, as aforementioned, is defined at the intra- and at the inter-codeword level. The data-parallelism level defined by intra-codeword parallelism is constrained by the alignment allowed by the memory engine of the GPU. Whereas CUDA-enabled GPUs allow a maximum alignment of 128 bits[170], OpenCL (AMD) GPUs allow for 512 bits[221]. This means that at the selected 8-

bit LLR bitwidth, considered for its trade-off between being a power of 2 and, therefore, being able to align with different memory engines, it allows for a negligible performance loss when compared to single-precision floating-point representations[3,236]. At the inter-codeword level, the execution grid is replicated so that consecutive words are accessed consecutively at the intra-codeword level, and with a stride of the dimension of the memory size required to store all data elements[182].



(a) Throughput vs. data-parallelism

(b) Latency vs. data-parallelism

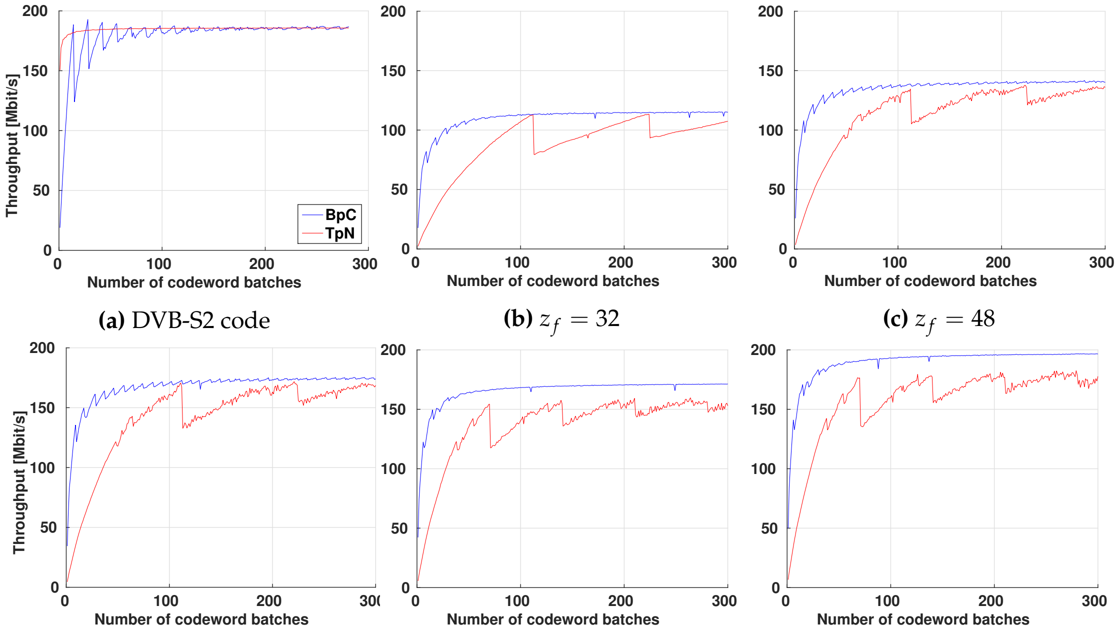
Figure 4.16: Single-GPU regular TpN decoders throughput: a)–b) dataset IV on GPU architectures G2, G8 and CPU C1 for comparison.

As observed in Figure 4.16, the decoding throughput increases with the number of codewords that are given to the GPU to decode concurrently. This trend is largely independent of the GPU device, of the vector datatype bitwidth, and also of the programming model employed, with negligible performance differences observed between CUDA and OpenCL. The throughputs obtained are able to improve with the more codewords that are loaded, with diminishing returns, since the memory engine will eventually reach its peak transaction rate for the LDPC decoding algorithm memory pattern, and also, the number of active threads will not increase. Instead, portions of the execution grid are sequentially executed as more thread blocks are awaiting execution[170]. This strategy, however, comes with the cost of increased latency. For dataset IV, execution of a single codeword batch, yields a third of the maximum attainable performance at twenty-fold lower decoding latency, i.e., without considering memory transfers to the GPU. Clearly, it is a high price to pay, that annihilates any possibility of deploying decoders pursued with this methodology for field-deployment, even if the available power budget would be high to accommodate the power drawn by a GPU accelerator.

Thread-parallelism The MSA LDPC decoder sees the utilization of two thread-parallelism strategies, the TpN and the BpC granularities. The former can be applied without further considerations for the thread block size, i.e., the block dimensions are chosen so that they meet the optimal GPU operating point for the resources consumed by the programmed GPU kernels. On the other hand the BpC thread-granularity is set at the regularity factor of the underlying LDPC code. This strategy is not evaluated for regular LDPC codes constructed with PG methods, since there is no underlying Tanner graph construction regularity that can be exploited. On the other hand, LDPC-IRA and QC-LDPC codes possess this regularity which we exploit. For the QC-LDPC codes, the expansion factor z_f is a good candidate for the block dimension. The same way, a sub-multiple of z_f would also be a good candidate, allowing a partitioned exploitation of the regular properties of the Tanner graph. However, for the majority of QC-LDPC codes that are standardized, z_f is limited to $\max z_f = 96$[35]. For the case of LDPC-IRA codes, the block size is chosen as r_f, although, sub-multiples of r_f would also be possible[233].

The impact in the decoding throughput, and decoding latency, of these approaches are illustrated in Figures 4.17 and 4.18. Since the TpN approach entails two-sized executions grids for CN and VN processing at M and N threads per block per codeword batch, and the BpC approach z_f or r_f threads per block per batch[2], the two strategies must be compared at GPU operation modes that are somewhat comparable. As a consequence, we plot the decoding throughput and latency against a variable workload, to compare both approaches with regards to their attainable peak performances. As observed, the decoding throughputs are monotonically increasing the TpN approach. However, as referred previously, this entails a rising decoding latency. Again, for the TpN granularity, similar levels of throughput to latency increase are observed to what reported before, e.g., dataset I-a) sees a 22% increase in decoding throughput from a single batch to its peak paying a fifty-fold increase in decoding latency. Similar ratios are found for the datasets III-a–e) and we can assume for any LDPC codes under a TpN or BpC decoder expression.

Relaxing the maximization of the decoding throughput to find an operating point, defined by the data-parallelism level, within a certain margin of the peak throughputs yields a better flexibility to our proposed decoding solutions. The tabulation of the decoding throughput and latency in Table 4.5 highlights the nature of the diminishing returns paid by the rising data-parallelism levels. It is clear that the difference between the TpN and the BpC will rank to almost negligible, especially for an operating throughput at 1% of the peak performance. In fact, under these circumstances, it makes almost no difference in decoding latency to chose one granularity approach instead of the other. With regards to throughput, in general, the TpN approach yields a better option. However, 1% of the peak performance can still be seen as a corner case in the analysis since it is still too close to the peak performance of the decoder, which is roughly equivalent to load the GPU with as many words as we are able. We are motivated into finding a better



(a) DVB-S2 code

(b) $z_f = 32$

(c) $z_f = 48$

(d) $z_f = 64$

(e) $z_f = 80$

(f) $z_f = 96$

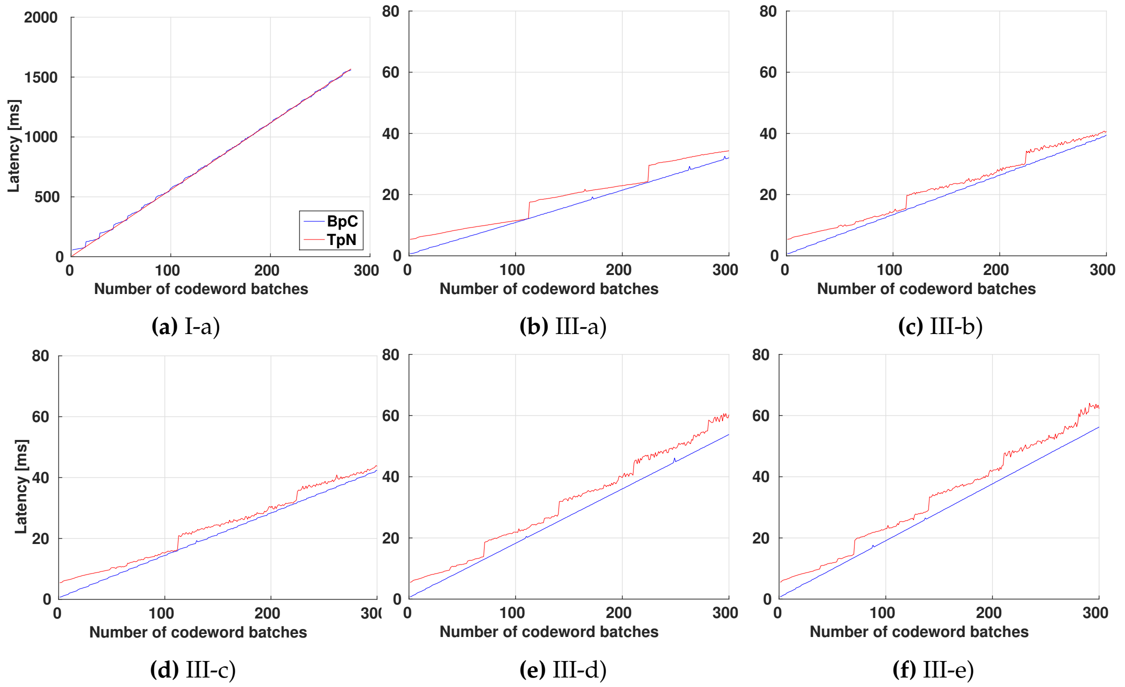
Figure 4.17: Single-GPU DVB-S2 and WiMAX TpN and BpC decoders throughput: a) dataset I-a); and b) to f) datasets III-a)–e).

compromise to the throughput and latency that does not follow this heuristic. Setting the analysis threshold to 5% of the peak decoding throughput shows that much lower latencies can be attained, with operations points still close to full decoder potential. In this case, we can see that relaxing this constraints can lower the latency eight and threefold for the TpN approach for datasets I and III. In addition, almost a twofold reduction in latency is observed for all datasets using the BpC approach. For all the considered cases, the TpN granularity represents a better strategy as it is able to deliver higher throughputs (Figure 4.17) at lower latencies (Figure 4.18).

Table 4.5: Throughput and latency at 1% and 5% of the peak performance for datasets I-a) and III-a–e).

		1% to peak performance						5% to peak performance					
		Throughput (Mbit/s)		Latency (ms)		Workload		Throughput (Mbit/s)		Latency (ms)		Workload	
	Approach	TpN	BpC	TpN	BpC	TpN	BpC	TpN	BpC	TpN	BpC	TpN	BpC
Dataset	I-a)	183.9	190.9	158.1	135.2	28	20	176.5	183.2	23.5	77.0	4	14
	III-a)	114.3	112.5	17.3	24.3	161	224	109.7	107.9	6.2	11.48	55	100
	III-b)	140.7	134.9	23.3	29.5	177	217	136.5	130.9	8.8	15.22	65	109
	III-c)	173.9	170.5	17.2	15.9	121	111	166.9	163.6	8.0	15.8	55	105
	III-d)	169.9	158.7	26.8	23.2	148	210	163.0	152.3	8.1	13.8	43	69
	III-e)	195.0	183.2	28.3	52.2	150	258	187.1	175.8	9.5	14.2	48	68

Asynchronous memory transfers could be defined which would hide the memory transfers time completely behind computation[156] and would also work in favor of the intended high throughput, low latency objective. Considering that the workload can be broken down into distinct streams of computation, a number N_s of streams, each de-



(a) I-a)
(b) III-a)
(c) III-b)
(d) III-c)
(e) III-d)
(f) III-e)

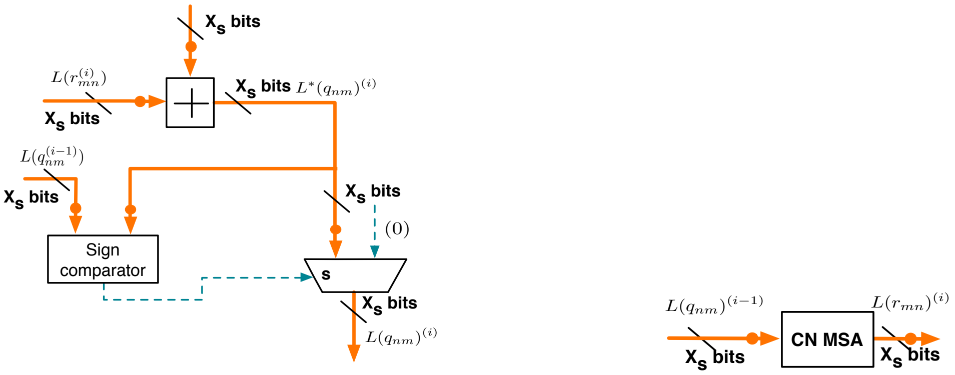
Figure 4.18: Single-GPU DVB-S2 and WiMAX TpC and BpC decoders latency: a) dataset I-a); and b) to f) datasets III-a)–e).

coding N_b codeword batches, in theory achieves the decoding throughput associated with $N_s \times N_b$ codeword batches, with a decoding latency associated with the decoding of N_b batches. Practice, however, states otherwise. In particular, one of the limitations to the streamed approach is that is loosely absent from the OpenCL GPU implementations. Thus, it is limited to CUDA-enabled devices only. Additionally, the Fermi architecture, the first architecture to implement the overlapping of execution between different streams, could not cope well with the successive calling of multiple kernels belonging to different streams[234]. It was the introduction of the Kepler family that permitted to reach within a negligible margin to the theoretical bound mentioned. In particular, Wang *et al.*[156] show that N_s=32 for N_b= $\in \{1, \cdots, 8\}$ reaches within a desired operation range of under 2 ms latency for WiMAX and Wi-Fi decoders.

Self-correction Decoding Architecture The *self-corrected min-sum algorithm* (SCMSA) has been formalized in Alg. 2.8. However, prototyping the SCMSA into a decoder system requires further reasoning, thus, we are interested in defining guidelines to how this algorithm can be efficiently realized on a massively parallel processor such as the GPU. The nature of the self-correction is substantially different than the scaling or offset corrections. First, there is no *signal-to-noise ratio* (SNR) dependency with which to deal, so the correction is optimal in that sense. Secondly, it involves the knowledge of a previously produced message, $L(q_{nm}^{(i-1)})$, to be compared against a new tentative $L^*(q_{nm}^{(i)})$.

Furthermore, if in the previous iteration there had been a sign change, then an erasure was introduced to the $L(q_{nm}^{(i)})$ message, and the self-correction technique is not to be in effect for this particular LLR[3,83]. Clearly, the previous message $L(q_{nm}^{(i-1)})$ and the incoming message from the CN, at the same edge, $L(r_{mn}^{(i-1)})$ must be both loaded for the VN processing. The latter is a part of the standard MSA VN processing algorithm, although the former is not (c.f. Algorithm 2.5).

Algorithm Validation For a software representation, this poses a less critical constraint as memory available is sufficient to hold the value of $L(q_{nm})$ and $L(r_{mn})$ simultaneously. Typically, ASIC and FPGA solutions save on the memory space by performing partial-parallel configurations able to read and write LLRs consumed and produced by the nodes in-place[185,229]. The major drawback is in this case, the extra memory bandwidth required involved in loading an extra message to perform the SCMSA additional steps, since the data moved by the VN processing increases by a third if the unoptimized approach, illustrated in Figure 4.19 is taken.

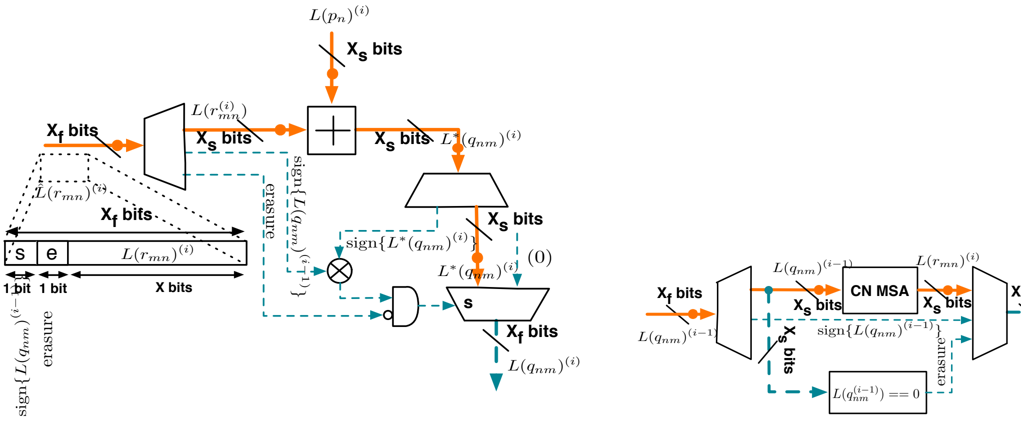


(a) VN datapath diagram.

(b) CN datapath diagram.

Figure 4.19: Non-optimized SCMSA datapath compared to the MSA: a) VN processing; and b) CN processing. The former sees twice the number of messages loaded, whereas the latter is the same as in the MSA.

Algorithm Optimization A closer inspection to Algorithm 2.8 shows that the knowledge regarding an erasure introduced in the previous iteration can be quantified by a single signaling bit $erasure^{(i-1)}$, instead of comparing the value of the $L(q_{nm}^{(i-1)})$ message. This also serves the purpose of turning the self-correction on or off. When set to 1, the behavior of the SCMSA is formally that of the MSA, and when set to 0, it evaluates numerically if the algorithm operates the self-correction or proceeds its message update using a pure MSA approach. Also, in addition to the single bit *erasure*, the sign bit from the previous iteration, $sign\{L(q_{nm}^{(i-1)})\}$ can also be stored. A solution to this is depicted in Figure 4.20. Therein, the bitwidth of the LLR messages exchanged is elevated from X_s to $X_f = X_s + 2$. The two *most significant bits* (MSBs) of the new word store the sign and the



(a) VN datapath diagram.

(b) CN datapath diagram.

Figure 4.20: Optimized SCMSA datapath compared to the MSA: a) VN processing; and b) CN processing. Both see two extra bits added to the LLR message bitwidth to represent the sign and erasure of the previous iteration.

erasure bits, signaling the sign-bit of $L(q_{nm}^{(i-1)})$ and whether an erasure was introduced at iteration $(i-1)$. The increased bitwidth requires a number of new operations to be performed. At the input, the LLR and the control bits must be sliced, with the former proceeding to the MSA datapath. At the end, data is to be bundled again, after the new control bits have been updated and the new LLR message after self-correction has been updated. This leads to the forgoing of the comparator in Figure 4.19 and to the introduction of a simple XOR-OR chain of operations that control the introduction of an erasure. While this describes the VN optimized processing, the CN algorithm remains mostly the same, the only difference is that data must be sliced and bundled again, in order to update the LLR and also to preserve the control bits that are merely forwarded by the CN processing[3].

Self-correction Performance Evaluation of the BER performance of the SCMSA yields very promising results. The correcting capabilities of the self-correction technique has been tried for datasets I-a–b), since they represent the selected codes (in Table 4.3) with the greatest error-correcting capabilities due to their code block length. Whereas in the I-b) there is a small gap on the performance, close to 0.5dB between the performance of the MSA and the SCMSA, for I-a) this gap is close to 0.7dB. The great advantage is that the self-correction is not SNR dependent on the latter. Thus, there is a positive non-diminishing gain from using it when compared to the plain MSA. Compared to the *normalized min-sum algorithm* (NMSA) this is an advantage, since, as observed in Figure 4.21, the BER performance diverges from its initial behavior and would meet the MSA at a BER level higher than the error-floor, whereas the SCMSA BER will only meet that of the MSA in the error-floor region. This is due to the NMSA optimization of the scaling factor had been performed for higher BER than those covered by our simulation, which shows the potential caveats concerning the use of the NMSA and performing a BER evaluation that is not error-floor deep[79].

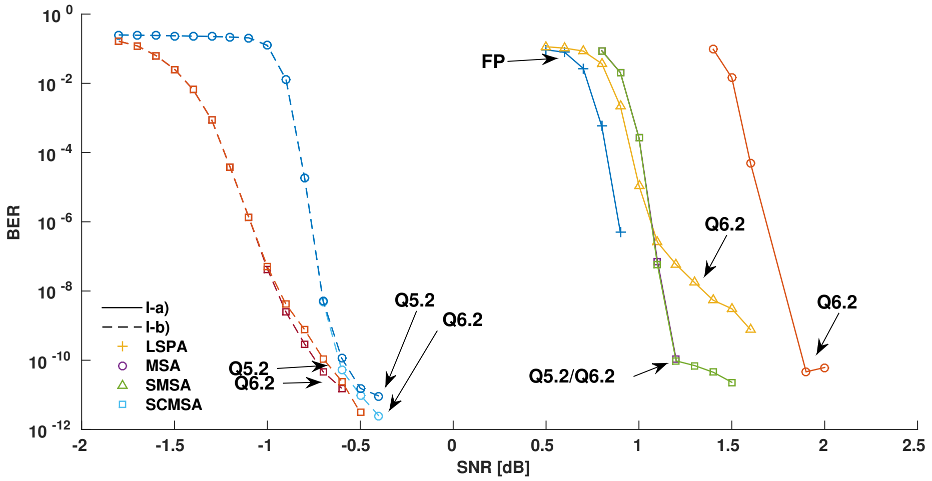


Figure 4.21: BER performance comparison of LSPA and MSA-based algorithms using datasets Ia–b) for the LSPA, MSA, NMSA and SCMSA for different data representations marked by the label near each curve. It should be noted the same BER performance obtained for the SCMSA for both $Q5.2$ and $Q6.2$.

To weigh in the overhead posed by the control bits, that add two bits to the LLRs bitwidth, we defined a quantization of $Q5.2$ and $Q6.2$ for dataset I-a). As seen, there is negligible BER between both representations, leading to a net overhead of just 1-bit to the LLR bitwidth, when compared to the performance of MSA with an extra bit devoted to the LLR quantization. E.g, a BER gain of 0.7dB is possible for I-a) using 9-bit extended LLRs that devote 7 bits to data and the remaining 2 to the control bits[3,15]. The BER simulation was able to achieved within the error floors of the LDPC codes of dataset I in a feasible execution time because the GPU device has been exploited to the fullest of its capabilities as a BER simulator device. The surrounding discussion of the BER simulator is given on Section 4.4.

Summary The proposed binary LDPC decoders are able to reach within high decoding throughputs at contained decoding latencies. The proposed thread-granularities show that the attainable decoding throughputs is essentially equivalent between the TpN and BpC approaches. The former allows for a faster exhaustion of the GPU resources with the rising workload, whereas the latter requires more codeword batches to do so, leading to much higher latencies. The introduced Monte Carlo BER simulator has the ability to provide a tremendous speedup to the BER characterization of new LDPC codes and decoding algorithms. In particular, we have shown the GPU architecture suitability for simulation and prototyping in the case of the self-correction technique applied to the

MSA (SCMSA), showing that the self-correction permits considerable BER gains not only in the BER waterfall region, but also in deep BER error-floor ranges.

4.3.5 Non-binary LDPC Decoding

The "holy-grail" of non-binary LDPC decoding is to find a decoder whose complexity grows with m and not with 2^m [237]. Until then, the odds are that non-binary LDPC codes will see a limited field of deployment, at least for very high data rate applications. Nevertheless, this still leaves room for their application in lower throughput uses and given the immense computational power at our disposal today, the possibility to reach within reasonable decoding throughputs and latencies must not be discarded. Consequently, we discuss in this section, a methodology to define GPU-based non-binary decoders that could meet these objectives, using the CUDA data-parallel programming model on single GPU devices.

FFT-SPA Decoding on GPU Among the different decoding algorithms proposed, the *FFT sum-product algorithm* (FFT-SPA), as discussed in Chapter 3, is the most promising algorithm as it yields no sub-optimality, the *extended min-sum* (EMS) trades an m factor in the complexity by a n_m factor but with the cost of sub-optimality, and it does not require computation in the Galois domain. Due to the different stages required by the FFT-SPA (Algorithm 2.12), each phase will see a distinct kernel definition that exploits differently the GPU engine capabilities. Since both the CN and the VN processing deal with Hadamard products, or pointwise multiplication of the *probability mass function* (*pmf*) elements, and the factor graph of the non-binary LDPC code introduces only a simple permutation and depermutation of *pmfs*, the bulk of the algorithm is the *fast Walsh-Hadamard transform* (FWHT). Due to this, the basic conception of the FFT-SPA into GPU kernels is illustrated in Figure 4.22.

Therein, the basic assumption is having data flowing from the global memory space to registers in all kernels, except of the FWHT. In the latter, the shared memory space will be employed so that synchronization between each stage of computation[238] in the decoder is faster than having to fence through the high latency global memory space.

Operational transform FWHT

The *sum-product algorithm* (SPA), used for decoding binary LDPC codes can be extended to deal with LDPC codes over GF(2^m)[46]. Its numerical complexity, however, grows non-linearly with the field's order m which detracts its usage as a suitable decoding algorithm over GF(2^m), namely due to the CN processing step[46,91]. The complexity of equation (2.42), follows $O(M \cdot d_c \cdot 2^{d_c \cdot m})$ and can be ameliorated by switching from the

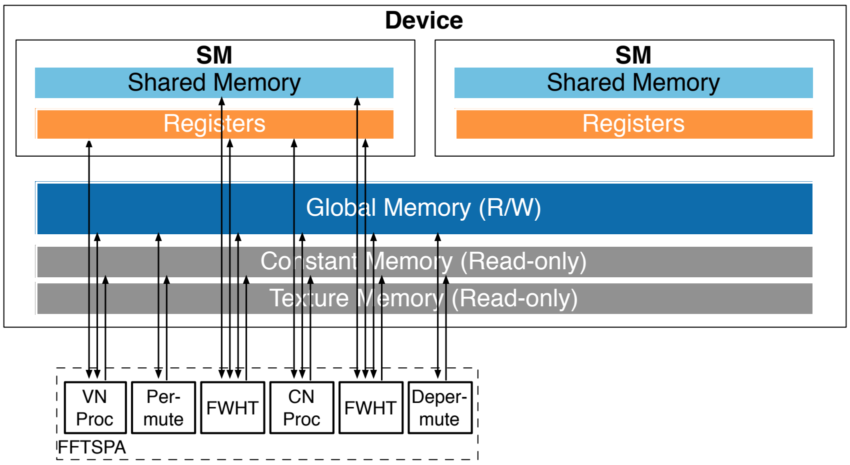


Figure 4.22: FFT-SPA employment of the GPU memory hierarchy. The most intensive FWHT kernel takes full advantage of the memory hierarchy of the GPU.

pmf domain to the Fourier domain[47], which transforms the convolution in (2.54) into a product (2.54), where the *Walsh-Hadamard transform* (WHT) is employed instead of the *discrete Fourier transform* (DFT), since in the GF(2^m) domain, the Fourier Transform consists of the WHT, as dicussed in Chapter 3. The Walsh-Hadamard matrix is obtained by following the Kroenecker product $\bigotimes$[238]:

$$\mathbf{H}_{2^k}{=}\mathbf{H}_2 \bigotimes \mathbf{H}_{2^{k-1}}, \text{where } \mathbf{H}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}. \tag{4.2}$$

Since the numerical complexity associated with (2.54) is $\mathrm{O}(M \cdot d_c \cdot m \cdot 2^m)$ and that of the FWHT follows $\mathrm{O}(2^m \log_2 2^m)$, the FWHT is the highest computational burden of the FFT-SPA. Hence, a primary challenge of the realization behind Fourier-domain decoders, is the ability to efficiently compute the FWHT[4]. The LDPC decoding context favors the execution of several WHTs concurrently, since each *pmf* sees its domain defined over the probability and the Fourier domains depending on the processing phase in which they are. As a consequence, we are able to exploit two levels of parallelism in the parallel expression of the FWHT to a single-GPU implementation, 1) intra-FWHT where multiple threads to concurrently compute each WHT are spawned, and 2) inter-FWHT with the launching of several blocks per execution, each computing their own WHTs.

The transform sizes $N{=}\{128, 256\}$ are of particular interest, since over GF(2^m) they correspond to the dimensions of the WHT for the high-order fields GF(2^7) and GF(2^8). We define kernels that utilize the low latency, high bandwidth register file and the on-chip shared memory using two levels of parallelism, intra- and inter-FWHT, as shown in Figure 4.24. In particular, we discuss the particularities of finding a suitable indexing scheme for the shared memory storing of data of the inner computation produced at each stage in the FWHT factorization.

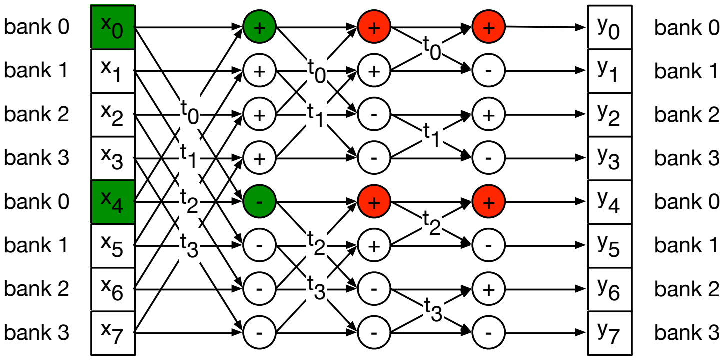


Figure 4.23: Shared memory conflicts on the 8-point radix-2 FWHT. The underlying shared memory architecture is assumed to be a B=4-bank addressing space, with one thread t_i per butterfly. The in-place computation of the FWHT will be conflict free (shown in *green*) for bank 0 in the first stage, as only thread t_0 accesses it, but will yield conflicts in the second and third ones (shown in *red*), since it is accessed by threads t_0 and t_2. Similar patterns can be extrapolated for higher dimension FWHT and for wider shared memories, with $B \in \{16, 32\}$.

Since it would be impossible to devise an indexing scheme that would unfold bank-conflict free accesses for in-place computation, the different stages consume data and produce data according to distinct strides. This way, several different strides can be employed to ensure minimal to no bank-conflicts, since they are optimized for a particular stage. The naive implementation of the 8-point radix-2 FWHT illustrated in Figure 4.23 is shown in Figure 4.24 depicting how data traverses the different memory addressing spaces. It is clear that too low factorizations would impair the number of threads that can be spawned per FWHT and would also entail more stages to compute, thereby increasing the shared memory utilization, in its turn, reducing the GPU ability to keep a high number of active threads. On the other hand, computation split across different warps or wavefronts, requires the enforcing of synchronization points at each stage computed. Hence, the higher the number of threads, the greater the number of warps, and the higher the overhead associated with synchronizing all the threads.

Moreover, the specificities of the CUDA GPU engine with regards to the shared memory are such that optimal strides for 0-bank-conflicts must validate the following constraint. Assuming the memory access is strided, as defined in Listing 4.4, the stride s

```
extern __shared__ int shared_data[];
int data = shared_data[BaseIndex + s * tid];
```

Listing 4.4: Shared memory strided access of the `shared_data` array with a stride s, by thread tid, starting at an arbitrary $BaseIndex$ offset position.

must conform to the following constraint to avoid memory bank conflicts when accessing data. For two arbitrary threads tid and $tid + n$, a bank conflict occurs whenever $s \times n$ is a multiple of the number of banks B or, equivalently, when n is a multiple of $32/d$, and

$d = \gcd(32, s)$. Hence, there is no bank conflict only if the warp size is less than $32/d$, which happens for d=1 and s odd. The solutions finding the better compromise between maintaining a high number of active threads intra-FWHT, but not too many so as to keep synchronization overheads low, and that are able to finding odd strides that minimize or eliminate all the shared memory bank conflicts are discussed next.

The breaking down of the transform size N into factors assumes that no factorization higher than radix-8 occurs. Thus, the factorization of different N powers of two can be written as seen in Table 4.6. Considering that thread block sizes equal to the warp size do not maximize the occupancy of the GPU hardware, and the aforementioned limitations a fixed size of 64 threads is employed. Given this constraint, for some transform lengths 2^m, the number of WHT computed per thread block can be higher than one. Then, each of the 64 threads loads a number of data elements and proceeds to store these into registers. This is the prologue phase, illustrated in Figure 4.24. Then, radix-n computation is applied and data is produced to shared memory, and from it consumed by the threads when computing the next stage. Between each storing and loading from shared memory, a `__synchthreads()`, or a `barrier(CLK_LOCAL_MEM_FENCE)`, is called so that threads from the two warps composing the block can correctly consume data that has been produced, i.e, so that *write-after-read* (WAR) hazards do not occur[238,239]. The strategy pursued to avoid any bank conflicts is to define a shared memory array with a dimension higher than the transform size. Thus, there are empty elements at each stage being computed, however, every array element at a certain point retains data, and the pattern with which certain indexes are left without data is due to the adjusting to the stride s so that conflicts are avoided. For the particular case of N=256-point WHT, the prologue and the indexing at the first stage is exemplified in Listing 4.5. As shown, the 272 elements in the shared memory array, allows for non-multiple of 32 strides to be employed, as s=68 will not entail a d=1 for any threads in the warp.

2^m	2	4	8	16	32	64	128	256
Radix Factorization	2	4	8	8×2	8×4	8×8	8×4×4	4×4×4×4
No. WHTs per block	128	64	32	16	8	4	2	1

Table 4.6: FWHT factorizations employed.

Experimental results The proposed methodology is benchmarked for $N \in \{128, 256\}$, because these dimensions correspond high order GF(2^m) that attain good BER performance[192]. The scalability of the indexing is tested against $B \in \{16, 32\}$ bank GPU architectures, in particular, using GPU devices G1, G2 and G6 (Table C.1), which have compute

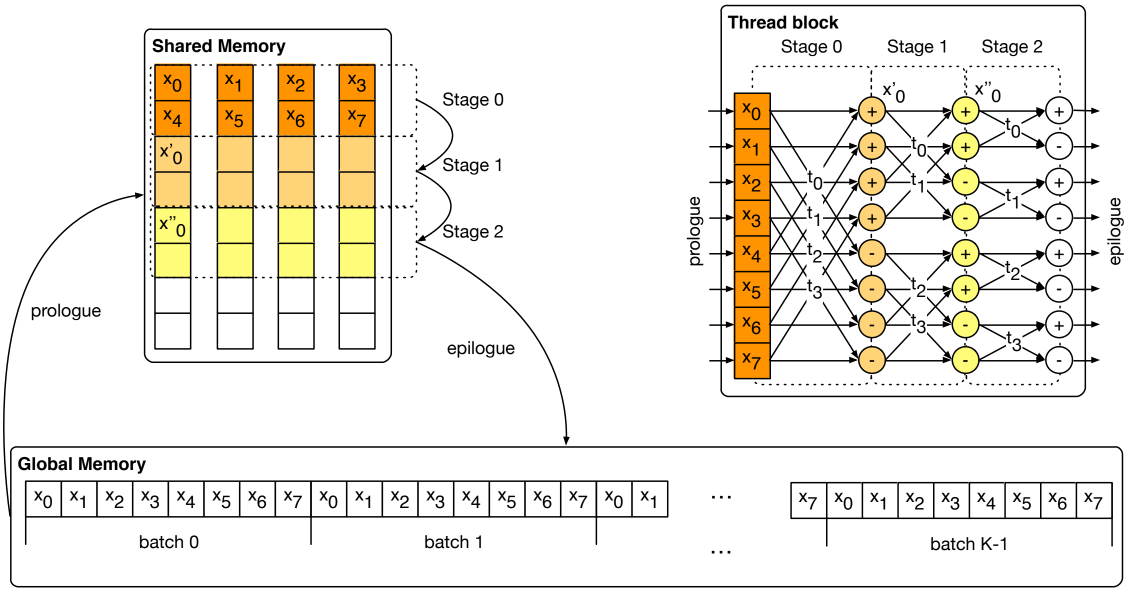


Figure 4.24: 8-point radix-2 FWHT computation through the GPU memory. Data stored on the global memory is moved to the shared memory and out-of-place computation is performed on the data in the shared memory. At the FWHT kernel epilogue data is moved back to the global space.

```
__shared__ float sMem[272];
float a[4];
offset =  blockIdx.x * 256 + threadIdx.x;
//---------- ---------- prologue
a[0] = in[offset + 0];
a[1] = in[offset + 64];
a[2] = in[offset + 128];
a[3] = in[offset + 192];
radix4_kernel(a); //---------- compute first radix-4 stage
j = threadIdx.x & 3;
i = threadIdx.x >> 2;
sMem[threadIdx.x + 0]   = a[0];
sMem[threadIdx.x + 68]  = a[1];
sMem[threadIdx.x + 136] = a[2];
sMem[threadIdx.x + 204] = a[3];
__syncthreads(); //---------- synchronize threads
a[0] = lMemLoad[j * 68 + i) + 0];
a[1] = lMemLoad[j * 68 + i) + 16];
a[2] = lMemLoad[j * 68 + i) + 32];
a[3] = lMemLoad[j * 68 + i) + 48];
__syncthreads(); //---------- synchronize threads
```

Listing 4.5: Indexing for the CUDA N=256-point FWHT optimized for B=32 using a radix-4 factorization and 64 threads per block. The indexing show is for the prologue and the first stage of computation.

capabilities 1.3, 2.0 and 3.0, respectively. To highlight the importance of bank optimization, we have profiled the bank optimized versions for all GPUs, thereby testing versions not optimized for the particular number of banks in the shared memory.

Table 4.7: Throughput and profiling of the 128- and 256-point FWHT kernels.

	Compute Capability	N	WHTs/ms	IpC	Bank Conf. Prob. (%)
Optimized $B = 16$	*1.3*	128	37434	N/A	0.00
		256	18665		0.25
	2.0	128	67766	0.678	0.51
		256	18783	0.578	0.69
	3.0	128	69521	0.731	0.44
		256	17178	0.509	1.34
Optimized $B = 32$	*1.3*	128	38959	N/A	0.00
		256	27303		0.07
	2.0	128	75007	0.725	0.00
		256	34035	0.938	0.00
	3.0	128	72754	0.788	0.00
		256	51699	1.424	0.00
$B = 16$[240]	*1.3*	256	3531	N/A	2.01
	2.0		4039	0.364	5.92
	3.0		3138	0.278	4.85

As seen in Table 4.7, the memory engine is able to access data elements in the shared memory whenever the indexing scheme has been optimized for the number of banks in the architecture. In fact, optimizing for B=32 and deployment on the B=16 architecture entailed a low percentage of conflict, and has actually lead to performance improvements for that architecture. However, the reciprocate case, where the optimization has been performed for a lower number of banks than what the GPU provides sees a high performance degradation. Even though the relative number of bank conflicts is low, at most 1.34% of accesses incurs a conflict, the equivalent IpC degradation has tremendous implications for the throughput of the FWHT kernels. For instance, dropping from 0.678 to 0.578 leads to a decrease in throughput from 67766 to 18783 WHT computed per millisecond, with similar patterns observed seen for the other equivalent cases. It is interesting to note the differences in the IpC and throughput are non-negligible even for extremely low levels of bank conflicts, e.g., for the B=16 GPU, the FWHT optimized for B=32 yields 0.07% of bank conflicts, however, this translates into a 30% reduction in throughput.

The impact of the variation in bank conflict probability on the throughput can be assessed by comparing the FWHT optimized for $B = 32$ with the one for B=16, since the former always yields lower conflict probabilities than the latter. The speedup range for this comparison is represented in Figure 4.25, and spans from 1.04 to 3. As expected the highest speedup occurs for point F at 3 which experienced a 1.34% conflict probability reduction, whereas point A sees a negligible 1.04 speedup. As observed, speedups are higher for the larger 256-point FWHT, regardless of experiencing a lower variation in bank conflict probability as seen with setups B and D (B=32) and C and E (B=16), experiencing speedups of 1.46, 1.81, 1.11 and 1.05, for variations of 0.18%, 0.69%, 0.51% and 0.44%, respectively. It is, thus, of extreme importance to optimize for the shared memory engine when utilizing it on the developed algorithms. Considering its pivotal role n the non-binary FFT-SPA, the methodology herein described is critical to maximize the decoding throughput[4].

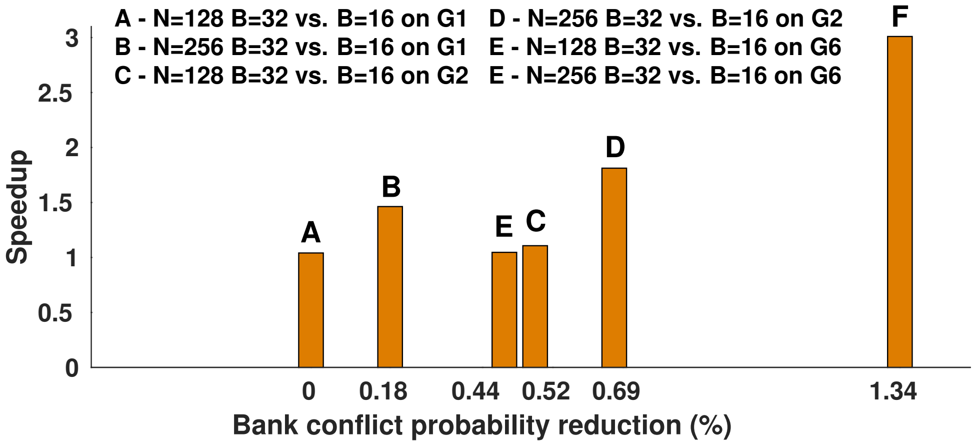


Figure 4.25: Speedup of the B=32 FWHT compared to B=16 vs. bank conflict probability.

Data-Parallelism, Layout and Representation The Tanner graph indexing methods discussed in Section 4.3.3 can be extended to GF(2^m) easily. The FFT-SPA requirement to load 2^m probabilities, organized in *pmfs*, introduces a non-unitary stride to the same methods therein discussed[4]. This regularity works in favor of the GPU memory engine since there is always 2^m consecutive elements loaded per message. In particular fields of higher dimension, such as GF(2^8), work best in favor of coalesced memory accesses. For instance, an LDPC code defined GF(2^8) entails accesses to 2^8 consecutive elements. If the underlying bitwidth is set at 128-bits, for CUDA devices, or at 512-bits, for AMD devices, will saturate the memory engine alignment and a single *pmf* occupies a full transaction. In other words, regardless of the memory layout, accesses are coalesced and maximize the global memory bandwidth.

However, a limitation arises with the data representation of individual probabilities in the *pmf*. Nor CUDA- nor OpenCL-devices address at the programming model level, or have special ALUs, to deal with fixed-point computation. Whereas for LLR-based algorithms, that apply only additions, subtractions, with the odd-scaling of data made possible through a temporary conversion to floating-point representation, multiplications and divisions require emulation through LUT-indexed computation, for instance. Since this would mean a massive number of threads constantly accessing LUTs that for high orders of GF(2^m) would not fit into the constant memory. As a consequence `float4` vector types are devoted to storing 4 distinct *pmfs* belonging to different codewords. The codeword batch size is then 4 as opposed to 16, as in the majority of the proposed binary decoders.

Thread-parallelism The granularity of the proposed FFT-SPA decoder follows the TpE approach. The rationale behind this choice is mostly due to the complexity of the algorithm which compels us to keep more threads active to deal with the extra data to be processed per message. Furthermore, due the L2-cached memory engine the majority of the transactions that are redundant between threads are actually merged onto a single transaction. This allows the elimination of the asymmetry behind the TpN approach, whereupon the CN processing had less threads spawned than the VN processing. In this case, all the kernels launch one thread per *pmf*, except the FWHT which, as discussed, launches one block per *pmf*[13].

Table 4.8: Execution time and throughput of the FFT-SPA single-GPU decoder for variable GF(q) and different FWHT implementations.

Dataset	V-d)			V-e)			V-f)			V-g)		
Iterations	5	10	15	5	10	15	5	10	15	5	10	15
Throughput [Mbit/s]	1.63	0.82	0.55	0.78	0.39	0.26	0.42 $3.34^{\dagger}$	0.21 $1.67^{\dagger}$	0.14 $1.11^{\dagger}$	0.26 1.52* $2.37^{\dagger}$	0.13 0.77* $1.20^{\dagger}$	0.08 0.52* $0.79^{\dagger}$
Execution Time [ms]	7.51	14.93	22.38	15.80	31.55	47.41	29.01 $3.67^{\dagger}$	58.02 $7.36^{\dagger}$	86.88 $11.01^{\dagger}$	48.27 8.07* $5.20^{\dagger}$	96.43 15.90* $10.28^{\dagger}$	145.16 23.81* $15.51^{\dagger}$

† Proposed bank-optimized FWHT; * bank-optimized with greedy algorithm[240]; unmarked is a generic radix-2 factorization.

Throughput of the FFT-SPA LDPC Decoder on GPU The FFT-SPA decoder tested the GF(2^m) dimensions $2^m \in \{2^5, 2^6, 2^7, 2^8\}$, i.e., datasets V-d–f)[95]. Table 4.8 shows the decoding throughputs and latency obtained for the G1 GPU device (c.f. Table C.1). To illustrate the impact of the FWHT on the overall performance we distinguish the following cases, The FWHT kernel can be 1) a radix-2 factorization suitable for all GF(2^m) orders and withholds any optimization made to the correct usage of shared memory banks, 2) a greedy approach to minimizing the number of bank conflicts is used for a B=16 bank architecture, whose kernel is designated as FWHT*, and 3) a shared memory optimized for $B \in \{16, 32\}$, designated as FWHT†. Experimental results for the relative weight of the developed GPU kernels are shown in Table 4.9. It can be seen that the radix-2 FWHT implementation consumed 94.3% of the kernels execution time for GF(2^8). A significant improvement was achieved by using the greedy-optimized FWHT, which lowered the FWHT weight to 65.1%. The proposed FWHT further lowered it to 45.4% of the total execution time.

By using the greedy-optimized FWHT and the proposed FWHT, throughput is raised, at 5 iterations, from 0.26 Mbit/s to 1.52 and 2.37 Mbit/s, respectively for GF(2^8). For the GF(2^7) case, the throughput at 5 iterations was elevated from 0.42 Mbit/s to 3.34 Mbit/s. This is equivalent to speedups of 9.11× and 7.95× when compared to the radix-2 FWHT and the proposed bank-optimized FWHT.

Table 4.9: Relative execution time of kernels in the FFT-SPA for GF(2^8).

Kernels	FWHT	FWHT*	FWHT†
FWHT	94.3	65.1	45.4
CNProc	1.7	10.6	16.3
VNProc	1.6	10.1	15.8
Permute	1.2	7.2	11.4
Depermute	1.2	7.0	11.1

Related Work Other authors have proposed non-binary LDPC decoders on the GPU engine. Wang *et al.*[154] have developed a GPU-based Min-Max LDPC decoder for GF(2^m) using the OpenCL programming mode. However, due to the Min-Max algorithm different characteristics, some of the optimizations therein described are limited to algorithms dealing with computation in GF(2^m). In particular, forward-backward searching of the configuration set for each particular CN pertains to these type of algorithms only. Romero and Chang[147] have presented a non-binary LDPC decoder on GPU where a the decoding schedule is purely sequential, since in high order fields, there is a high level of parallelism exposed at the GF(2^m) dimension. Comparison of these works with our proposed non-binary decoders on the GPU engine highlights that equivalent decoding throughputs and latencies can be obtained using distinct approaches on the GPU engine. In other words, both their works and the one herein presented on non-binary LDPC decoding on GPU stress the complexity of the problem, exposing two sources of bottleneck. The computation of configuration sets can be largely avoided by moving to the Fourier domain. However, this introduces the computation of the FWHT, in its turn, the major source of bottleneck in this domain, with over 45% of the decoding devoted to domain crossing[4].

Summary The proposed FFT-SPA GPU decoder extends the methodology proposed for the binary decoding case to the non-binary one. Due to the GF(2^m) definition, the data structures and memory layouts are scaled to accommodate the introduced GF(2^m) dimensions. The decoding throughput and latency show that modest performance is possible, nonetheless, sufficient to deal with lower data rate applications.

4.4 GPU-cluster Decoders

The tremendous boost that GPUs acceleration provides for Monte Carlo BER simulation[241] can be stepped up by moving towards the acceleration on GPU clusters[1,2]. Due to the layered composition of the distributed compute resources of the GPU cluster, discussed in Subection 4.1.4, a bottom-up approach to the Monte Carlo BER simulation is feasible. Thus, the decoder realizations discussed in the previous section, can be further composed into a BER simulator system. Then the single-GPU environment on which they operate can be expanded onto the multi-GPU distributed environment. To the best of our knowledge, it was the first time that a Monte Carlo BER on GPU has been scaled to GPU-cluster execution[1,2]. Wang *et al.* drive a quad-GPU system using a multithreaded CPU approach in order to realize a real-time LDPC decoder for WiMAX codes[156]. Even though very high decoding throughputs can be achieved with this approach, tremendous energy consumption refrains the use of a cluster as a replacement for dedicated ASIC or FPGA hardware decoders.

4.4.1 Fast BER Monte Carlo Simulation

As mentioned, the development of BER has been performed in two-phases. First, we define a methodology for the BER simulation operate efficiently in a single-GPU. Then the single-GPU system is expanded for multi-GPU execution, through the use of MPI, so that a SPMD execution model can be applied, as discussed next in Section 4.4. The motivation for the fast computation of the BER curves of LDPC codes and decoding algorithms combinations is obvious. There is a finite precision to the estimation of code thresholds put forward by *density evolution* (DE) or *extrinsic information transfer chart* (EXIT) charts. Thus, empirical evaluation is required, especially when the actual Tanner graph structure, decoding algorithm and system-level design features, such as data representation and numerical approximation of functions[79], are not taken into account by such models. The empirical evaluation is typically performed under Monte Carlo simulation. Often, instead of testing random codewords, the all-zero codeword, a valid codeword for all linear block codes, is evaluated multiple times through a communication channel. The simulation ceases its evaluation of a specific SNR condition whenever enough data has been found. Usually, as rule of thumb, 100∼200 invalid codewords and 1000∼2000 error bits must be found prior to stopping the simulation. For very low BER levels, such as when LDPC codes fall into their error floor region, a cap is usually set so that the simulation stops guaranteed that no errors exist at that threshold adjusted to the selected precision.

Hence, a greater drive for deploying large computational power levels to this problem is motivated by error-floors, that on LDPC codes can become double-edge swords, especially for long block lengths. They might lie further beyond what is required by a wireless communication application, but the LDPC code performance might bottom out just right after that level. In particular, while error-floors come at $\sim 10^{-8}$ for short to moderate lengths (Wi-Fi or WiMAX codes), they can be found in under $\sim 10^{-12}$ for long lengths (DVB 2 codes). E.g., for a block length of 64800 bits, a 10^{-12} BER level with enough statistical significance requires that 1000 bits have been found at least, which places the number of required simulated bits at $\sim 10^{19}$. Clearly, if the error floor probing is to see the light of day in an affordable execution time, superior computing power must be deployed. To this end, some authors have deployed a distributed FPGA system[196] to improve the BER simulation execution time. However, their attained FPGA decoding throughputs are inline with those obtained with single-GPU that due to their programmable processor are easier to target.

The system level diagram of the BER simulator is presented in Figure 4.26 it extends the communication channel system diagram (c.f. Figure 2.1) with other blocks pertaining the extraction of statistical data from the Monte Carlo BER simulation.

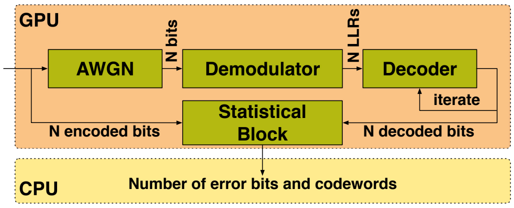


Figure 4.26: BER simulation model: the `AWGN` introduces noise to the generated codeword; the modulated bits are brought to the `demodulator` that computes LLRs for delivery to the `decoder` system; after its completion the `statistical block` keep track of how many bits and words were wrongly decoded.

AWGN and Demodulator Modules The proposed system leaves room for the inclusion of an `encoder` block that generates random codewords. However, to simplify the codeword generation process, and to accelerate the simulation since encoding is a sequential task ill-suited for GPU acceleration and, while well-suited for CPU, the CPU is in control of simulation system and is preferable to have it spin on GPU tasks, rather than GPU tasks have to spin on CPU tasks. Thus, the system generates the all-zero codeword (which can be generated by the appropriate `memset()` operation) already in its modulated stage, i.e. having seen the zero bits be assigned with the modulation corresponding quadrature and in-phase components[242].

Then, at the *additive white Gaussian noise* (AWGN) channel, a *parallel random number generator* (PRNG) with a very long period is employed to generate noise. To this end, the PRNGs defined in the *cuRAND library* (cuRAND) API[243] can be deployed. The XORWOW generator is particularly interesting, since it provides a tradeoff of smaller period (2^{90}) with faster generation when compared to the more powerful Mersenne Twister (period of $2^{19937-1}$)[244]. The XORWOW PRNG has the ability to move ahead to the k-th sample without requiring to draw all those samples. This ability brings high versatility to the PRNG to be utilized under SPMD execution model, since threads in different devices can sample the same pseudo-random sequence in the Monte Carlo simulation.

LDPC Decoder Module The LDPC decoder has been defined in the previous section. Since the decoder I/O interface comes in the form of GPU buffers that consume LLRs and produce decoded bits, any decoder whose implemented algorithm is LLR-based and sees the hard-decoding functions realized can replace this module. In addition, there is no restriction to the thread-granularity introduced in this block, although the finer the granularity, the quicker the performance will peak for a reduced number of codewords set for the data-parallelism level. The opposite holds true, the coarser the granularity the higher the number of codewords required to reach the peak throughput.

As a consequence, a tradeoff in thread-granularity is introduced by the decoder to the statistical block. In the previous section, we have discussed that the higher the number of codewords packed within an execution grid, the better the decoding throughput performance, even though latency is driven upwards. For the simulation, there is no restriction to how much latency can be tolerated and since the Fermi-GPU has a limited capability to hold CUDA streams, we cannot define multiple streams to handle a sufficiently large number of codewords that reach the peak decoder performance, that also keeps latency low[156]. On the other hand, if too many codewords are packed onto the same execution stream, the statistical block will have to perform batch evaluation of the errors produced after each execution grid has been simulated.

Statistical Module The statistical module is responsible for the evaluation of how many error bits, and codewords, have been sustained throughout the channel transmission after the *forward error correction* (FEC) system. In the particular case that the all-zero codeword is being used, the statistical block only has to compute the sum of all bits to gather how many were wrongly decoded. In addition, to find out if a word has been incorrectly decoded, detecting a single bit set to 1 suffices. However, the GPU engine is better off at performing an evaluation of the complete decoded words instead of early terminating when a 1 is detected. Thus, we overload the reduction primitive with two distinct inner kernels to find the number of error bits and words as follows

$$\text{No. errors}\{bits, codewords\} = \left\{ \sum_n H_w\left(\hat{c}_n\right), H_w\left(\underset{n}{+}\hat{c}_n\right)\right\}\ \forall n, \tag{4.3}$$

with H_w the Hamming weight and $+$ the bitwise OR operation (in this case, $\underset{n}{+}$, employed similarly to the summation operator, $\sum_n$). The advantage of kernels is that they can be applied vector-wise, e.g., $p-1$ LLRs packed in a vector type corresponding to $p-1$ different codewords, that are hard-decoded into a $p-1$-bit word array allows for a compact evaluation, provided $\hat{c}_n \leftarrow \left\{\hat{c}_{n,0}, \hat{c}_{n,1}, \cdots, \hat{c}_{n,p-1}\right\}$, where $\hat{c}_{n,i}$ is the n-th decoded bit of decoded word i. The Hamming weight $H_w()$ is an intrinsic instruction, available through the `popcount()` primitive that returns the number of set bits in a single clock cycle[188]. If an encoding system is introduced to the BER simulator, the reduction kernels are then updated to

$$\text{No. errors}\{bits, codewords\} = \left\{ \sum_n H_w\left(c_n \times \hat{c}_n\right), H_w\left(\underset{n}{+}c_n \times \hat{c}_n\right)\right\}, \tag{4.4}$$

with × the bitwise exclusive-OR operation, and c_n the state of encoded bit n.

While the vectorized statistical gathering of data is suitable for maintaining high-levels of data-parallelism, there is a drawback in statistical biasing of the results, in particular, of the average number of executed iterations. Since the maximum number of iterations is not usually required, a genie-aided early termination scheme can be employed to make the decoder cease the issuing of a new decoding iteration when all the codewords that are in the data-parallelism level kept at the vectorized element have been correctly decoded. To do so, we resort to the knowledge of the encoded word, and simply feedback the statistical module information onto the decoder decision to issue new codewords. Whenever the statistical module finds that all codewords have been correctly decoded, then the Monte Carlo simulation will proceed to a new iteration. The biasing occurs because the worst decoding case packed within a vector is the one determining the number of iterations issued for all the 16 codewords. However, separating the data so that each codeword can be independently evaluated for errors and unpacking it from the vector datatype and packing a new codeword adds too much overhead to the simulation. This is partially motivated by the poor bandwidth achieved by reduction of a single bit array. The least bitwidth under which operations are efficiently performed is 8-bit integers, which by themselves, make the ALUs active using only a quarter of their bitwidth[170]—leading to a 10× increase in the execution time of statistical module, let alone the repacking procedures required. The greatest source of biasing occurs for when an error is detected on the error-floor regions, since the required decoding iterations is at a low level, and overall, since only one error is detected per a very high number of successfully decoded bits, this influence is not relevant, though it exists.

4.4.2 GPU Cluster Execution

As previously mentioned, the Monte Carlo BER simulation on the GPU cluster follows the SPMD execution model. First, for the binary case, the finest-granularity level set is for TpN decoders executing 16 codewords at the time. Not only does this permit to explore the maximum alignment for which the GPU memory engine bandwidth peaks, but also in conjunction with the TpN approach allows to keep the GPU thread engine fully occupied. While under shorter codes the performance penalty can be non-negligible, this difference is slighter for long length block codes, as seen in the throughputs obtained for the short block length codes III and IV and for the longer code of dataset I-a) (c.f. Figure 4.18). As a consequence of this design decision, it makes little to no sense to have GPUs communicate with one another in the simulation, since the communication via the PCIe buses would incur in tremendous overheads. Hence, the SPMD model, whereupon each GPU handles its own BER simulator that executes a static workload defined by the MPI rank 0 process. A K number of codewords to be simulated in the cluster are divided between all the P assigned GPUs and the PRNGs, independently operated by each compute node, are configured so that they sample non-overlapping sequences of the same pseudo-random number sequence[243].

One of the challenges to be overcome with the execution on a distributed system is the ability to see the execution time scale linearly with the number of employed GPUs. An advantage to the SPMD model employed is that we can contain communication to a minimum. In fact, communication exists in the beginning of the simulation in the form of the sending of the MPI initialization routine and the access of the MPI nodes to the arguments that define the simulation. Then at the end, each MPI rank outputs its results as soon as it finishes to an exclusive `file stream` that handles the compute node log, and once completion of all ranks has been communicated to the master, the master merges all the logs into a single one.

Experimental Results To showcase the tremendous performance boost we can accomplish for the Monte Carlo BER simulation of LDPC codes, we benchmark two distinct LDPC decoders under cluster decoding and benchmark the performance and scalability of the simulator for one of those cases. The employed GPU cluster is summarized in

Table C.1. It consists of a 1 master to 16 compute nodes dual-GPU cluster. The cluster performance is profiled for dataset I-a) (c.f. Table 4.3)[b].

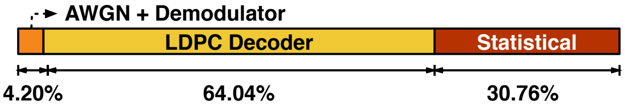
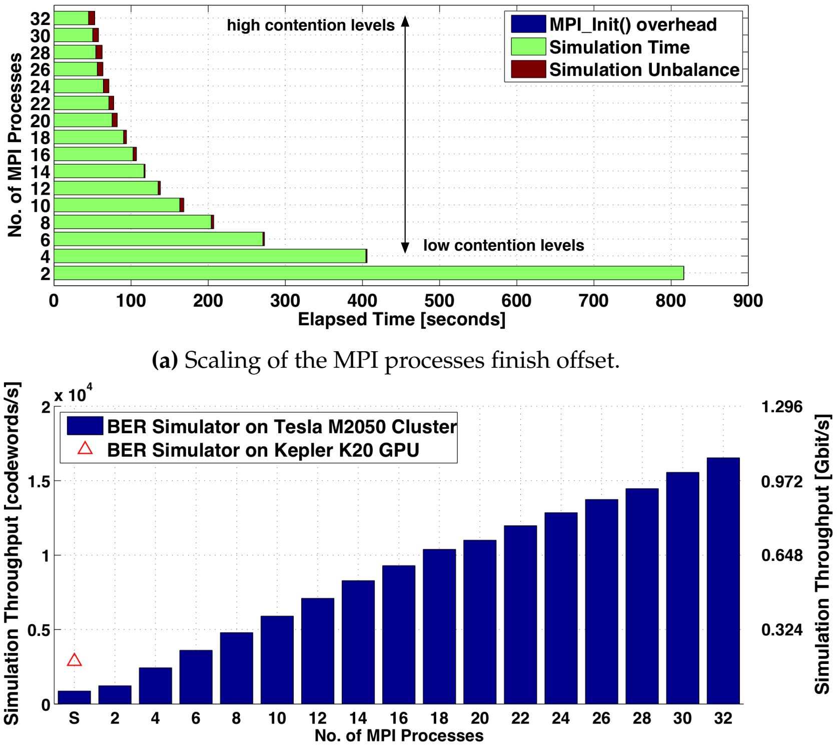


Figure 4.27: Cluster BER simulator kernels relative occupancy at SNR conditions of 3.0 dB (8.51 issued iterations in average).

GPU-cluster BER Simulator Performance We have analysed the cluster simulator performance regarding its scalability, throughput and distributed-aware execution. The DVB-S2 rate 1/2 code BER performance was run on the GPU cluster for a SNR of 2.0 dB, which lies well within the error floor of the LDPC code, yielding 17.42 iterations issued in average, for 2 to 32 GPUs. The data collection methods *i-iii*) suitability under the SPMD model is discussed in face of the obtained results. In Figure 4.28a)), we measure the asynchronous behaviour of the processes running in the MPI distributed environment. For profiling the developed BER simulator, the GPU-cluster ran exclusively the simulation, implying a low contention level of the cluster system. Asymptotically, the BER simulator will collect bit and codeword error levels that are uniformly distributed across the MPI processes. However, if the simulation is to test a low number of codewords across several GPUs, then the lower the number of codewords and the higher the number of GPUs, the greater the computational unbalance. This is measured as amount of time taken between the first process to finish, the one that encountered the least errors, and the last process to process, the one that encountered the most. We make this assumption without considering external interference from other computational tasks, although, for difference reasons, this measure of the computational unbalance would still act as a constraint to the final step of the BER simulator, gathering and computing each node BER, involving communication across the cluster interconnection network. In fact, as seen in Figure 4.28a)), the time difference between the first process to reach the `MPI_Finalise()` function and the last one is lower when the number of MPI processes of the BER simulator is low, which means that either overall data collection strategy *i*) or *ii*) are well-suited to compute the final statistical results on the cluster system. Under higher contention we could rely on strategy *iii*) to avoid spinning on the final MPI gather or reduce routines.

The BER simulator execution on the Kepler G4 GPU achieved a 3 times speedup when compared to the Fermi G3 GPU, as seen in Figure 4.28b). The developed BER simulator is,

[b] In addition to this dataset, the BER simulator herein described has permitted the evaluation of the BER performance for datasets I-a) and I-b) under different decoding algorithms and memory systems for other works with the works communicated in [3,15,16].



(a) Scaling of the MPI processes finish offset.

(b) BER simulator scalability across a variable number of GPUs.

Figure 4.28: MPI execution overhead and computational unbalancing scalability running 10^6 codewords at a SNR of 2.0 dB: a) the MPI spawning overhead is hardly perceptible, unlike the computational unbalance that increases with the number of processes spawned; b) throughput in codewords simulated per second, the decoder running on a single Fermi(G3) and Kepler (G4) systems are shown for reference (S stands for single-GPU).

Table 4.10: Cluster simulation time and speedup running 10^6 codewords at a SNR level of 2.0 dB.

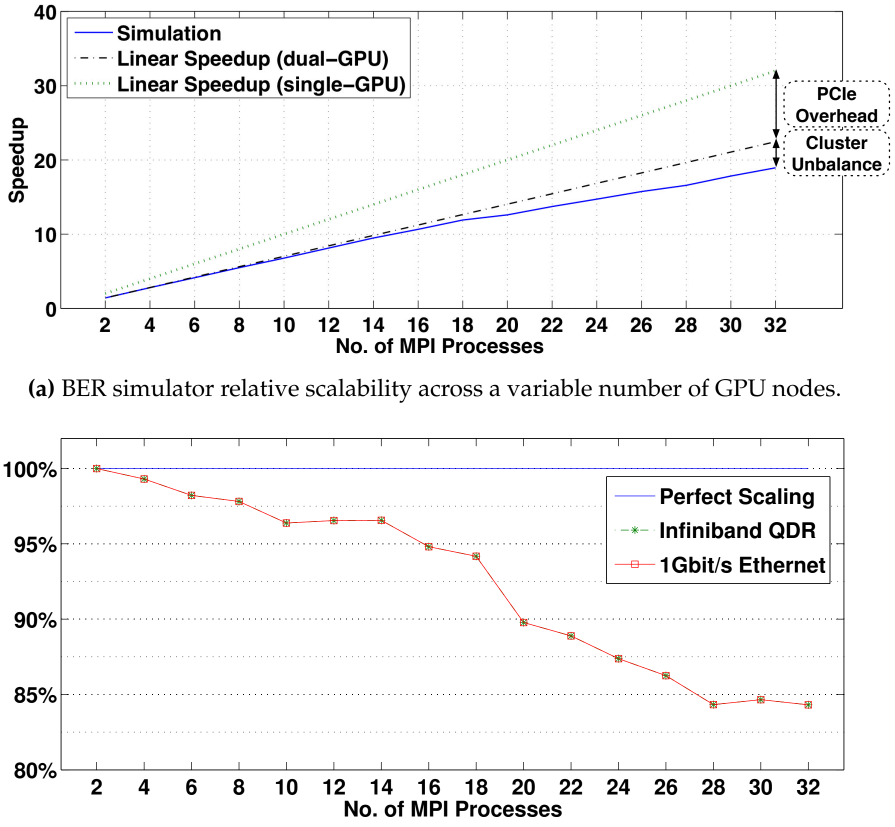
No. of GPU	1	2	4	6	8	10	12	14
Speedup	-	1.4	2.8	4.1	5.4	6.8	8.1	9.3
Simulation Time [s]	1145.8 348.8*	816.16	411.7	277.0	210.89	168.4	141.0	123.4
GPUs	16	18	20	22	24	26	28	30
Speedup	10.7	11.7	12.7	13.9	14.6	15.9	16.7	18.2
Simulation time [s]	107.3	98.0	90.4	82.6	78.4	71.9	68.7	63.0

* stands for single Kepler G4 GPU execution.

thus, able to scale on newer architectures, allowing us to speculate that the BER simulator execution time would be further reduced if a Kepler GPU cluster setup is utilized.

GPU-cluster Scalability The scalability exhibited by the cluster simulator scales sub-linearly to the number of GPUs employed, as seen in Figures 4.29a)) and 4.29b)) and also in Table 4.10. The sub-linear scalability is due to the cluster environment as the usage of a dual-GPU node does not see a 2 times speedup but a 1.4 speedup. This is the result of the MPI environment and also of process interference in the same node. Figure 4.29a)) shows the measured simulator scalability with two distinct speedup scenarios. The highest slope curve assumes a perfect scalability of the BER simulator extrapolated from the single-GPU BER simulator performance, i.e. the BER simulator would operate at $k \times T_{c,1}$ when using k GPUs. The other curve assumes a perfect scalability of the BER simulator extrapolated from the dual-GPU performance, which corresponds to the performance of a single dual-GPU node. Hence, for this case we assume the BER simulator would operated at $k \times T_{c,2}$ with k the number of used dual-GPU nodes. In order to reach a fair conclusion regarding the cluster efficiency, the scalability of interest is that of the dual-GPU. Consequently, we can measure the relative efficiency of scaling the BER simulator on the cluster, which is shown in Figure 4.29b)). Under our experimental setup and developed algorithms, we find that the execution of the BER simulator using the Infiniband or the Ethernet card yields the same scalability, meaning that communications, although critical for some distributed applications, do not pose any issue. The scalability with the number of GPU cards involved, given the asynchronous execution flow, should follow a close to linear speedup. However, assuming that the single-GPU performance accounts to the full potential of a GPU engine, it was observed that $\sim$ 25% of the GPU potential power was not extracted. In order to infer what caused such behaviour, we ran the GPU locally on the node to compute the speedup values shown in Table 4.10, and also on the master node through the `mpirun` command. As expected, no significant overhead was imposed by the MPI routines. This leads us to a three-fold conclusion:

inter-process interference in the same dual-GPU node is responsible for a share of the sub-linear scalability, since only a 1.4 speedup is achieved with two GPUs; secondly, the random nature of the AWGN channel is responsible for the remaining share, as some processes faced with more unrecoverable errors will execute more decoding iterations than processes with fewer unrecoverable errors, and thus, take longer to execute. However, as previously said, the asymptotic behaviour of the BER simulator, i.e. when the number of simulated words is extremely large per GPU, will see a distributed load of unrecoverable and recoverable simulator words per process, and along these lines, the asymptotic computational unbalance is not a large contribution for the sub-linear scalability. In the former conditions, and if the BER simulator executes concurrently with other computationally intensive tasks, the computational unbalance between MPI processes will be due to the computational resource sharing between processes.



(a) BER simulator relative scalability across a variable number of GPU nodes.

(b) GPU cluster scaling of the MPI BER simulator efficiency.

Figure 4.29: Cluster scalability compared to single- (non-MPI) and dual-GPU (MPI) execution: a) the speedup achieved compared to a perfect scaling single-GPU decoder performance and also compared to perfect scaling of the dual-GPU node performance; b) scalability defined as efficiency of resource utilization with regards to dual-GPU execution.

An important performance parameter of the BER simulator, and that of applications which follow the SPMD execution model, is the scalability across nodes on the cluster.

We represent the scalability of the BER simulator, taking the single-GPU simulator as reference, in Figure 4.29a), and its efficiency regarding a perfect scalability in Figure 4.29b). From it, we can discuss how the dual-GPU configuration influences the scalability. As observed, even for dedicated PCIe buses, there is a considerable reduction in the simulation throughput when using both GPUs simultaneously than when using a single GPU. Therefore, the attainable theoretical peak is not bounded by the number of GPU devices, but rather the number of devices weighed by this reduction. With regards to this new theoretical peak, we can then estimate the efficiency of the GPU cluster using both the 1Gbit/s Ethernet and the Infiniband QDR interconnection networks. The lower latency response and data rate of the Infiniband interconnection does not outweigh that of the Ethernet for the Monte Carlo simulation. In fact, under both networks, the efficiency attained drops to $\sim$84.3% using all the GPUs in the cluster[2].

With regard to a corresponding simulation time on a GPU-cluster for a variable number of tested codewords, we show on Figure 4.30 the feasibility of testing $10^{\{14,15,16,17\}}$ bits, using the case study DVB-S2 code at a AWGN channel SNR of 2.0 dB. From the execution time shown in Figure 4.30, we are able to define dataset size limits to the reasonable utilization of a GPU cluster. The impractically of simulating a very large data set, such as 10^{17} tested bits, is well illustrated, as the time required to evaluate the BER for a particular SNR would involve a colossal near-decade of computation. Assuming that 3 months of computation is reasonable enough to compute a single BER data point, then we can define upper bounds to the utilization of GPU clusters in the context of the BER simulation. Specifically, 10^{15} tested bits is feasible within these bounds. In fact, testing 5×10^{15} would take slightly longer than our assumed reasonable maximum execution time. Such large datasets are necessary to the evaluation of error floors, while the BER of a particular code in its waterfall SNR range, can be readily inferred in a much lower number of simulated bits, bringing the execution time from a minutes to just a few seconds. Hence, the utilized GPU cluster, is able to fully characterize the BER performance of LDPC codes in waterfall SNR ranges and in error floor SNR ranges, up to close to 5×10^{15} simulated bits.

As seen in Figure 4.30, migration of the Fermi to the Kepler generation of architectures permits the bringing of a year-long computation workload in a single-GPU to just above the week range in a similar equipped 32-GPU cluster with 16 compute nodes. As the scaling down of devices technology nodes is foreseen to continue to elevate the computational performance of processors, it is expectable that the proposed methodology will also continue to scale with it[1].

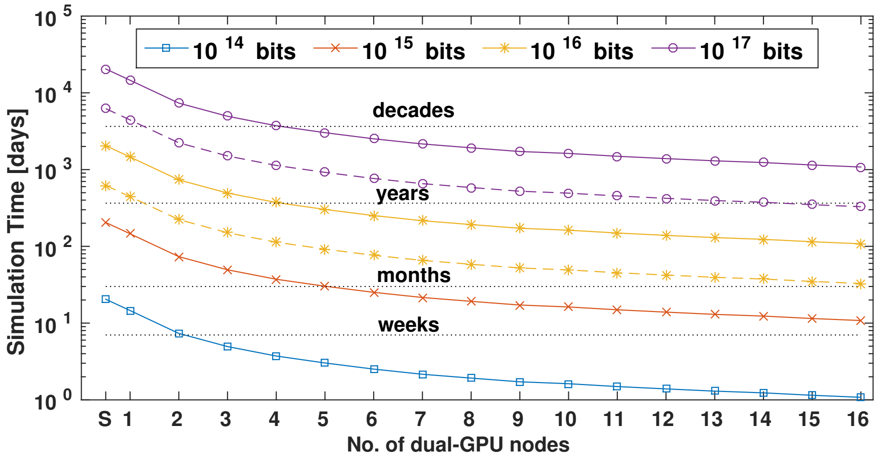


Figure 4.30: Cluster simulation time scenarios: time undertaken simulating $10^{\{14,15,16,17\}}$ DVB-S2 codewords. The execution times are presented for single GPU execution (S) and for 1 to 15 dual-GPU nodes with the *solid* lines representing the Fermi nodes and the *dashed* lines the projected gains with an equivalent Kepler configuration.

4.5 Hybrid CPU/GPU Decoders

Hybrid CPU/GPU architectures are particularly relevant to address one of the key challenges with GPU computing. As GPU cards are discrete devices connected through a PCIe interface on a host computer system, despite their high global memory bandwidth, the device and the host memory space are physically separated by a lower bandwidth bus (PCIe). As a consequence, cooperative computation can only boost performance for as long as each device can execute their workload while the other is also executing their share and, still, room has to be made for moving data from the host or the device memory space, depending on the DDG task dependencies. While the movement of data from CPU to GPU has been addressed by using streams, provided under asynchronous execution in CUDA and OpenCL, there is no general streaming interface to support joint CPU/GPU execution when devices are discrete. On the other hand, the Ivy bridge Intel family and the introduction of *AMD accelerated processing unit* (APU) devices has lead to the integration of a GPU device onto the CPU die. This creates better opportunities to explore joint computation, since both devices are now physically on the same chip, and therefore, share the same physical memory addressing space.

The most fit for purpose model to explore this type of devices is the OpenCL programming model, as it is able to capture under the same platform the two different devices. While there is still a logical distinction between the memory addressing space of the CPU driven as an OpenCL device, of the GPU and the host system (which is the CPU), in reality the physical addressing space is the same. The CPU OpenCL device can have its buffers defined to reside in the host memory and so no data movement or replication of data ensues. Also, the GPU buffers can be allocated accordingly but then mapped to the CPU memory space, or the other way around depending on the manufacturers' OpenCL implementation of this feature. The two-fold result is 1) the so-called zero-copy occurs for the CPU OpenCL device which sees no data transfer on its memory space, and 2) the GPU accesses the host memory as well. Under the Intel Ivy bridge architecture, utilized in the proposed CPU/GPU LDPC decoder methodology, data is shared at the LLC, which is the L3 as seen in Figure 4.2a).

Relation to CPU-based LDPC decoders Falcao *et al.* proposed OpenMP decoders for regular LDPC codes attaining decoding throughputs low decoding throughputs for a quad-core CPU architecture (∼2 Mbit/s)[124], while under the OpenCL model, the same regular codes see an improvement to ∼7 Mbit/s[186]. Under the same programming model, Grönroos *et al.* are able to push the decoding throughput to ∼40 Mbit/s, only at the cost of increased data-parallelism and latency[128]. While Le Gal *et al.* using SSE and AVX extensions were able to push a quad-core design onto a range between 168∼533 Mbit/s at relatively low latencies (0.46∼2.40 ms) for short to moderate length codes[165]. All these approaches use the TpN fine granularity, except for the latter which relies on the *core-per-codeword* (CpC) parallel-expression. However, these approaches do not the case where the CPU is used in conjunction with a GPU system for heterogeneous computing of the LDPC decoding algorithms.

4.5.1 Potential of the CPU Co-accelerator

Having surveyed the state-of-the-art in LDPC decoding acceleration on programmable architectures, with particular focus given to GPU computing, we can observe that a conservative approach is taken towards homogeneous computing as opposed to heterogeneous. While certain decoder realizations will not do so much for increasing the throughput, since they present one or two orders of magnitude of throughput performance below the GPU peak performance[124,186], other approaches highlight that we can be missing on ∼40 Mbit/s of decoding performance, which is well above the WiMAX required throughput. Also, if we consider that certain LDPC decoder might be constrained by power such that a power hungry top-notch GPU device is not used, then at the CPU/GPU-chip level with a *thermal dissipation power* (TDP) of 70 W, such throughput value can be as high as 50% of the processing power.

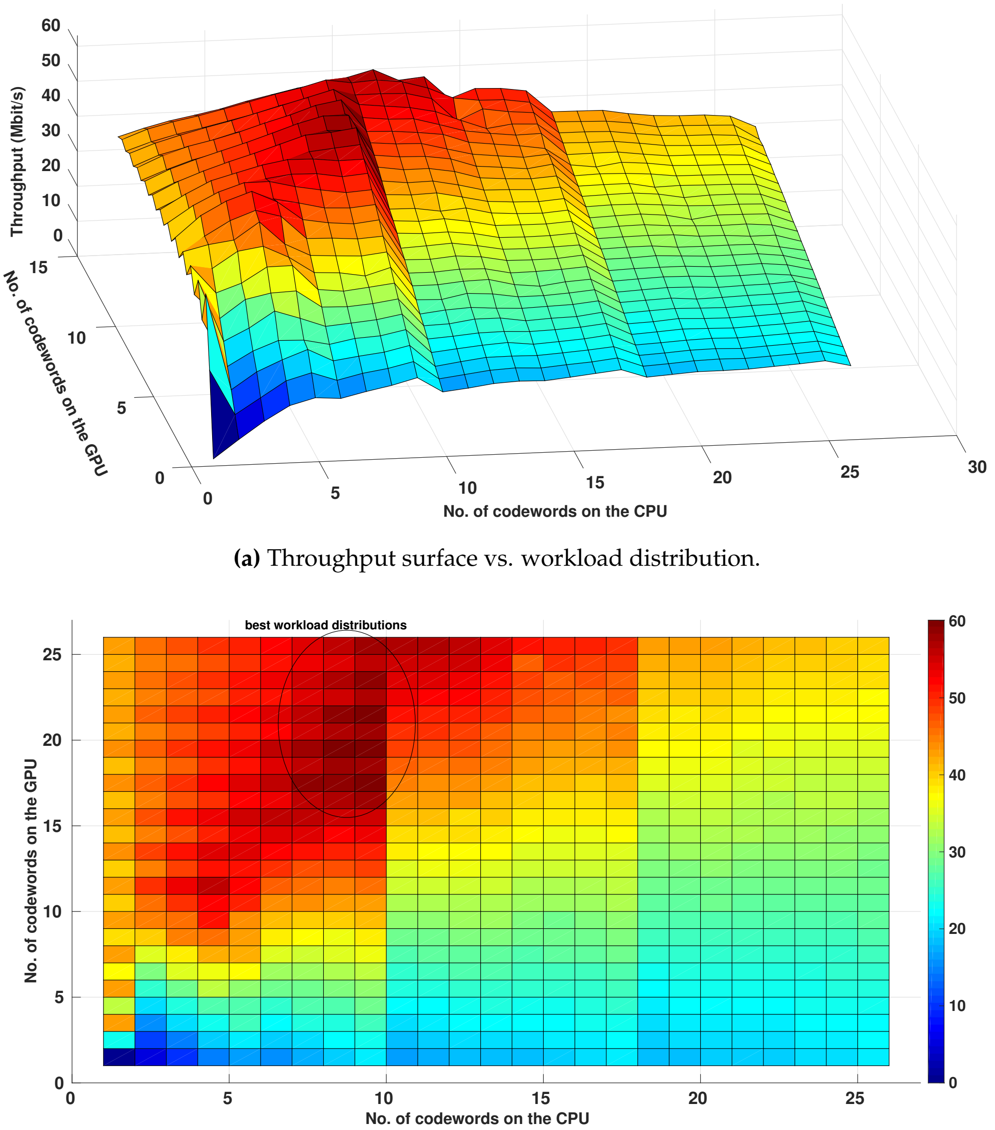
4.5.2 Experimental Results

To illustrate our point, we have established a simple experimental apparatus. If a known LDPC decoder design is taken and optimized for execution over the GPU, we can then use the remaining CPU cores that are spinning on the return of the GPU command queue instructions to perform a workload which takes exactly the same time that the GPU workload. Taking into account the lower logic resources that Intel and AMD CPU/GPU chips have available for multithreading than what Nvidia and AMD discrete GPUs have, the TpN approach is not as suitable as the BpC approach. For the one, on the CPU, each workgroup will be allocated to a CPU *hyperthread* while on the GPU the TpN decoder would see the serialization of numerous work-items. Also, since less work-items are able to be concurrently active on the GPU the *block-per-node* (BpN) quickly depletes the device capability to perform fully parallel execution at the defined data-parallelism. This is equivalent to saying that the performance quickly peaks under this approach for this device architecture.

The benchmarked code is the normal frame DVB-S2 LDPC code (tagged as dataset I in Table 4.3) under the BpC approach under the Intel OpenCL programming model, targeting the Ivy Bridge i7-3770k (C2 in Table C.1). Since the 128-bit alignment criteria holds for this device, the minimum data-parallelism level is set at 16 8-bit codewords packed onto `int4` datatypes. The finest data-parallelism level is then set at 16 codewords per codeword batch. To explore the data-parallelism design space, the CPU and the GPU handled all configurations possible for workloads of up to 25 batches per accelerator. The BpC approach benchmark emulates the *M*-factorizable architecture[232] assigning a work-item as if it were a FU in a 360-work-item workgroup configuration[10].

The resulting decoding throughput design space surface is presented in Figure 4.31 a) and its color map in Figure 4.31 b). As observed, in the color map, the most suitable workload distribution occurs when the latencies of the GPU and the CPU match. The top five-tier distributions are shown in Table 4.11 and show that a static workload distribution can be picked that allows for latency ratio to be close to 1. This ratio defines the latency of the CPU device to that of the GPU device. These workload configurations permit a 0.93∼1.04 ratio operation. However, to reach within these ratios, the GPU and the CPU need to issue a moderate number of codeword batches (7∼8 for CPU and 16∼20 for GPU) which elevates the total latency to the 400 ms range, leaving out its potential for real-time decoding. Notwithstanding, the optimal top-tier configuration showcase that 50% of the GPU performance is left untamed in the chip if the CPU is not employed for cooperative heterogeneous computation.

A limitation to further performance boosts brought on by the CPU OpenCL are related to the compiler ability to vectorize the instructions issued by the OpenCL parallel kernels onto SSE and AVX registers. Nevertheless, even when the compiler reports the successful vectorization of the instructions, the decoding throughput and latencies are much further than what is reported for an SSE-based LDPC decoder under the same



(a) Throughput surface vs. workload distribution.

(b) Color map of the throughput vs. workload distribution.

Figure 4.31: CPU/GPU decoding throughput (Mbit/s) vs. workload configuration (16-codeword batches per device).

Table 4.11: CPU/GPU top tier configurations.

CPU/GPU	No. of batches			Thr. (Mbit/s)	Latency (ms)			
Decoder	CPU	GPU	Ratio$^{a)}$		CPU	GPU	Total	Ratio$^{b)}$
Configurations	7	16	2.29	59.08	402.67	389.15	404.32	1.04
	8	18	2.25	59.59	414.29	398.53	414.78	1.04
	7	18	2.57	59.74	445.11	435.14	452.61	1.02
	8	20	2.50	60.02	450.77	484.57	483.79	0.93
	8	16	2.00	60.08	414.29	398.53	414.78	1.04

$^{a)}$ GPU to CPU workload ratio; $^{b)}$ CPU to GPU latency ratio

LDPC code. In fact, the decoding throughput and performances are off by a ten-fold factor[165]. As a consequence, some computation power can be unlocked by expanding the OpenCL LDPC decoder execution to the CPU with little code refactoring. However, to fully extract the performance of the CPU engine, low-level intrinsics must be utilized.

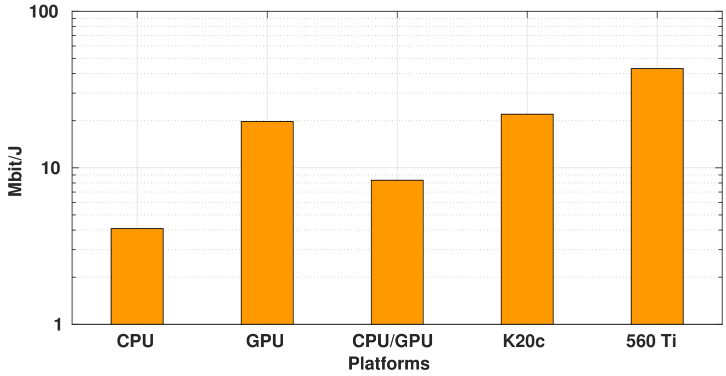
4.5.3 Energy efficiency of the CPU/GPU decoder

One figure of merit of decoders on programmable architectures is their energy efficiency that yields a metric on how many bits is the decoder system able to decoder under a fixed power budget. While the decoding throughputs achieved for the current generation of CPU/GPU systems is shown to be behind to what a GPU is capable, and also behind the full potential of the CPU computational power using a lower-level approach[165], to showcase the potential of including the CPU along with the GPU computation, we analyze this efficiency compared to other cases. Namely, against single discrete GPU decoders and against the execution of the decoder using the CPU, the GPU and both at the same time in the x86 hybrid die. This study is presented in Figure 4.32.

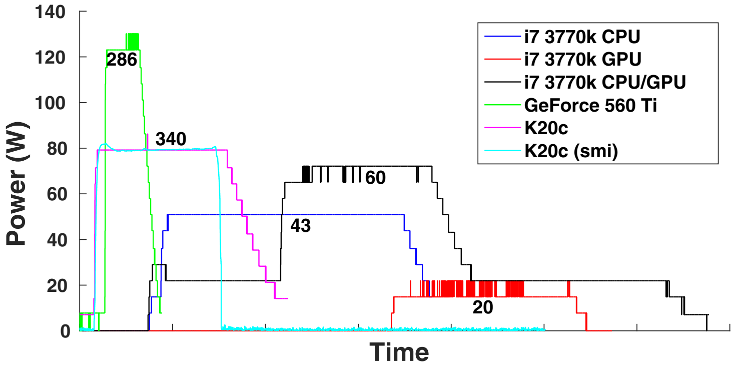
An interesting effect observed is that while the CPU and the GPU efficiency are one order of magnitude apart, the unleashing of more performance to the decoder system by activating the CPU to perform heterogeneous computation of a workload, the energy efficiency drops down. The other key feature is that the pure-GPU driven decoder shares the same order of magnitude of efficiency when compared to the discrete and more powerful K20c and 560 Ti GPU devices. Considering that a power budget constraint might apply in addition to a decoding throughput, forgoing the fact that latency is too high for real-time decoding in any considered case, we see that the inclusion of the CPU decoder elevates the power by almost 50%. Under a scenario where this would surpass the required budget, certain parts of the device must be untamed for computation. In other words, it is visible the ability of the current generation of CPU/GPU devices to put certain parts of the chip in a deep sleep state. As turning on or off the CPU and the GPU adds either $\sim$20 W or $\sim$40 W to the power drawn by the chip, this type of devices offers a better opportunity at managing several decoding throughputs met and power drawn levels.

We test the hybrid CPU/GPU decoder against a high-end scientific GPU (Tesla K20c, G4 in Table C.1) and a lower end gaming device (a GeForce 560 Ti device, G9 in Table C.1). The results have been found using the Power Sensor listed in Table C.1, and also querying the appropriate registers in the CPU and GPU hardware[c)].

[c)] The power registers in the Intel CPU confirm the measurement provided by the sensor.



(a) Energy efficiency of the decoders in Mbit/J (log scale).



(b) Power profile of the decoders execution (the figure near each profile is the throughput in Mbit/s).

Figure 4.32: CPU/GPU decoding throughput (Mbit/s) vs. workload configuration (16-codeword batches per device): a) CPU, GPU and CPU/GPU refer to the x86-based approach; b) shows the profile of the power drawn by each decoder (time scale not normalized).

4.6 Summary

In this chapter[d)], we have proposed methodologies to efficiently explore LDPC decoders on programmable architectures using data-parallel programming models. In particular, rapid prototyping of new decoding algorithms has been performed with the proposal of a bitwidth optimized data representation of the SCMSA[3]. Using the CUDA programming model we have shown the different tradeoffs regarding data-parallelism in the form of multiple codeword batch decoding and thread-parallelism using the TpN and the BpC granularity approaches[2,241]. The overall evaluation performed for GPU-based LDPC decoders has shown that the great potential for LDPC GPU-based decoding lies at fast Monte Carlo BER simulation for quick evaluation of deep error-floors[1,2]. Under the BER simulation motto, a bottom-up approach for the composition of a simulator under a distributed environment, in particular a GPU-cluster, was proposed. These methods are scalable across multiple GPU generations and allow the execution of estimated year-long computation in under a week[1]. Moreover, we have studied the realization of LDPC decoders for codes defined over GF(2^m) and discuss its suitable representation in GPU multithreaded execution under the TpE thread-parallel strategy. The importance of the FWHT in the context of GF(2^m) decoding was also established, to which high bandwidth methods that explore all the addressing spaces within the memory hierarchy are discussed[13].

Revisiting the initial questions posed at this chapter introduction, we find that *i)* the *high-level synthesis* (HLS) tool generation of the final architecture can lead to decoding throughputs and latencies on a par with dedicated *register-transfer level* (RTL), especially for the approaches that are most suitable for the decoding algorithm expression, i.e. the dataflow decoder. ii) The wide-pipeline decoder quality of results are not impaired by the HLS tool nature *per se*, but rather by the underlying model of computation which is less flexible than the dataflow and the loop-annotated one. We can, thus, state that the decoding throughput and latencies obtained for the wide-pipeline decoders are due to the way computation must be expressed which does not properly match the nature of the LDPC decoding algorithms. Finally, *iii)* the scheduling of operations that favors the decoding algorithms the most have turned out to be those where the designer had the greater control. For the case where computation was deeply pipelined, as in the wide-pipeline case, scheduling was properly performed, but when commuting from one kernel to another there is inefficiency of computation brought on by reduced usage of the pipeline stages on the trailing kernel and the starting one.

[d)] Portions of the work discussed in this chapter have been communicated as a book chapter and in international conferences[1–4,10,13].

5

Reconfigurable LDPC Decoders

Contents

In this Chapter, we discuss the importance of the reconfigurable computing field to achieve efficient *low-density parity-check* (LDPC) decoder designs that are able to deliver very high decoding throughputs and low decoding latencies, at lower power budgets than programmable hardware[175]. In particular, we explore *high-level synthesis* (HLS) approaches[245] that target certain types of template accelerators such as 1) dataflow-driven accelerators whose computation is performed in space and whose description is obtained from Java descriptions based on the MaxelerOS[178]; 2) loop-accelerated accelerators based on C/C++ languages using the Vivado HLS[179], the one allowing the most flexibility of optimizations deployed; and 3) wide-pipeline accelerators whose description is obtained from *Open Computing Language* (OpenCL) kernels[12,177].

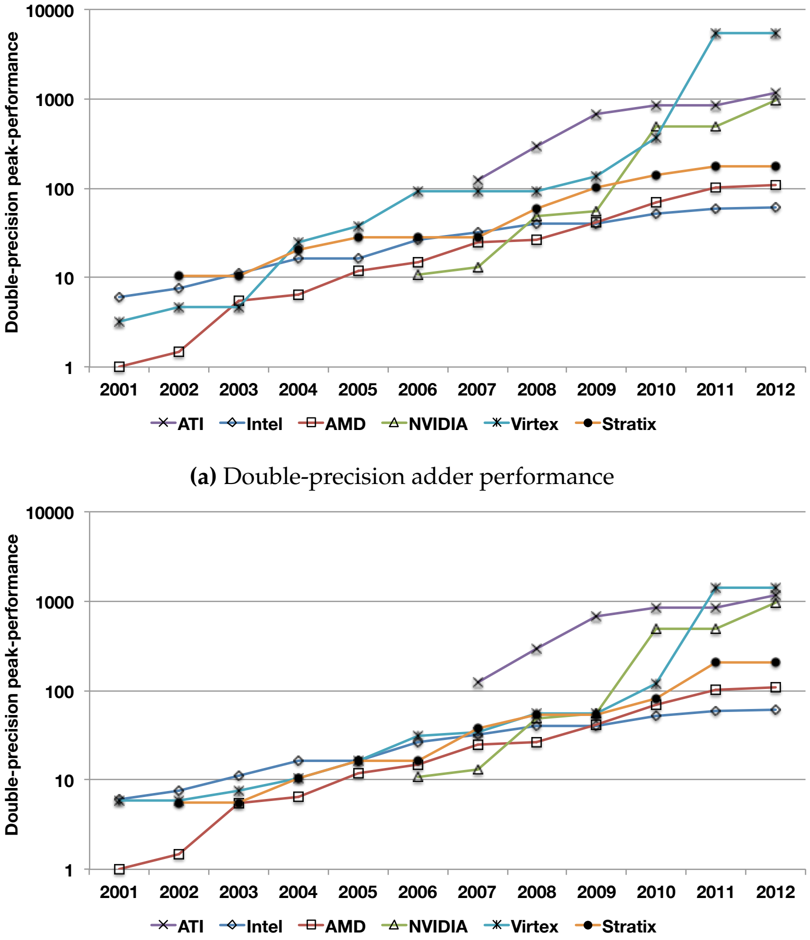
5.1 Reconfigurable Computing

*Field-programmable gate array*s (FPGAs) introduced in the mid-eighties[173] presented a larger capacity for "glue-logic" than their ancestor *programmable array logic* (PAL) devices. Within a decade, their growth in capacity made them suitable for logic emulation and prototyping[176], and soon their potential to customize a computer to a particular task was perceived. This opened the field of reconfigurable computing, with FPGA devices presenting a very promising alternative to low power *digital signal processor*s (DSPs)[246]. Since its dawn, it has been a very active field of research[175], inheriting a wide body-of-knowledge from disciplines such as custom hardware design, digital signal processing, general-purpose computing and *computer-aided design* (CAD)[56].

Reconfigurable computing is of renewed interest to LDPC decoding, as it allows the customization of an array device to the particular tasks required by an LDPC decoder. Due to their silicon density, FPGAs are able to play on a par with *application-specific integrated circuit* (ASIC) devices, both in terms of decoding throughput and latency, while offering at least a ten-fold lower power target than decoders on general-purpose architectures[56]. Their raw arithmetic performance has been steadily growing in the recent past, recently surpassing that of the *central processing unit*s (CPUs) and *graphics processing unit*s (GPUs), as observed in the adder and multiplier performance comparison shown in Figure 5.1.

5.1.1 Reconfigurable Architectures

Hardware substrates for reconfigurable computing span from fine-grained to coarse-grained arrays, with FPGA mostly composed of the former, while also integrated many widely employed computing blocks, such as DSPs and multipliers. For this reason, they provide very high flexibility of acceleration design, a highly appealing feature for both



(a) Double-precision adder performance

(b) Double-precision multiplier performance

Figure 5.1: Arithmetic performance comparison of CPU, GPU and FPGA devices: a) adder and b) multiplier.

scientific and industrial segments. FPGAs devices can be put to good use in host systems, providing a wide range of I/O connections (Internet in twisted pair or optical switches, SATA ports, PCIe power and other data pins). While the FPGA chip is manufactured standalone, it is also retailed integrated onto its own board package, or in some cases, it can be directly integrated onto a CPU socket on a motherboard.

Fined-grained Logic Unlike the traditional control flow computer architectures, whose general-purpose or specialized processors execute a group of instructions, FPGA are dataflow by nature. They possess a set of digital blocks that are configured on a circuit that performs a certain function. Nowadays, most FPGA devices combine both fine-grained and coarse-grained logic blocks—an overview of a typical island style FPGA architecture is shown in Figure 5.2—with *configurable logic block*s (CLBs), the finer ones, based on numerous *lookup-table*s (LUTs) available in small *random-access memory*s (RAMs). The CLBs, illustrated in Figure 5.3, perform combinational logic by storing a given truth

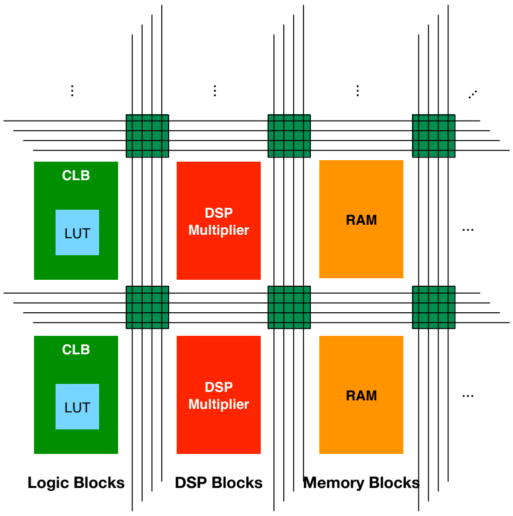


Figure 5.2: FPGA island style architecture. Functional blocks such as logic, DSPs and memory blocks are surrounded by an interconnection mesh that is routed according to the configuration to which the implemented algorithm is decomposed.

table and using the logic inputs as addresses to it. Sequential circuits are made possible by the existence of *flip-flop*s (FFs) at the output of the LUTs. Also, more complex functions can be broken down into chains of LUT configurations. The FF and LUT resources provided in the CLBs in conjunction with the DSP devices support massively parallel circuits that can attain very high bandwidths.

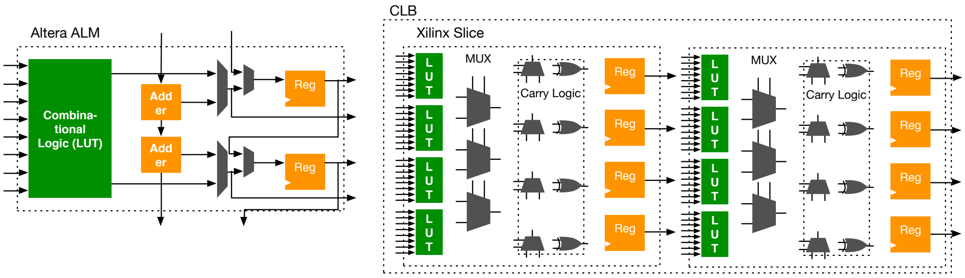


Figure 5.3: CLB example: Altera ALM and Xilinx double-Slice. The former providing an eight-input LUT, two adders, and registered outputs after a MUX chain, the latter providing a double-Slice design, each composed of 4 LUTs and 4 FFs. Current generations have 4 LUTs and 8 FFs. The CLBs can be configured to implement a certain combinational logic function, or a fraction of it in combination with other CLBs in the fabric.

Coarser-grained Logic Other coarser-grained resources exist in the FPGA fabric adding to its potential flexibility of design, namely, DSP units, usually in the form of multipliers. Their introduction has been responsible for the elevation of their raw arithmetic performance of FPGA devices[56]. Starting with 18×18 multipliers, the current generation of FPGA incorporates also 27×27 multipliers, allowing for full IEEE 754-compliant single-precision floating-point operation.

Among the remaining coarser-grained blocks available in the FPGA fabric, on-chip RAMs, usually in the form of *block RAM* (BRAM) elements are also available. Frequent accesses to external memory or an external interface can be avoided and data can be kept near the computation units. This way, a higher flexibility of design for a custom-tailored memory hierarchy exists, allowing FPGAs to flexibly overcome one of the great challenges of modern computer architectures, a widening gap between computational power and external memory bandwidth[167,208]. In most FPGAs, BRAMs are available throughout the FPGA islands in stacked columns that provide two physical access ports. Both major manufacturers of FPGAs provide BRAMs that can be configured as a single dual-port logical memory addressing spaces or as two single-port memories with half the size—20 Kbit or 2×10 Kbit in the Altera case, and 36 Kbit or 2×18 Kbit in the Xilinx case.

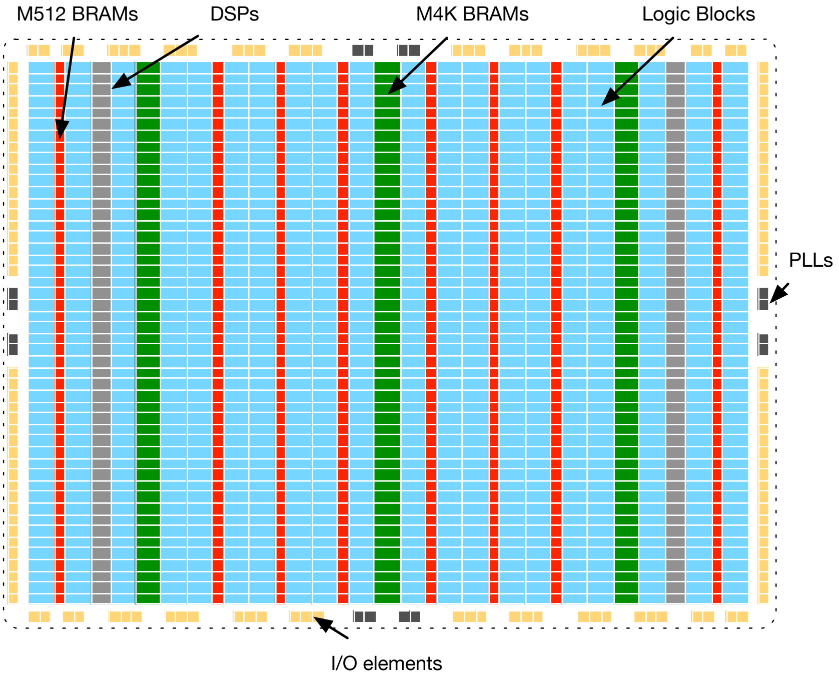


Figure 5.4: Stratix FPGA architecture. Resources such as DSPs and BRAMs are column-wise interleaved in the midst of logic elements for improved spatial proximity of resources. Other resources such as I/O pins and PLLs for clock signals are on the outermost side of the chip, surrounding all the remaining logic elements.

Furthermore, *system on a chip* (SoC) devices contain microprocessor cores, such as an ARM processor. The rationale is that increasing the heterogeneity of the FPGA fabric, each particular aspect of computation can be spread across the particular logic block which performs better the underlying task. This way, no ill-suited computing blocks are devoted to computation inside the dataflow accelerator[247]. The flexibility is maintained by the "glue-logic" provided by the fine-grained logic resources, while high performance is delivered from coarse-grained elements such as DSPs. In particular, the growing need of including a processing on-chip has lead to the embedding of hard processors and also of the availability of specialized microprocessor *intellectual property* (IP) cores to instantiate one using the FPGA FFs and LUTs, such as the Microblaze embedded processor[200]. This way, the necessity to integrate the FPGA onto a host system can be reduced by the mere necessity of powering up a board that already contains all the necessary memory, power, logic and I/O resources. For improved flexibility, some FPGA boards also allow PCIe integration onto a host system, in a similar fashion to how GPUs are integrated[177]. This model has been pursued for the FPGA boards that are used under the OpenCL programming model as kernel accelerators[177,200].

Programmable Interconnection All these fine- to coarse-grained resources are connected via a reconfigurable interconnection network that routes signals between the necessary logic blocks. Typically, resources are scattered around the FPGA die in small sets or groups, in column- or row- stacked fashion, interleaving the different logic blocks, so that there is a somewhat uniform distribution across the chip area[248]. As seen in Figure 5.2, each intersection of a row and column of logic blocks is provided with an interconnection box routing signals traversing from a logic block source to its destination across the chip. A commercial FPGA architecture is illustrated in Figure 5.4, showing the island-style distributed nature of the logic resources in the Altera Stratix FPGA family.

5.1.2 High-level Synthesis Programming Models

Traditional methods to design FPGA-based accelerators involve *register-transfer level* (RTL) descriptions, based on *hardware description language*s (HDLs) such as *VHSIC hardware description language* (VHDL), Verilog or SystemC, as a valid inputs to the underlying synthesizing infrastructure that ultimately generates the bitstream with which the FPGA device is configured in order to produce the desired computation from the data fed as input. The discussion concerning the pitfalls and disadvantages of each language remain partially out of the Thesis scope. Instead we consider that *non-recurring engineering* (NRE) costs associated with RTL projects make plenty motivation to seek alternative models for hardware description. For instance, if we consider the nested-loop structure in the `filter` function in Listing 5.6, we can argue on the feasibility of making fast adjustments on the micro-architecture design of the circuit generated. If a designer is willing to pipeline the filter kernel, so that in each clock cycle the generated hardware outputs a new element, explicit writing of the tree shown in Figure 5.5 would be required. This would entail writing the description of the outer loop fully unrolled so that the resulting accelerator would be pipelined into accepting a new loop iteration per clock cycle, thus computing

```
#define SIZE 64
void filter(float a[], float b[], float c[]){

  for(int i = 0; i < SIZE; i++){
    c[i] = 0;
    for(int j = 0; j < SIZE; j++)
          c[i] += a[i+j]* b[j];
  }
}
```

Listing 5.6: Nested loop structure of a filter function in C.

one iteration (one data element) per clock cycle. A change in the filter size would entail code refactoring of the `filter` RTL description, as would changing the output rate from one element per clock cycle to one every two cycles. This protracted design methodology could be enhanced with regards to code writing productivity.

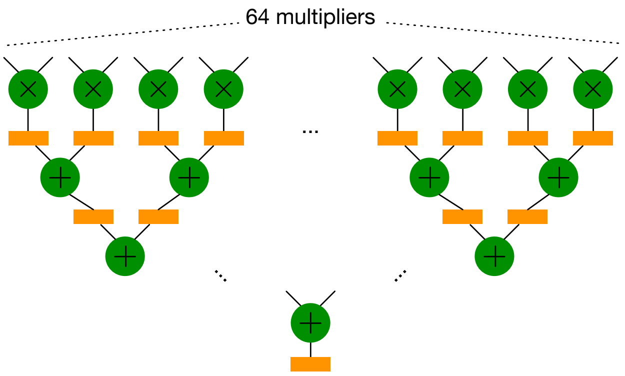


Figure 5.5: Fully unrolled `filter` function dataflow graph (Listing 5.6). By unrolling the outer loop in `filter`, each loop iteration can be computed per clock cycle.

RTL Design Flow What has been described earlier is a simplification of a formal chain of design stages under RTL design flow. As illustrated in the left of Figure 5.6, this process comprises numerous sub-design tasks, divided into various categories. The starting point is a user-fed specification of the algorithm functionality. If the algorithm is based on floating-point computation, such model is provided so that a fixed-point model is derived from it in order to save logic resources and keep the synthesized circuits simpler[186]. While Matlab is a popular choice for simulation, one can take a C, a *Compute Unified Device Architecture* (CUDA) or OpenCL algorithm functional description to perform the tasks concerning the system designer. After this stage, we step into the hardware design one, 1) where the accelerator micro-architecture must be elaborated, so that 2) its RTL design is performed using an HDL such as VHDL or Verilog, and finally, 3) area constraints such as floor-planning and timing constraints concerning clock domains and certain datapaths are defined. The project can, thus, be synthesized to gates, and to

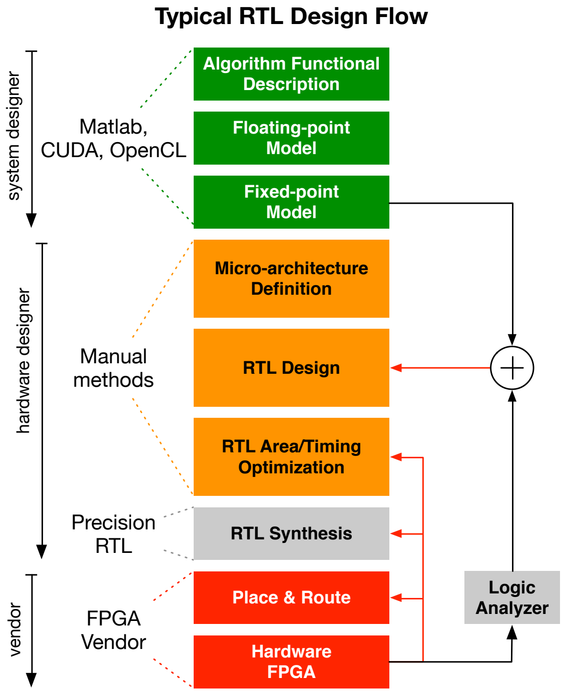


Figure 5.6: RTL design flow for hardware development.

generate circuits for the algorithm acceleration on the FPGA substrate. Next, the project steps into the vendor design stage, whereupon the synthesized circuits are placed and the datapath is routed on the resources available in a certain FPGA chip, or floor-planned chip area. While synthesis can be executed by a specific vendor tool, the process generates circuits from a hardware description, and thus, it cannot be included as tied to any particular vendor and is not drawn at the vendor stage in the Figure. Design details gathered from synthesis, placing and routing, are utilized, in conjunction with the fixed-point algorithm functional description, to simulate the generated accelerator behavior. This information can then be used to debug, improve or refactor the RTL design. By all means, this represents a slow and protracted way of designing hardware, with many inputs to refine the system leading to greater opportunities of circuit optimization, but also to error-prone decisions gathered from many design stage sources. Ideally, we would like to proceed directly from the algorithm functional description to the FPGA accelerator.

HLS Design Flow The necessity to overcome the high NRE cost and effort put into writing hardware accelerators via an RTL description has lead to the introduction of HLS tools[202,245]. A high-level language is used, extended to support hardware functionality not originally supported by the language through the introduction of hardware constructs or in the form of an *application programming interface* (API), so that code refactoring can share the same productivity as observed in software development models[245]. For

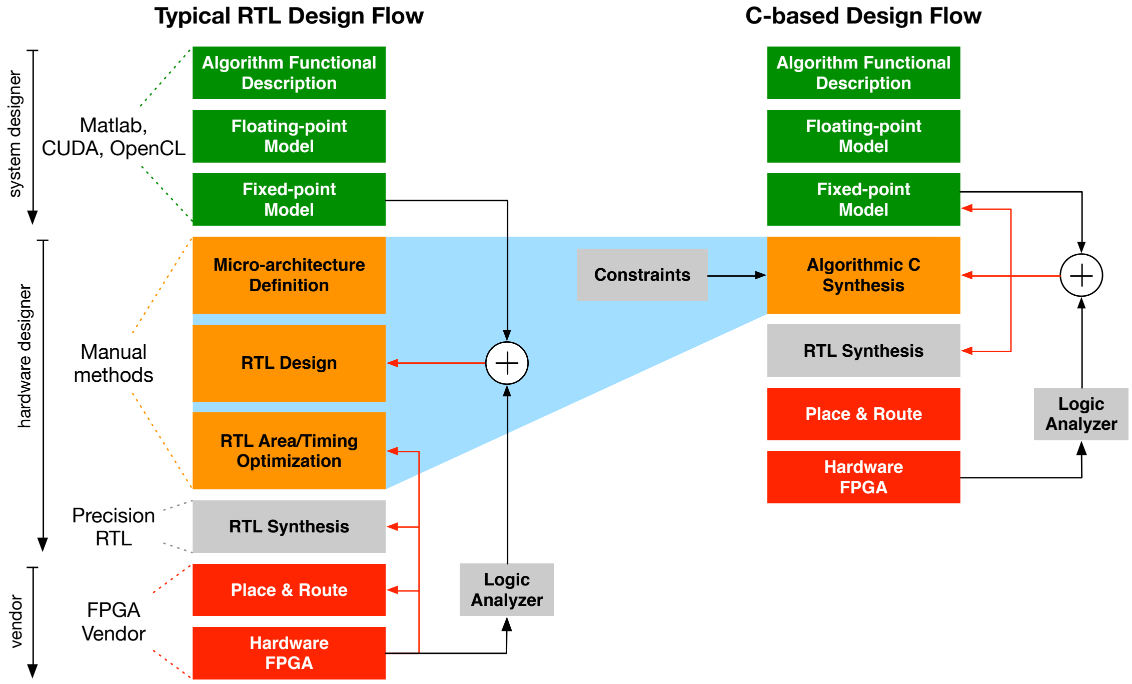


Figure 5.7: From RTL to C-based HLS design flow for hardware development.

instance, for the particular class of C-based HLS, illustrated in Figure 5.7, we can observe that the micro-architecture definition, RTL design and corresponding area and timing optimizations are replaced by a single algorithmic C-synthesis step. Naturally, there is no "genie-aided" conversion of C to gates. Instead, C-synthesis usually refers to the generation of an algorithmically equivalent RTL description to that fed as an input. To this end, most C-synthesizers use a number of RTL primitives for substitution of arithmetic functions, redefinition of functions as hardware modules, among other required translations.

Although hardware accelerators can always be obtained from a mature and stable HLS compiler, there are several functionally correct equivalents for each particular kernel description with regards to different optimization figures of merit (memory bandwidth, logic utilization or data throughput). However, without any designer guidance, a naïve version is generated. Thus, the synthesizer accepts designer-fed constraints to allow the generation of efficient hardware designs, based on it[202]. Usually this takes the form of 1) an API with functions that directly translate to a hardware block, for instance a FIFO, 2) through annotations in the code, typically in the form of directives, that guide the HLS tool to apply a certain feature to a particular block of code, or even 3) knowing beforehand that all code written within a function must expose parallelism through the supported language constructs as it will be broken down into a certain loop structure, as it happens with OpenCL.

Academia- and Industry-led HLS Tools According to the taxonomy introduced by Martin[245], we are currently stepping from the 3rd into the 4th generation of HLS tools. In the current generation, the prevalence of C-based tools is high, due to C popularity among the scientific community. Not only is this effort pushed by academia, but also by industry. BlueSpec is a toolset for both FPGA and ASIC design based on SystemVerilog. Behavior is expressed via *Guarded Atomic Actions*, a SystemVerilog extension of *finite-state machine*s (FSMs). The rationale behind BlueSpec is to improve productivity at the hardware design stage, instead of lowering the entry barriers placed by HDL via a familiar software language[249]. LegUp is an academia driven effort[250] to provide a HLS tool that generates an accelerator system from a C-specification. The C-synthesizer separates computation for circuits acceleration, while other data management and control functions are dealt by a soft-MIPS processor. This way a host system for the FPGA can be dismissed as its own system is included. The ROCCC tool, standing for the Riverside optimizing compiler for configurable computing, is also an academia driven effort providing a C to VHDL compilation tool[251]. Hence, it introduces a C-synthesis compilation step (Figure 5.6) upstream of the hardware design stages, rather than replacing them with a single C-synthesis stage as seen in the right-hand side of Figure 5.7. C-to-silicon by Cadence Design systems[252] uses SystemC to raise the abstraction level and, as argued, to improve the productivity of design by introduction of *Transaction Level Models*, and targets both FPGA and ASIC design. OpenCL models have also been getting traction, as seen with multiple academia driven efforts such as OpenRCL[253], a tool that generates MIPS-like cores to schedule fine-grained parallel threads; *Silicon-to-OpenCL* (SOpenCL)[254] generates a wide-pipeline architecture where threads become iterations in a deeply pipelined accelerator; and also industry-driven, with Altera and Xilinx supporting an OpenCL compiler for their respective FPGA and SoC devices[177,200]. The former, is the seminal industry-led OpenCL compiler, that alike the SOpenCL tool, generates a wide-pipeline architecture from an OpenCL input specification. Due to its cross-platform capabilities, the OpenCL standard is highly appealing, as the same code can be reused across CPU, GPU and FPGA devices. The Xilinx OpenCL compiler represents an addition to the Vivado design suite[179], which supported already C/C++ and SystemC inputs by the Vivado HLS tool. Therein, a C/C++ specification generates hardware exportable as an IP core for integration onto an RTL-based architecture for bitstream generation. Furthermore, CUDA for FPGAs has also been a subject of study, FCUDA allowed the generation of a custom FPGA accelerator from a CUDA kernel specification[255], and FASTCUDA generated a custom architecture based on multiple Microblaze processors[256]. Also, though not C-based, the MaxCompiler[178], by Maxeler Technologies, generates a *dataflow engine* (DFE) from a JAVA-written algorithm specification (instead of the C-synthesizer one would have a JAVA synthesizer in Figure 5.7 the C language is read, one would read JAVA for this case) that is pushed through all the HLS design flow stages until a valid bitstream is generated[178].

HLS Design Decisions HLS tools can be characterized by their ability to generate accelerators ready for execution or not. Some tools have a contained compilation flow, i.e., their range of design is limited and will not generate an accelerator for the FPGA instantiating all the required hardware blocks, such as clock and I/O interfaces and, consequently, lacking FPGA pin mapping. Herein lies a considerable productivity gap, since these tools require additional effort to be put into the design of the accelerator. After the algorithm accelerators circuits have been generated, they require integration onto a hosting system on the FPGA chip. For instance, C-to-Silicon and Vivado HLS work under this principle, the HLS mechanisms provided are targeted at augmenting the productivity of designing certain tasks in a global architecture but are not intended to describe the whole architecture. In the latter, the design can only be exported as an IP core for instantiation elsewhere. Other tools, on the other hand, relieve the developer from the need to design the full architecture. In these cases, such as the Altera OpenCL, LegUP and MaxCompiler, for instance, the output is a bitstream ready for execution. Requiring less effort but having decreased flexibility to instruct how the final accelerator architecture should be is one of the many tradeoffs concerning the use of a particular HLS model. In fact, careful evaluation of the following design decisions must be made in order to achieve high performance in the synthesized HLS accelerators. This takes into account that the more flexibility the more inputs are asked from the designer. However, the less the required inputs the more tied to the underlying HLS tool decision making heuristics will the accelerator be. Thus, we pose three key questions that our proposed methodology attempts to answer.

i) Should the final architecture be generated by the HLS tool, instead of hand-tuned by an experienced hardware designer?

ii) Is the high-level programming model suitable for the algorithm candidate for circuits-acceleration?

iii) Is the underlying accelerator architecture exploiting a scheduling of operations that favors the algorithm?

In order to address the challenge of producing efficient accelerators for LDPC decoders on FPGA devices using HLS models, we discuss how different HLS tools allow for certain characteristics to be better explored than others in the following sections of this chapter. To this end we target distinct accelerator architectures based on different

HLS tools. In Section 5.3, we explore how a pure dataflow model can be employed on binary LDPC decoders using the MaxCompiler. In Section 5.2, we discuss the employed LDPC codes for which the decoders were generated. In Section 5.4, we discuss how to proceed in the design space exploration towards a high performance LDPC decoders using C with directive annotations with the Vivado HLS tool. In Section 5.5, we analyze how wide-pipeline decoders can be developed for binary and non-binary LDPC decoders using the OpenCL programming model as the algorithmic input to SOpenCL and to Altera OpenCL tools. We also analyze how certain compiler optimizations work best in the generation of wide-pipeline accelerators, using the SOpenCL tool[12].

5.2 Synthesized LDPC Decoder Accelerators

To assess the performance obtained with the HLS-based reconfigurable LDPC decoders we utilize different LDPC codes and algorithms (c.f. Table 4.3). We divide the LDPC codes used to benchmark the proposed decoder designs onto many categories based on their 1) Tanner graph construction, 2) code length and 3) code construction. Considering that the methodologies discussed in the previous Sections for each decoder type—wide-pipeline, dataflow and loop-annotated—impose *a-priori* restrictions to obtaining very high decoding throughputs and low latencies depending on the code length, this criteria makes the most sense. Furthermore, the diversity of LDPC codes is not limited to those found on the encyclopedia of sparse graphs[184], nor to *LDPC Irregular-Repeat-Accumulate* (LDPC-IRA) of the *2nd generation DVB* (DVB 2) standards, let alone the *quasi-cyclic LDPC* (QC-LDPC) of the IEEE standards[33–35]. As a consequence it would be hard to make a clear distinction between each type of LDPC code based on a single feature alone, due to its large diversity.

Tanner Graph Indexing Indexing of the Tanner graph adjacency between nodes is found on the literature to follow one of three distinct approaches. Under ASIC implementations, hard-wired memory locations can be utilized, though they lead to non-scalable routing complexity, driving up the chip manufacturing costs the longer the LDPC code[231,257]. Found under both ASIC and reconfigurable computing, barrel shifter approaches that rotate the memory positions of data elements in fixed-line position can be found[103,185,258], mostly for a kind of LDPC-IRA codes. Finally, QC-LDPC codes see an approach similar to the former, tuned to the particular case of permutation sub-matrices[100,259].

Tanner Graph Regularity Irregularity within the Tanner graph is unavoidable due to the underlying code capacity which closes in on the Shannon limit for shorter lengths than regular codes. However, for simplicity of encoding, standardized LDPC codes are systematic with *parity nodes* (PNs) interconnected by *repeat-accumulate* (RA) structures. Furthermore, column or row swapping of the LDPC codes can be performed without changing the code attainable *bit error rate* (BER) performance, and, thus, regularization of its *check node* (CN) degree profile and *variable node* (VN) weight profile can be performed. With regards to non-binary LDPC codes, the field dimension, usually *binary extension field* (GF(2^m)), performs a code expansion similar to what is performed by the expansion factor z_f in QC-LDPC codes, i.e., a factor of regulariry of 2^m is present throughout the code structure. Moreover, the complexity of non-binary LDPC codes is captured at the arithmetic-level and not at the efficiency of memory access, and, thus, it takes a minor role under non-binary decoding.

Code Length Although code length has been proved not to be the main driving force to the Shannon limit[260], across the classes of LDPC deployed nowadays, their capacity is still mainly driven by length. Furthermore, length also drives the number of operations required to be performed in a reasonable amount of time, irregardless of the number of bits exchanged[61], since the code length N and number of parity-check restrictions M drives up the arithmetic complexity, along with d_v and d_c (c.f. Table 2.1).

We capture the aforementioned diversity onto the LDPC codes summarized in Table 4.3. Regarding structured Tanner graph indexing, scenarios I, II and III allow on-the-fly computation of indexes of data elements[6,7,12], while IV and V require sparse matrix indexing methods[11,12]. Scenarios posed by II–IV represent short to moderate length LDPC codes, while I is representative of long length codes. Finally, these two features are mixed with the field where the code is represented with I–IV over GF(2) and V over GF(2^m).

5.3 Dataflow LDPC Decoder

In this Section, we consider the development of LDPC decoders under a dataflow-driven approach based on DFEs[178,261]. The underlying programming model utilized to pursue this type of accelerator architecture is based on the Maxeler's MaxCompiler[261], which targets the FPGA as an accelerator using a stream-based approach, hiding details such as memory controllers for communication of the host computer system with the FPGA, the interconnection of the different kernels that compose the accelerator, and other controllers required for the FSM implementation. Instead, the designer focuses on the dataflow expression of the algorithm as a kernel written in Java extended with dataflow extensions supported by the MaxCompiler infrastructure (illustrated in Figure 5.8).

The dataflow Java extensions allow the implementation of the Maxeler system comprising kernels and a manager. A kernel is defined as a hardware datapath performing

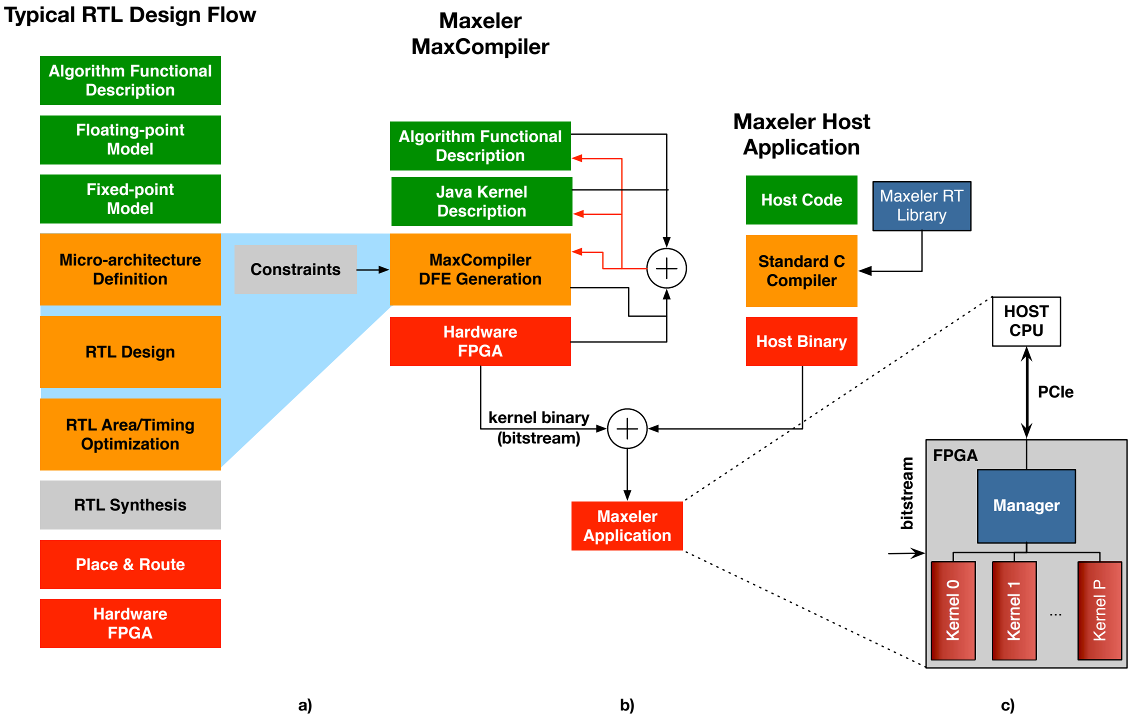


Figure 5.8: MaxCompiler design flow for hardware development: *a)* hardware design flow where all micro-architecture RTL-related tasks are performed by the MaxCompiler DFE generation, *b)* host program development flow linking the Maxeler RT library to a program using a standard C compiler, and *c)* system level block diagram of the host, manager and kernel DFEs.

the arithmetic and logical computations that are the mapping result of the algorithm description. The data required for the kernels to operate is fed by the manager through off-chip I/O in the form of streams, with the manager also orchestrating the kernel calls for execution. The compiler exploits a streaming model for off-chip I/O to the PCIe, to implemented DFEs via MaxRing[261], and to *dynamic RAM* (DRAM) memory. The objective is to keep the utilization of the available bandwidth of off-chip communication channels high, without the need to dig deeper onto low level FSMs that control the way data is flowing. Under this philosophy, by keeping communication and computation separate through the use of a manager and kernels, the latter can be as deeply pipelined without encountering synchronization issues. Under this streaming model both communication and computation occur concurrently.

In order to provide a control by a host application, Maxeler provides, in addition to the Java extensions with which the kernels are described, a C API for integration within a C program, much alike the CUDA and the OpenCL programming models. The Java extensions provide an in-depth customization of the kernels up to the bit manipulation level, a feature usually omitted by other HLS tools that are C-based, for instance[177,201]. Furthermore, it allows the definition of basic accelerator blocks that can then be instantiated at the Java kernel description level as arrays of dataflow *functional unit*s (FUs) for improved parallelism exploitation. This way, a fine-grained parallel expression is possible, keeping the control of the actual parallelism level to the hardware designer. Furthermore, the Maxeler RT API provides all functionality and interfacing required for handling the FPGA accelerator composed of DFE.

5.3.1 M-modulo dataflow LDPC decoder

The working system of the dataflow decoder is discussed herein. We designated the decoder system as M-modulo dataflow decoder due to its modular architecture composed of a number M of FUs as detailed in the discussion that follows. Three separate units compose the streaming datapath as seen in Figure 5.9, a front-end, a processing block, and finally a back-end. The front-end loads all data from the input stream and redistributes it to the BRAM units inside the FPGA, the processing stage performs LDPC decoding over the incoming streams, and, finally, the back-end outputs the decoded data back in the output stream. Double-buffering is performed so that the developed pipeline is fully utilized at all times, with computation and data communication occurring simultaneously to keep the processing stage actively decoding data[207].

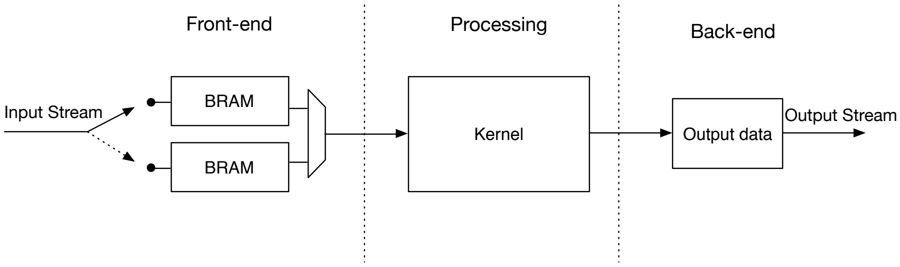


Figure 5.9: Basic dataflow accelerator topology implementing a double-buffering strategy at the input. This allows for a maximum three streams of data to reside on the system 1) a data stream flowing from the host to the accelerator, 2) another one being processed in the accelerator kernels and 3) a stream of output data leaving the accelerator towards the host.

Due to the precise level of control allowed by the dataflow approach, we are able to design LDPC decoders based on LDPC codes developed with certain hardware traits in mind, retaining these traits for improved decoding performance[61,258]. In particular, LDPC-IRA codes such as those developed for the DVB 2 standards allow highly customizable decoding architectures based on the factorization of the construction factors that provide regularity to the LDPC Tanner graph construction. In particular, this class of codes allows for partially parallel accelerators, designated as M-modulo accelerators, with M is a sub-multiple of the regularity factor r_f with which the code protograph is expanded[61,232,233]. This architecture provides a tradeoff between logic utilization and throughput by adjusting the number of FUs devoted to the decoding of nodes, as illustrated in Figure 5.10. It is worth noting that this type of architecture can also be reworked for LDPC codes outside of the LDPC-IRA type of the ETSI DVB 2 standards. While the

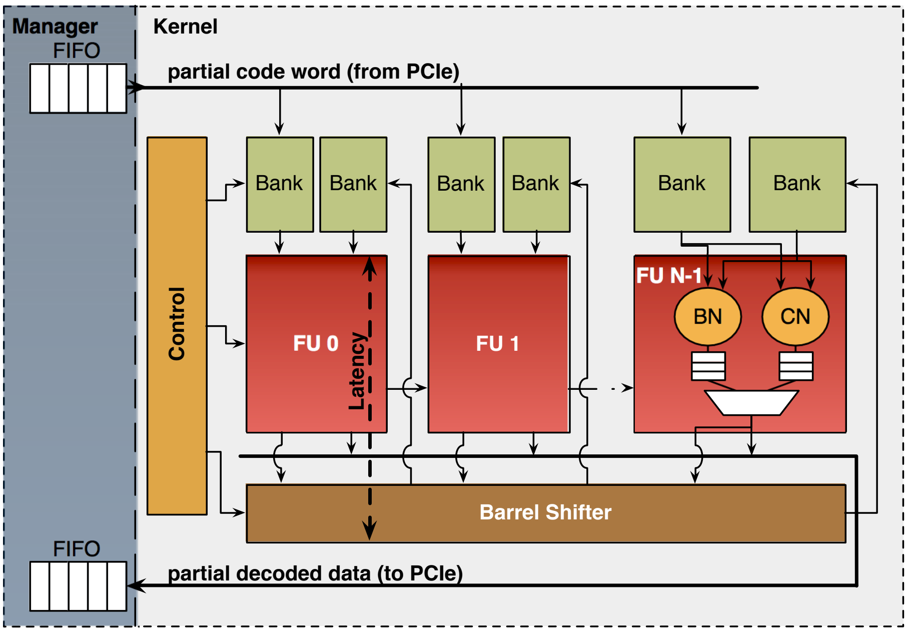


Figure 5.10: *M*-modulo dataflow architecture. Data is streamed from the manager onto the appropriate BRAM memory banks and is decoded in successive partitions with the granularity of the *M* FUs defined.

M-modulo architecture in ASIC technology explores another tradeoff regarding complexity of the memory system supporting each *M*- configuration, in FPGA this tradeoff is not evident, since the required memory blocks (BRAMs) already exist distributed across the FPGA logic resources. Furthermore, routing contention is not clearly anticipated since a higher number *M* of FUs forcibly utilizes more logic resources, thus spatially covers a broader area, which does not necessarily means poorer routing since computation is well divided among memory banks. On the other hand, lower *M* levels lead to spatially more contained accelerators, in their turn requiring less memory units, since less memory banks are required. At a first glance, computation logic to memory logic requirements are perceived to grow in a balanced way.

Moreover, if we consider that in hardware it is near-impossible achieving the parallelism levels of programmable architectures such as GPUs by assigning each thread of computation to a physical execution instance on the FPGA logic, the *M*-factorizable architecture serves yet another purpose. It can be adjusted to the amount of logic available in the target FPGA. Thus, a strategy equivalent to that pursued by *thread-per-node* (TpN) parallel expression becomes serialized in *M*-chunks of nodes composing the Tanner graph of the LDPC code. This way, taking into account the regularity inbuilt in most standardized codes the scheduling of the nodes in the Tanner graph for update does not become limited to the *Turbo-decoding message-passing* (TDMP) scheduling as it can also support TDMP decoding.

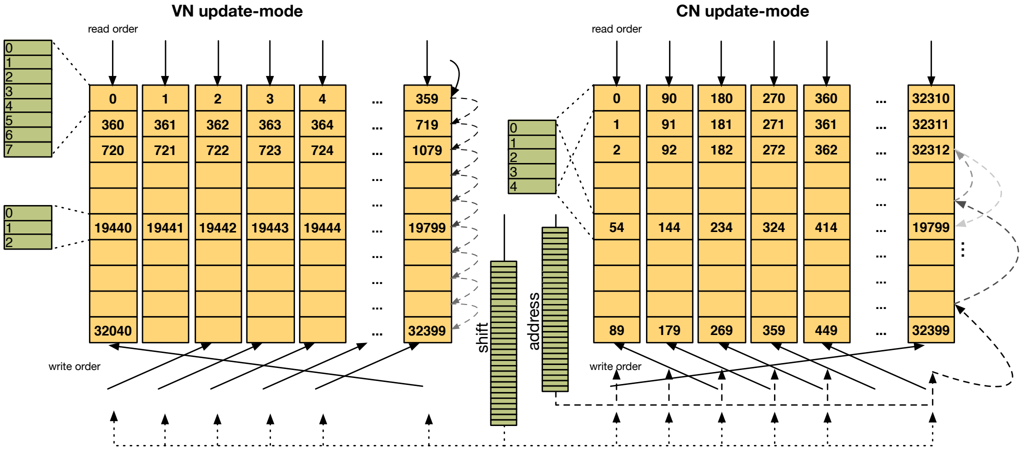


Figure 5.11: Memory layout for the M-modulo dataflow decoder. Address and shift LUTs index the Tanner graph corresponding to the INs. The access pattern is in-order line and bank-aligned reading, shifted-bank writing for the VN update mode, and address-indexed shifted-bank reading and reverse shift-bank writing for the CN update mode. The enumerated indexes correspond to the rate N=64800 bits 1/2 DVB-S2 code.

Tanner Graph Indexing The Tanner graph of the LDPC-IRA codes in use with DVB 2 standards is constructed from expanding a set of independent binary columns into the final dimensions of the LDPC code[61], as discussed in Chapter 4. However, in this case, using the dataflow approach, we have the flexibility to configure the memory on which the messages are retained as a 2-dimension memory space. The number of elements required to map the Tanner graph is given by

$$N_{elements} = 2 \times \left(f_j \times K_j + f_3 \times K_3\right), \; K_j + K_3 = K, \tag{5.1}$$

with f_j= max (d_v), f_3=3 and K_j and K_3 the number of VNs with weights f_j and f_3 respectively.

The memory layout of the *log-likelihood ratio* (LLR) messages is illustrated in Figure 5.11. It divides VNs messages in row-major storage and CNs in column-major storage[103,233]. When data is required for access by the VNs, all banks in a line are accessed simultaneously, and, thus, the VNs that will be updated are consecutively distributed. On the other hand, CNs are updated in non-consecutive fashion. The memory number of banks can grow to a maximum width depending on the LDPC code. Also, due to being able to produce and consume data elements in-place[232], the memory size required has $M^{'}$ banks and a number of lines that defines the total size to be $N_{elements}/2$. Since the number of banks relates to the number of FUs in the design, and the number of stored LLRs remains the same, for a given $M^{'}$ configuration, the number of memory lines is given as

$$N_{lines} = \frac{N_{elements}}{2} \times \frac{M}{M'} = \frac{M}{M'}\left(f_j \times K_j + f_3 \times K_3\right), \tag{5.2}$$

and, thus, increases with the inverse of the number of FUs in the accelerator.

5.3.2 Pipelined FU Execution

Inside the FU expressing the finest-granularity within the dataflow decoder, i.e., at the node level, computation can be performed in one of two ways. Data can be streamed from the corresponding memory locations onto the FU and then computation applied to all data elements in parallel, or in the presence of data dependencies, executed in parallel for the majority of the arithmetic operations involved. However, this approach comes with a major limitation. By requiring data elements to be made available for computation at once, this adds further pressure to the utilization of BRAM units which have a high, albeit limited, available bandwidth. To prevent this, arithmetic units inside the FUs would starve for data until all data requests would be served. Naturally, this approach is ill-suited to maximize the available bandwidth, both delivered by the BRAM units and by the arithmetic units. Furthermore, parallel trees of computation introduce a logic utilization overhead that can be prevented.

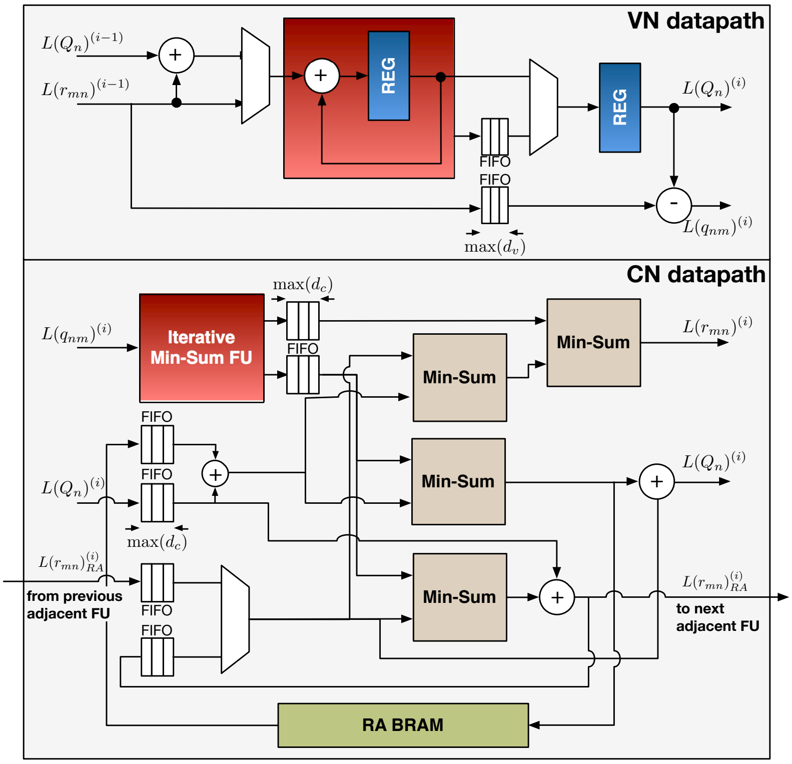


Figure 5.12: Dataflow pipelined MSA FU VN and CN datapaths. Data is sequentially fed into the VN or the CN datapaths which provide fully pipelined execution up to the maximum pace of 1 LLR entering the pipeline per clock cycle.

On the other hand, definition of a pure sequential approach, allows data to flow through the datapath in FU implementing the decoding algorithm without incurring into additional BRAM utilization for keeping the hardware in the FU fully active. Naturally, the decoding architecture would not be able to withstand the latency of the FU for each message flowing through it, and, thus, the FU is fully pipelined so that it can accommodate a message per stage of the FU datapath. This datapath definition is illustrated in Figure 5.12 for the particular case of the *min-sum algorithm* (MSA), although any other algorithm that is message-passing and LLR-based can be supported with no modification of the architecture except for the primitive arithmetic instruction issued by each block in the system level diagram[262]. In the illustrated MSA case, the `Iterative Min-Sum FU` computes the CN update rule (2.30) and sequentially outputs after i cycles the update corresponding to the i-th fed LLR message. In addition it also outputs the absolute minimum that is required to process data corresponding to the RA code sub-matrix, i.e., the PNs. The `Min-Sum` system blocks compute the following expression

$$\texttt{Min-Sum}(a,b) = \text{sign}(a) \times \text{sign}(a) \times min(|a|,|b|). \tag{5.3}$$

Fully Pipelined Execution The pipelined FU supports a dual-mode of operation. Both the VN and the CN datapath are defined within the FU, as opposed to a separate FU for VN and for CN datapaths. This helps to contain the logic overhead to a minimum. Not requiring a control unit for two separate FUs, allows sharing the logic required to implement the data-flow through the correct pipeline. As observed in Figure 5.12, a certain number of FIFO elements needs to be inserted into the correct locations for a two-fold purpose. To allow the pipelining of LLRs in the datapaths up to a maximum rate of one LLR per clock cycle, and as computation occurs fully pipelined, the FIFOs can hold simultaneously data from two different VNs or CNs. Correct execution in this case depends on the resetting of the registers inside the VN and CN datapaths.

Considering that the pipeline feature of the FU introduces an additional latency to the execution of the LDPC in both CN and VN operation mode, data produced by the trailing executing nodes must be streamed to their correct locations prior to the commence of the next decoding phase, i.e., prior to the nodes that will consume this data specifically execute. This approach ensures coherent streaming of data in the consumer-producer relationship between nodes in the Tanner graph. Thereby, the introduced coherence-proof design guard allows for the safe commuting of the FU mode of operation, from CN to VN. Looking into the particular data streams' consumption and production patterns, computation of nodes can be rescheduled in such way that it still matches the M-factorizable scheduling of nodes in M-chunks, only reordered to avoid memory hazards.

The systematic approach to the rescheduling of the nodes execution order involves matching the order access of the memory bank lines by shifting the scheduled order taking into account the latency of the VN and the CN datapath. *A-priori* analysis of the LDPC codes involved in the DVB-S2 standards, for instance, shows that all normal frame codes, under sustained latencies of up to 18 clock cycles can be reordered as tabulated in

Table 5.1. The greater the M, the more pressure is put on the correct consumption of produced data streams, although analysis shows that for a factorization level of M=360, only the rate 1/2 code requires reordering of the VNs starting execution. Lower factorizations of the decoder architecture do not entail a reordering of the nodes' execution schedule. Also, this principle would also hold for other standardized codes such as QC-LDPC decoders, even though there are slight differences to how mapping of messages in memory is performed[135].

Table 5.1: Dataflow decoder rescheduling of nodes' order of execution. The critical part is when CN and VN computation overlaps for the duration of the datapath latency. In the example provided—DVB-S2 rate 1/2 normal frame LDPC code for M=360 FUs—there would be a stream of data consumed incoherently since it had not been produced by the corresponding CNs (highlighted in red). By performing an appropriate rotation, coherent consumption and production of streams is achieved for this case, and also for the remaining codes and M-configurations.

In-order execution of DVB-S2 rate 1/2 code (M=360, CN/VN latency 18 clock cycles)																			
Memory lines simultaneous accesses	Trailing CNs	249	256	258	91	207	264	307	434	42	61	123	272	349	🟥 4	153	262	280	311
	Starting VNs	0	1	2	3	🟥 4	5	6	7	8	9	10	11	12	13	14	15	16	17
	Trailing VNs	432	433	434	435	436	437	438	439	440	441	442	443	444	445	446	447	448	449
	Starting CNs	212	218	246	288	403	82	87	167	291	304	36	129	221	294	335	18	52	133
	Out-of-order execution rotated by 16 positions on the VNs																		
	Trailing CNs	249	256	258	91	207	264	307	434	42	61	123	272	349	4	153	262	280	311
	Starting VNs	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
	Trailing VNs	448	449	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	Starting CNs	212	218	246	288	403	82	87	167	291	304	36	129	221	294	335	18	52	133

Finally, as soon as the final iteration data is being streamed back onto the host computer, the double buffering mechanism is already streaming data to the memory lines no longer in use in the decoder architecture and issuing the execution of the nodes in the Tanner graph thereof corresponding.

Data-parallelism vs. Task-parallelism The ability to instantiate an accelerator composed of several fine-grained FUs allows us to explore what the optimal tradeoff between data- and task-parallelism might be. Under the M-factorizable decoding architecture, it can be summarized in the following way. Taking a fixed number of $M^{'}$ FUs how can they be divided onto K M-modulo LDPC decoders ($K \times M = M^{'}$) on the FPGA die such that the decoding throughput is maximized and the decoding latency is kept at bay. Due to the decoding architecture nature, a reduction of the number of FUs by a factor K_r leads to an increase of the latency by the same factor. Thus, the decoding solutions with higher number of FUs devoted to a single codeword are expected to achieve minimum latency. However, there is a conceptual equivalence in terms of data throughput in having a 2M-modulo decoder or two M-modulo decoders in the FPGA accelerator.

In the latter, the tradeoff nature is due to how the compiler generates the circuits and how well can the placement and routing tool perform its task. Considering that data-parallelism is driven upwards, further pressure is put onto the routing of data to the correct BRAM locations, which increases in the proportion of the number of decoders by which the $M^{'}$ FUs have been divided. In other words, while there is not pressure to consuming more logic due to more FUs, memory resources are increasingly procured the more decoders are included for a fixed number of FUs. As a consequence, we can anticipate that the quality of the routing solutions for the more decoders case will degrade, as opposed to a lower number of decoders.

5.3.3 Experimental results

The methodology discussed for the M-modulo dataflow decoder discussed in Section 5.3 has been benchmarked for dataset I in Table 4.3. The hardware has been generated for the MAX2336B and the MAX3412A boards, respectively architectures F2 and F3 in Table C.1, using the MaxCompiler, in its turn based on the 2012.2 Xilinx Suite[178]. The decoders incorporate the Maxeler *Simple Live CPU* (SLiC) interface using its real-time API to allow communication between the host computer system and the FPGA board through a 2nd generation x8 PCIe interface (4 GB/s peak bandwidth). Certain decoders, using the MAX2336B board, have been executed for profiling the power drawn by the board using the Power Sensor described in Table C.1. This sensor apparatus estimates power based on the measure of electrical current, through the Hall effect, circulating in the PCIe buses, it allows a maximum range of ∼250W with quantization steps of 7W (±3.5W precision).

Next follows the discussion on the performance attained with the M-modulo binary decoder methodology under the dataflow approach. In particular, we analyze the decoder system ability to capitalize on the instantiation of a great number of FUs that can then be broken down into more or less sub-decoder systems. This capability is due to the overall logic resources available in both the F2 and F3 architectures, with the former providing the fewer resources than the latter (c.f. Table C.1).

Logic Utilization The M-modulo decoder that is based on designs optimized for LDPC-IRA codes[185,232] allows for factorization levels that are related to the regularity factor with which the LDPC code is constructed. In the dataset I case, the normal frame DVB-S2 codes are expanded by a factor of 360 allowing the factorization of the code structure by any of its multiples. In particular its prime factorization yields $360{=}2^3{\cdot}3^2{\cdot}5$, thus making it straightforward, with regards to indexing of the Tanner graph, to instantiate a number of FUs that is a sub-multiple of 360.

$$\text{Available No. of FUs} = 2^i \times 2^j \times 2^k,\ 0 \le i \le 3,\ 0 \le j \le 2,\ 0 \le k \le 1 \tag{5.4}$$

We explore the sub-multiples factorization that reduces the 360 value by powers of 2, i.e., that reduce the number of available FUs of each decoder (5.4) to an admissible range given by $\{45, 90, 180, 360\}$, since it is amenable to just require one more, or less, bit in the control logic of the LDPC hardware due to the power of 2 difference factor. Furthermore, considering that the properties of the Tanner graph cannot be exploited to the fullest beyond the regularity factor with which the code is constructed, whenever the logic resources permitted the instantiation of more than 360 FUs, such level defined the maximum number of FUs assigned to each sub-decoder system.

The synthesis results for FPGAs F2 and F3 are shown in Tables 5.2 and 5.3, respectively. As aforementioned, due the ample logic resources provided by each FPGA device, we are able to elevate the number of decoder sub-systems p in the LDPC decoder accelerator, with each sub-system responsible for decoding one codeword. Thus, we are able to elevate the data-parallelism levels of the LDPC decoder without the need to redefine computation within the *arithmetic and logic unit*s (ALUs), which unlike the GPU approaches discussed in the previous chapter, or the wide-pipeline approaches in the following subsections, elevates the data-parallelism at the cost of vectorized instructions applied to words with wider bitwidths packing several data elements. In Tables 5.2 and 5.3, (p, m) stand for the number of decoders instantiated in the accelerator and m for the number of FUs instantiated per decoder, with $p\times m$ the total number of FUs instantiated in the overall LDPC decoder design.

Table 5.2: Dataflow LDPC decoder hardware characteristics (MAX2336B FPGA): FPGA logic resource utilization, clock frequency and throughput at 10 decoding iterations for the benchmarked LDPC decoders. In addition, the number of FUs and its configuration in decoders and FUs per decoder is read at the top rows.

			MAX2336B (F2)						
	No. of FUs $p\times m$		45	90		180			360
	Config. $(p,m)^*$		(1,45)	(2,45)	(1,90)	(4,45)	(2,90)	(1,180)	(1,360)
Resource Util. (%)	LUTs		12	19	19	33	33	33	62
	FFs		16	25	25	43	44	44	82
	BRAMs	Total	27	41	29	71	47	35	49
		Kernel	13	14	14	18	18	18	24
Perf.	Clock (MHz)		300	250	260	180	150	200	175
	Thr. (Mbit/s)		270	450	468	648	540	720	1260
	Min. Dec. Iterations		0.76	1.27	1.32	1.82	1.52	2.03	3.54

Under the MAX2336B FPGA, the LDPC decoder is limited to 360 FUs that push the utilization of FFs to over 80%. Due to this, alternative configurations (p, m) with $p\times m{=}360$ were impossible to implement by the tools. As a consequence, we were limited to test all admissible configurations for up to 180 FUs. As observed in the throughput obtained, the highest throughput is, naturally, obtained for $(1, 360)$, but equivalent designs show that the highest throughput is always achieved for the lowest number of decoder sub-systems in the design. Thus, the highest throughput $(1, 90)$ and $(1, 180)$ are the decoder systems which attained the best fitting performance, as expressed by the clock frequency of operation of the designs. There is a clear tendency for the increasing complexity brought upon by defining more FUs in the design lowering the clock frequency of operation, in particular, for levels of p other than 1 an additional penalty is added due to the overhead in logic control.

Furthermore, statically splitting the instantiated FUs through a different number of decoders sub-systems comes with a variable utilization of BRAMs. For each increment in p a complete set of memory banks is instantiated along with the corresponding logic that emulates in the supported JAVA language the barrel shifter behavior[10,232]. Thus, the utilization of BRAMs is the highest for $(4, 45)$ which supports up to 4 distinct memory systems for the indexing of the LLR messages of 4 codewords. Notwithstanding the memory footprint doubling from the $(2, 90)$ to the $(4, 45)$ design, the BRAM utilization level does not indicate it, since the tool is able to re-utilize any space available of BRAM units whose space has been partially allocated. Furthermore, considering that we have defined a rate of memory accesses that see a line read and a line write in the same cycle, but the reading and writing of data never fall on the same memory line, the compiler was able to reschedule the access to the BRAM so that processing is never stalled due to insufficient memory ports.

Table 5.3: Dataflow LDPC decoder hardware characteristics (MAX3412A FPGA): FPGA logic resource utilization, clock frequency and throughput at 10 decoding iterations for the benchmarked LDPC decoders. In addition, the number of FUs and its configuration in decoders and FUs per decoder is read at the top rows.

			MAX3412A (F3)					
	No. of FUs		45	90		180		
	Config. $(p, m)^*$		(1,45)	(2,45)	(1,90)	(4,45)	(2,90)	(1,180)
Resource Util. (%)	LUTs		6	9	9	16	16	17
	FFs		4	7	7	14	14	14
	BRAMs	Total	6	10	6	19	12	8
		Kernel	2	2	2	3	3	3
Perf.	Clock (MHz)		260	250	260	200	150	200
	Thr. (Mbit/s)		234	450	468	720	540	720
	Min. Dec. Iterations		0.66	1.27	1.32	2.03	1.52	2.03
	No. of FUs $p \times m$		360				720	
	Config. $(p, m)^*$		(8,45)	(4,90)	(2,180)	(1,360)	(8,90)	(2,360)
Resource Util. (%)	LUTs		32	31	32	32	63	64
	FFs		26	27	27	27	52	53
	BRAMs	Total	37	22	15	13	43	24
		Kernel	5	5	5	5	10	10
Perf.	Clock (MHz)		150	150	170	175	90	80
	Thr. (Mbit/s)		1080	1080	1224	1260	1296	1152
	Min. Dec. Iterations		3.04	3.04	3.44	3.54	3.65	3.24

The synthesis results obtained for the MAX3412A board are written in Table 5.3. Due to the FPGA chip in the board be of a newer family than that of the MAX2336B board, it provides more logic resources and the CLB within provides double the number of FFs, the most pressed logic resource after BRAMs in the MAX2336B designs. In this case, we are able to drive the number of FUs to 720, although we fall in the same situation as before. On the one hand, we were able to synthesize the $(2, 360)$ and the $(8, 90)$ designs, but on the other, the remaining admissible configurations could not complete even though the logic utilization of all elements is kept at under 70%. Nevertheless, we must note that the new generation of FPGA enables the design of increased parallelism, but does not elevate the decoding throughput beyond the levels already witnessed for the MAX2336B board. In part, this is due to the speed grade of the FPGA chip that quantifies how fast can the logic elements in the chip commute. In other words, while more resources exist, the clock frequency of operation will not scale.

As observed in the logic utilization levels, the overhead brought on by higher levels of p is mostly observed at the BRAM utilization ratios and at the critical path of the LDPC decoder system. The higher the p, the lower the clock frequency of operation, rendering the increase in data-parallelism mostly an unwanted operation. Exception must be made on the $(8, 90)$ design that instantiated more decoder sub-systems than the equivalent $(2, 360)$ design. However, in this case, the basis for comparison should be the $(1, 720)$ which is impossible to define under the assumed dataset I, as the regularity factor of the Tanner graph is set at 360[61].

Considering the tradeoff between data-parallelism levels, driven through the number of decoder sub-systems p, and attained decoding throughputs, weighed by the clock frequency of operation, it is clear that setting p=1 is the overall best strategy, in spite of the decoding latency not having been considered yet. Since latency per decoder word increases by a factor p for a fixed $p \times m$, it becomes clear that the practical use of defining more sub-decoder systems within the LDPC decoder accelerator is limited.

Roofline Analysis Considering that both the MAX2336B and the MAX3412A boards have limited bandwidth provided by the external PCIe interface and by their DRAM systems, we need to analyze under what conditions is the bandwidth enough to feed data to the LDPC decoder accelerator without having the processing system starve for data that is moving from the host system to accelerator, or that is being flushed from the accelerator to the host system. The case of the LDPC decoding algorithm is particularly interesting. Due to its iterative nature, so long as a sufficiently high number of decoding iterations is executed, computation can mask the movement of data, in particular, if data is flowing in streams in a pipelined fashion. This way, up to three streams of data can occupy the accelerator system, since 1) a stream of data moving from the host to the accelerator, 2) a stream of data being computed in the accelerator, and 3) another one being moved from the system to the host can reside simultaneously on the system. This has been the considered scenario, as illustrated in Figure 5.9, in anticipation of the limited bandwidth provided the board interfaces.

In particular, we need not consider the movement of data through the DRAM banks, since data is flowing directly from the host computer system via the PCIe interface and is being written directly on the BRAM memory space. Thus, we need only to consider the 4 GB/s peak memory bandwidth provided by the x8 2$^{\text{nd}}$ Generation PCIe interface. Moreover, the analysis pertains to the LDPC decoding domain in the sense that we use a figures of merit for the bandwidth that make sense under such domain. Thus, the roofline analysis illustrated in Figure 5.13, expresses the decoder bandwidth as decoding iterations executed per second and its relation to the number of decoding iterations issued per codeword. The former can be interpreted as a proxy to the *signal-to-noise ratio* (SNR) conditions on which the decoding system is operating. The worse the conditions, the lower the SNR and the higher the number of executed iterations before a new data stream is decoded, with the opposite relations also holding.

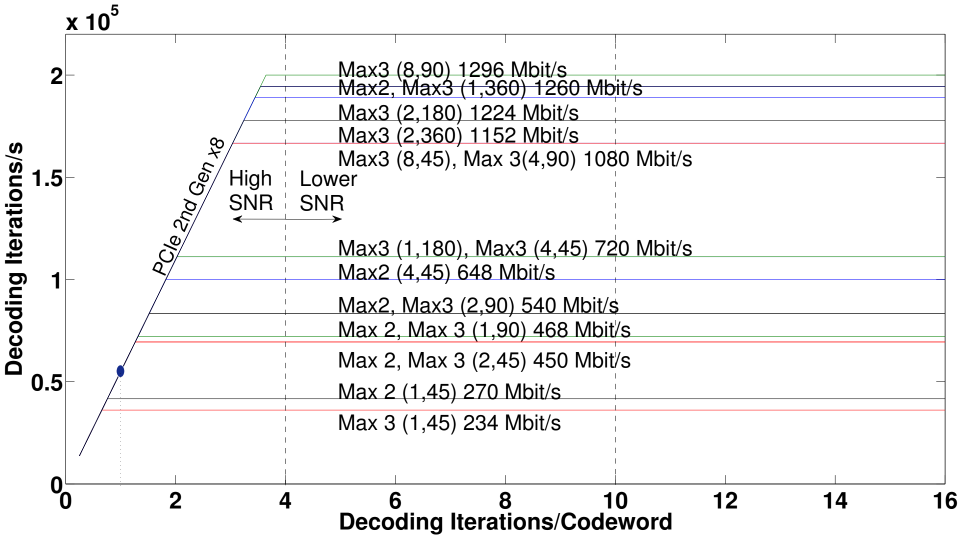


Figure 5.13: Roofline analysis of the dataflow M-modulo LDPC decoder.

In Figure 5.13, the slope curve represents the number of decoding iterations that can be executed per second for the LDPC code in dataset I bounded by the memory interface of the FPGA board, in this case the PCIe bus. After a certain level of computation executed, the accelerator is not longer bounded by memory, but rather limited by computation. This turning point occurs when the curve bottoms up and saturates onto a null slope which is the peak throughput of the decoder. Under our analysis, the turning point, where computation meets memory, corresponds to an SNR operation point that triggers a certain number of decoding iterations per codeword received for decoding. Naturally, the lower the throughput the less constrained by the interface limited bandwidth will the decoder be. On the other hand, very high throughputs, such as those sustained for the 360 and 720 FUs instantiated that reach the Gbit/s range are limited to not be able to deliver their peak decoding throughputs if the number of issued decoding iterations is lower than ∼4[6,10].

Energy Efficiency The LDPC decoder designs have been benchmarked using the Power Sensor (c.f. Table C.1) for evaluation of their power efficiency with regards to equivalent programmable decoders in order to provide a real insight into how much energy is saved from pursuing reconfigurable computing designs.

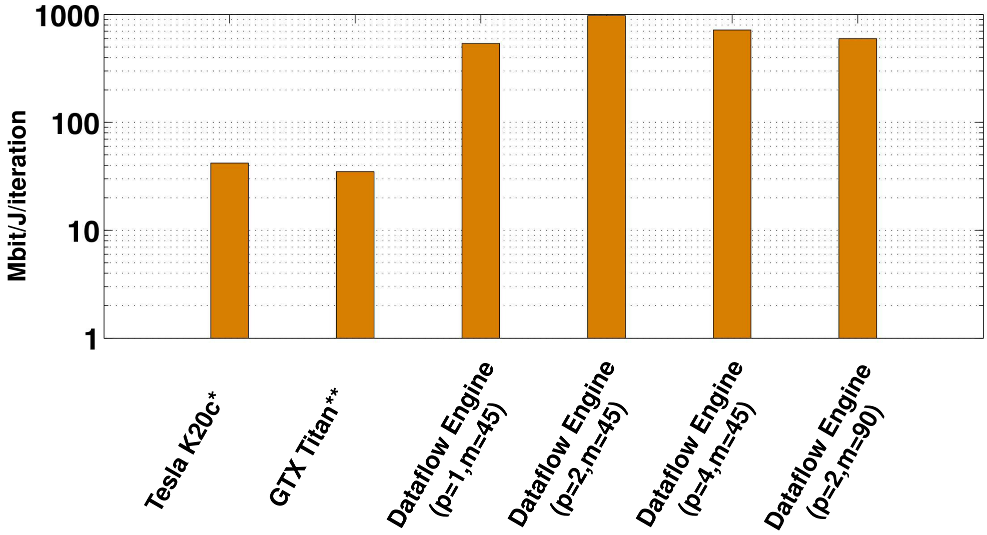


Figure 5.14: Energy efficiency of the dataflow M-modulo LDPC decoder measured in Mbit/J/iteration compared to two programmable designs: * from[10] and ** from[156].

As observed from the designs profiled, the energy efficiency obtained ranks close to the ∼1000 Mbit/J/iteration while equivalent GPU decoders will draw ∼100 Mbit/J/iteration. Given that there is a twofold difference in decoding throughput between the dataflow decoders on the FPGA and the programmable decoders on the GPU engines, the effective power gap ranks close to fivefold[10,156].

Relation to RTL Approaches Some comments are due regarding the M-modulo LDPC decoder designed through RTL project, instead of a HLS approach[185,232]. In particular, if we consider the work of Gomes *et al.*[185], their decoder design achieved clock frequencies of operation lower than those achieved by the dataflow design. Two factors should be weighed in to help explain this difference. On the one hand, the Virtex II Pro FPGA chip utilized is from an earlier FPGA family. As a consequence, it provides a lower number of logic resources, making the instantiation of more than 360 FUs, or even the breaking down of the instantiated FUs into several decoder sub-systems, not possible. On other hand, the speed grade of that generation is much faster than the Virtex 5 and Virtex 6 families that the MAX boards provide. Thus, in principle, the fitting tool should be able to provide a higher clock frequency of operation. However, re-synthesis of the RTL LDPC decoder design shows that the clock frequency of operation obtained is not better when using a more recent version of the Xilinx tool infrastructure, nor when using either the same FPGA chip as the MAX2336B or the MAX3412A boards.

Reflecting on the fact that the critical path timing is met within a comfortable slack, there should be room to improve on the clock frequency of operation. However, the synthesizer and the fitter will not pursue this goal on its own. This requires the correct input from the hardware designer at the appropriate design flow stage (see Figure 5.7), which highlights that under the HLS appropriate approach, we are able to refrain from dwelling into deep level optimizations such as defining timing constraints for the hardware generation tools. In this particular case, the LDPC decoder design was able to see its constraints met while having been described at the HLS level and not at the RTL micro-architecture level, as originally intended.

5.4 Loop-annotated LDPC Decoder

In this section, we consider the development of LDPC decoders under a method designated as loop-annotated decoder. In essence, this HLS approach relies on annotations of the LDPC decoder source code, via C,/C++ directives or through SystemC descriptions, to drive the hardware generation. The underlying compiler tool utilized for this case is the Xilinx Vivado HLS[179]. The design flow of the Vivado HLS is illustrated in Figure 5.15. The hardware mapping performed generates circuits for a C function marked as the top-level, similar to an RTL-based flow. All the functions, logic and arithmetic, inside the top-level are mapped onto block primitives in hardware by the C-synthesis. At this stage the behavior of the C-synthesized functions can be analyzed for functional correctness at the clock cycle level prior to performing the circuits synthesis (c.f. C-Logic Analyzer in Figure 5.15).

The hardware designer is able to drive the circuits mapping of the C code due to annotations performed in the form of Tcl directives fed into the C-synthesis process or as inline directive annotations using the `#pragma` C language construct. Among the available directives, we enumerate the directives of interest to the generation of efficient LDPC decoders.

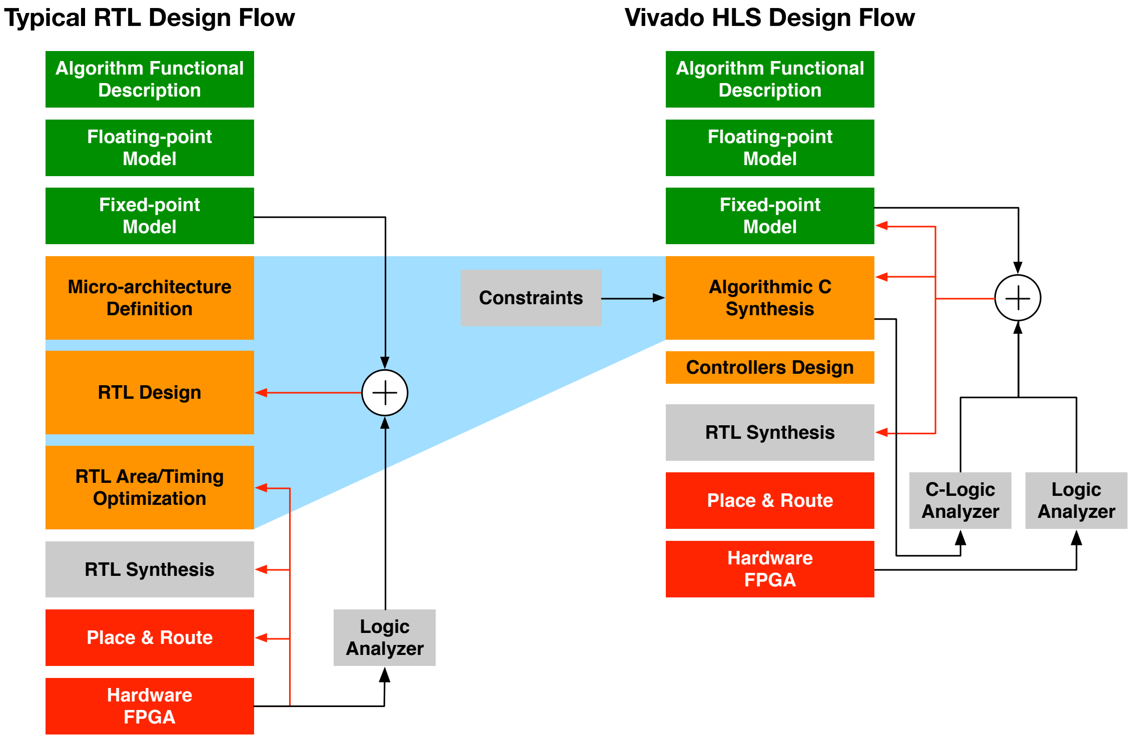


Figure 5.15: Vivado HLS design flow for hardware development. The majority of the RTL design is performed at the C synthesis level, although the integration of the HLS-built blocks in RTL is still required, namely for hooking pin locations for I/O and defining memory controllers.

i) Loop directives that affect the scheduling of iterations in the enclosing loop or structure of loops

a) unroll - allowing the parallel scheduling of iterations due to unrolling of instructions within the enclosing loop partially or completely;

b) pipeline - allowing the initiation of iterations aiming at certain target *initiation interval* (II);

c) flatten - controlling how loops in multiple nested levels interact are flattened to the containing loop structure;

d) merge - controlling if enclosed loops should be merged with their enclosing ones.

ii) Memory directives that influence how memory is mapped

a) partition - defining how elements are distributed across arrays in order to expose higher bandwidth of access;

b) reshape - similar to partition only creating wider word arrays in the process.

iii) Resource directives that drive the instantiation of specific hardware units in the C-synthesized code *a)* memory units such as BRAMs configured for single- or dual-port access, read-write or read-only access (ROM);

b) specific arithmetic units, such as multipliers, multiplexers or FIFOs;

c) port protocols such as AXI4[263] for I/O integration purposes and top-level port handling.

Template Architecture After the design cycle is complete at the C-synthesis level, the generated RTL accelerator is exported as an IP-XACT core for integration in a system-level architecture that instantiates as many cores as seen fit in addition to the required clock and memory interface controllers. To accelerate the design cycle, we take into consideration a template architecture that instantiates the required clock interfaces and memory controllers for DRAM access. To this end, we consider the template architecture illustrated in Figure 5.16.

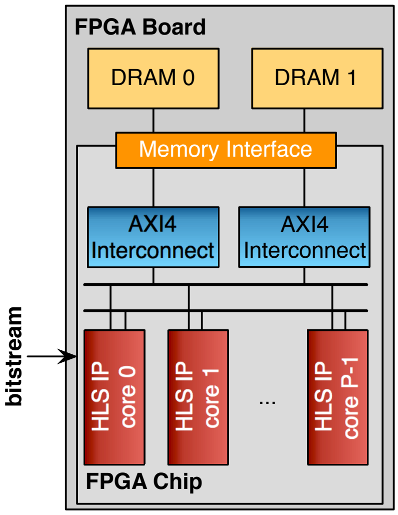


Figure 5.16: Template architecture for the loop-annotated LDPC decoder that hosts the HLS IP cores and instantiates the clock and memory controllers. The memory interface is a MIG[264] core managing the physical connection to the two DRAM banks which are split into two AXI4[263] interconnection cores.

The architecture hosts the IP cores that are exported after C-synthesis up to a limit of P=16 HLS IP cores, due to limitations on the supported number of ports for each AXI4 interconnect. Furthermore, the rationale behind this topology is having data being streamed from the DRAM via a double-buffering scheme that divides output and input streams in two distinct physical DRAM banks. This way, data streamed for decoding is never racing for bandwidth to DRAM with decoded data being streamed back to the host. Furthermore, it is desirable that prior to computation, data streamed to the DRAM banks is moved to BRAMs utilized by the HLS IP cores so as to keep data being computed close to where the utilized logic performing computation lies. A drawback of this particular HLS tool is the inability to define the template architecture at the C-synthesis stage.

Instead an RTL-based project providing the template architecture without the LDPC decoding kernels instantiated as HLS IP cores is recycled through several decoder solutions. This project is obtained with the Vivado *memory interface generator* (MIG) and the clocking wizards and conforms to the correct configuration of the Virtex 709 FPGA board specific pin locations and clock generators. It can be fully described in Verilog, VHDL or generated on-the-fly via the appropriate Tcl description of the template[179] to either HDL.

5.4.1 LDPC decoder isomorphic mapping to hardware

Under the HLS model pursued in this section, each C function is mapped onto its specific hardware instance following a purely sequential approach. All operations performed to data are translated sequentially to a *dataflow graph* (DFG). Naturally, some optimizations are automatically applied by the C-synthesizer whenever parallelism is detected, but for the majority of times, no optimizations are really performed. This also applies to loops that perform the same computation pattern to data without dependencies between iterations. In other words, the tool guarantees functionally correct behavior of operation, but does not perform any additional effort of optimization unless instructed in that direction via the available annotations[179]. To this end, we define a functional division of the Tanner graph operations in order to better express computation for the underlying HLS tool. This isomorphic mapping of the computation is illustrated in Figure 5.17. The represented operations in Tanner graph can be configured for the majority of the LDPC decoding algorithms covered in Chapter 2 and is not limited to the binary or the non-binary case. In fact, the Tanner graph structure illustrated is for a non-binary

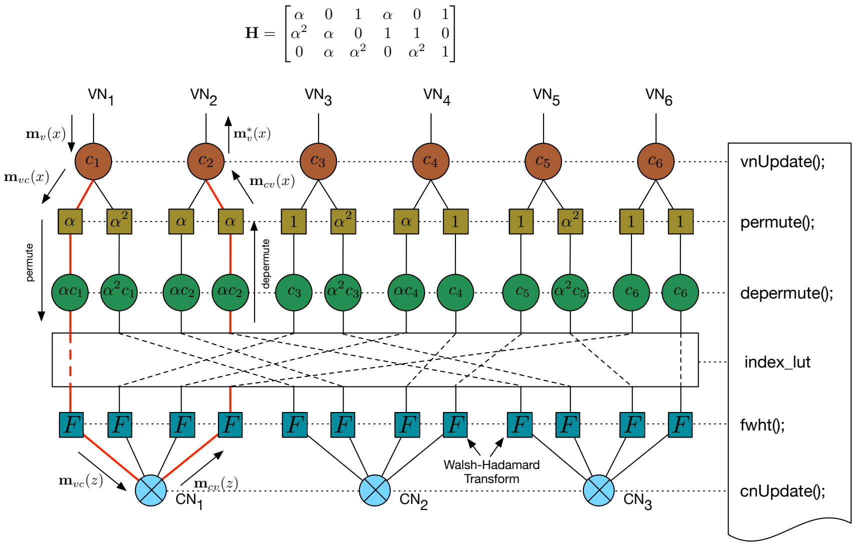


Figure 5.17: Vivado HLS isomorphic mapping of the message-passing message-passing algorithms. Each operation operation applied at the Tanner graph node-level or edge-level is mapped to a C-function definition shown on the right.

LDPC code, since it represents the most complex case. The binary case would see the trimming of all the operations defined at the edge-level. At this level, permutation and depermutation of the *pmfs*, and the *fast Walsh-Hadamard transform* (FWHT) applied are mapped onto the `permute`, `depermute` and `fwht` C-functions. At the node level,which is the only level of computation retained in the binary LDPC decoder case, the VN and the CN computations are mapped onto the `vnUpdate` and `cnUpdate`. This division makes it simple to expose parallelism at a fine-grained level, since cycling over the data that is consumed and that is produced is performed, under a C description, is made via loop structures that are not uniform across the different computation patterns applied. The memory locations of messages passing through the Tanner graph are indexed by appropriate LUTs, designated in the Figure 5.17 as `index_lut`.

The application of this design methodology to LDPC decoding of both binary and non-binary codes is simplified if nodes scheduled for execution follow the *two-phased message-passing* (TPMP) instead of the TDMP scheduling[265]. This way, we can split computation into three distinct dataflow regions 1) the `cnDataflow`, 2) the `vnDataflow` and 3) the `hdDataflow`, as seen in Figure 5.18. Each dataflow region requires that all data consumed by it is available at the time computation begins, but does not enforce this need within the dataflow region boundaries. Data can flow through each stage in the region without the need to synchronize or apply a memory fencing mechanism. On the other hand, fencing is strictly requited between regions. Between the execution of the CN associated kernels and those of VNs, and the other way round, messages are passed on the Tanner graph from one type of node to the other. However, the `hdDataflow` does not require fencing, as it can consume messages for hard-decoding as soon as they are made

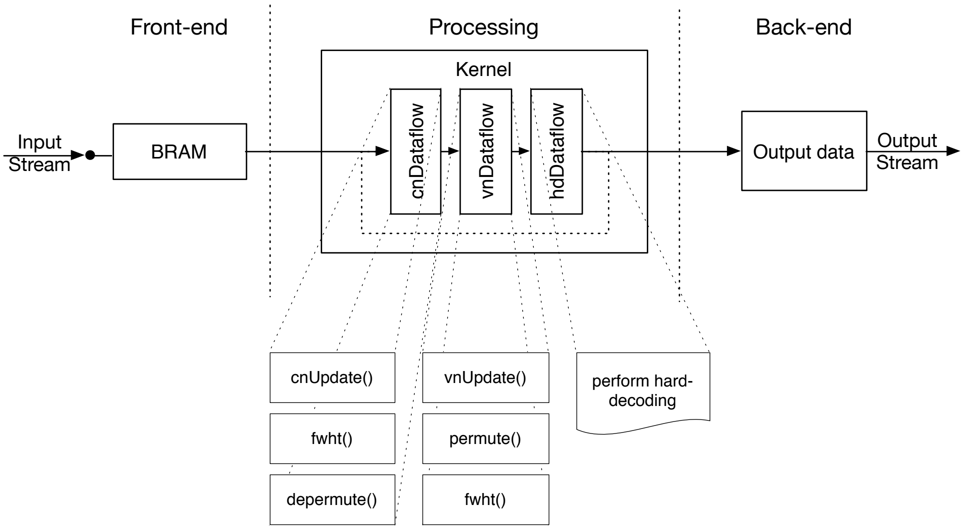


Figure 5.18: Vivado HLS dataflow regions within the LDPC decoder accelerator: `cnDataflow`, `vnDataflow` and `hdDataflow`.

available. Nonetheless, a fencing mechanism would still be required between the VN to CN dataflow regions, thus it is only a question of whether it is applied after or before the hard-decoding stage.

Similar to the dataflow decoder streaming model, data to be decoded is streamed on the FPGA accelerator and distributed to BRAMs that are spatially closer to where computation occurs in the chip at the front-end stage. The processing stage issues the computation defined in the kernel by the different dataflow regions. Finally, the back-end streams the decoded data back to the main memory.

5.4.2 Loop-acceleration

Due to the C programming language utilized for the description of hardware kernels, repetitive computation or computation that is applied to sets of data elements with certain patterns is made through loop structures. The general case of LDPC decoding is divided across multiple dimensions in the different node- and edge-level functions as follows.

i) Node-level (`vnUpdate` and `cnUpdate`)

   a) M CNs processed;

   b) N VNs processed;

   c) d_c messages processed by the CNs;

   d) $d_v + 1$ messages processed by the VNs;

   e) 2^m or $2^m - 1$ likelihoods handled in *pmf* or LLR format.

ii) Edge-level (`permute`, `depermute` and `fwht`)

   a) $M \times d_c$ edges processed in CN to VN traversing;

   b) $N \times d_v$ edges processed in VN to CN traversing;

   c) 2^m or $2^m - 1$ likelihoods handled in *pmf* or LLR format.

This leads to a general prototype of the functionality as seen in Figure 5.19. The trip count of each loop, and its relative position in the nested-loop structures, determines what optimizations can be performed to it, under the Vivado HLS model. Pipelining cannot be enforced on a loop without unrolling any inner loops therein contained. On the other hand, unrolling of an outer loop causes unrolling of any enclosed loops, except if an optimization directive exists for the pipelining of an enclosed loop. These constraints affect how the optimizations process can be driven by the loop directives. The inner loops in the `cnUpdate` or `vnUpdate` kernels have usually much lower trip counts than

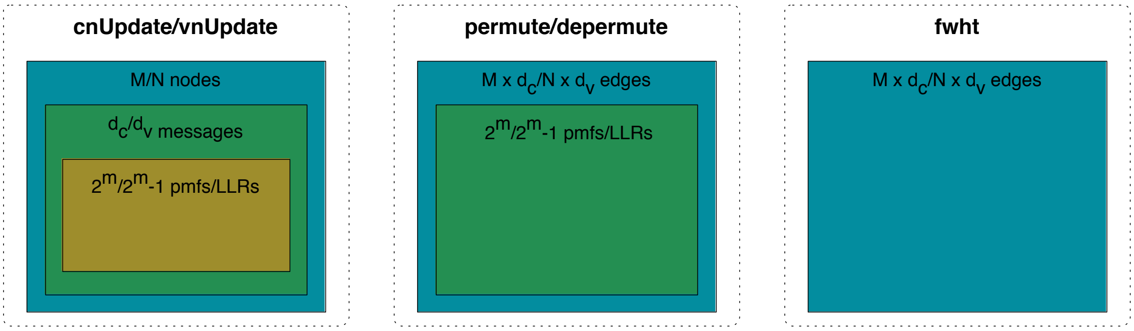


Figure 5.19: Vivado HLS loop nest structure of LDPC decoding kernels.

their enclosing loops. For the binary case using LLRs the innermost loops in the CN, VN and permutation kernels are effectively removed, while d_c and d_v have relatively low levels, leading to a manageable complexity level when optimizations such as pipelining or unrolling directives are requested for the outer loop iterating over the nodes or the edges in the Tanner graph. On the other hand, the greater the order of the GF(2^m) the greater the loop trip count in the innermost loop which in its turn puts pressure on the bandwidth purported by the BRAMs in the FPGA accelerator.

A complete loop unroll annotation in the code is adhered to by the C-synthesis, which effectively removes all loop control structure. However, the scheduling of all the operations in the loop by the different iterations will not necessarily be concurrent. Only if the number of available memory ports are able to service all the iterations simultaneously will a complete unroll map onto fully parallel execution. As a consequence, it does not suffice to move data from the DRAM space onto BRAM blocks in the FPGA chip. We need to address how data is stored across the BRAM banks so that enough memory ports are able to serve all the instructions fetching data across multiple iterations.

Furthermore, not only is unroll affected by the lack of enough BRAM ports. Pipelining of the outermost loops in the decoder design can only ensure the fastest II of one iteration per clock cycle if enough ports are available to provide data to each loop stage in a single clock cycle. Since the outermost loop is enclosing loops fetching, d_c/d_v messages and then 2^m or $2^m - 1$ data elements, there should be enough bandwidth to deliver at least $d_c \times 2^m$ and $d_v \times 2^m$, respectively from outermost to innermost loop.

5.4.3 Memory mapping

By streaming all data from the DRAM to BRAMs at the accelerator front-end, data is pushed onto memory units spatially closer to where computation occurs. However, BRAMs are physically two-port memories, which can be split onto two single-port half-size BRAMs, and the C-synthesizer default behavior is to store arrays on the minimum number of required BRAMs, with elements consecutively distributed. As illustrated in

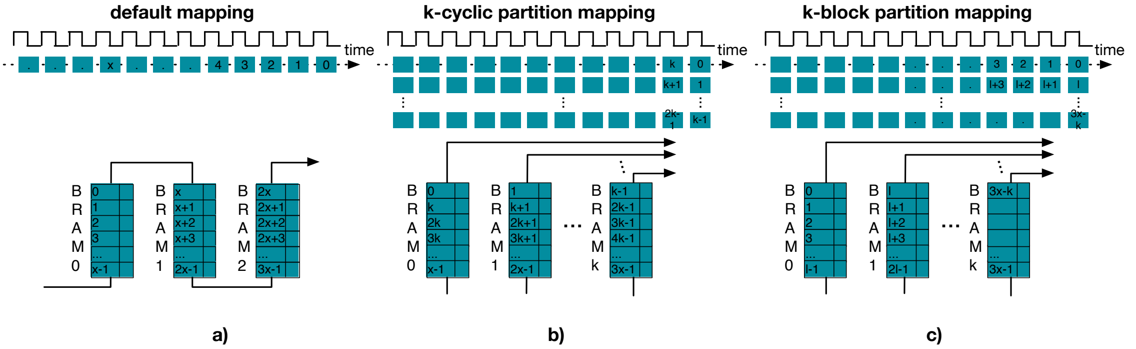


Figure 5.20: Vivado HLS BRAM-array partitioning: *a)* default behavior allocating a new BRAM whenever needed; *b)* doing so by cyclically spreading data across k BRAM units; and *c)* block partitioning across k BRAM units. In *b)* and *c)* the same number of BRAMs is allocated, only data elements are stored differently.

Figure 5.20, the default behavior reserves a port for writing and another for reading, making in-order data accesses not to be able to occur concurrently since the port cannot deliver more than a single element per clock cycle. The alternative, thus, is to divide data using the meachanisms defined by the appropriate directives and more BRAMs are utilized so that the extra ports can provide more bandwidth and allow for simultaneously accesses of data elements. This is shown in Figure 5.20*b–c)* for, respectively, cyclic and block partitioning by a factor k. For both cases, data is split across a number of BRAMs such that at least a minimum of k ports can produce k elements per clock cycle with the appropriate access order of data elements. In the cyclic partitioning case, this entails access orders that make use of data elements having been stored across k banks with a stride of k elements. Thus, each block of k consecutive elements (aligned to 0) can be fetched in a single cycle. Likewise, the block partitioning divides elements consecutively such that the minimum rate of k elements accessible per clock cycle is obtained if data is accessed concurrently with a stride of k elements (again, aligned at 0)[179].

It is clear that if cyclic partitioning is instructured to be performed by a factor that is a multiple of the LDPC code GF(2^m) order, then the complete unrolling of the inner-most loops in the decoding kernels will see iterations successfully scheduled in parallel. Moreover, since the BRAMs technology offers a word access width of 36 or 72 bits, when configured in single- or dual-port operation respectively, we can further improve the bandwidth delivered by the BRAMs by reshaping the arrays from which, and to where, data is streamed by each kernel. This process is shown in Figure 5.21. Cyclic and block reshaping takes a form equivalent to the partitioning process, only that partition occurs after data elements are reshaped onto wider words.

With his scheme, we can effectively provide more data access bandwidth from the BRAM, as under an emulated perspective, if k words fit onto the wider word of the mapping, then each port becomes equivalent to k ports, with regards to the number of ele-

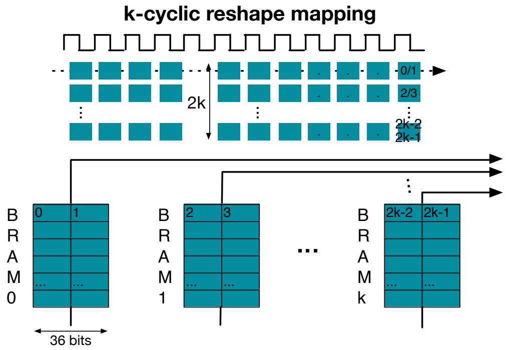


Figure 5.21: Vivado HLS BRAM-array reshaping for improved bandwidth. Assuming data elements of 18 bits, under single-port configuration, each two elements are mapped into a 36-bit word. To this word reorganization is applied the cyclic partitioning, thus making the of the reshape directive a combination of mapping elements into words and partitioning of these wider words.

ments it can produce in a single clock cycle. Thus, pipelining of the outermost loops (c.f. Figure 5.19) will not be as constrained by limited bandwidth as it is under HLS models that do not allow the designer to explicitly optimize the data layout in the BRAM memory space[177,261], as dicussed under the OpenCL programming model in Section 5.5.

One way to further improve the access bandwidth within the memory space provided by BRAMs is to forgo of full-precision data elements and define fixed-point data types appropriately. While under the C/C++ programming languages, there is no support for arbitrary precision elements, as so happens with certain C extensions that add support for hardware constructs but not data types[12,177,201], Vivado HLS effectively extends the language so data arbitrary precision integers and fixed-point data types are available through appropriate C types or C++ classes[179]. This allowed us to readily define the fixed-point precision bitwidth required for the correct functioning of the LDPC decoder, without the need to write supporting arithmetic functions, that replace the set of arithmetic operations at C-synthesis stage. This way, coupled with the configuration of the BRAM memories as two-port interfaces, we are able to push onto a memory transaction with 72 bits nine data elements at an 8-bits bitwidth, for instance.

Code refactoring and Tcl directives The major problem arising from providing an algorithm description in C/C++ for hardware generation, is the ability of the designer to step from a software stance onto the hardware generation C-synthesis process and realize that equivalence of expressions are not realized by the compiler. In other words, while the functionality of the C program remains the same through two different approaches, for instance when a pointer is incremented and then dereferenced for data access instead of having the pointer be dereferenced at a certain address, the C-synthesis process produces better quality of hardware for one case and not the other. Solving this singularities can prove to be a tricky process, more than defining an algorithm that is functionally correct—under RTL simulation of the C-synthesized circuits netlist. Certain code refactoring details are discussed in the experimental data obtained for the loop-annotated LDPC decoders.

On the other hand, once the kernels' description is stable, with regards to code refactoring, applying optimizations at a micro-architecture level as deep as pipelining the outermost kernel instead of unrolling it, or splitting data across BRAMs block-wise instead of cyclically, which in HDL would entail a tremendous NRE refactoring of the kernels description, under the discussed model, we can as simply as rewrite the accompanying `Tcl` directives that serve as C-synthesis design constraints, as seen below. In Listing 5.7, the

```
set_directive_resource -core RAM_T2P_BRAM "ldpcDecoder" Lq
set_directive_resource -core RAM_T2P_BRAM "ldpcDecoder" Lr
set_directive_resource -core RAM_T2P_BRAM "ldpcDecoder" LQ
set_directive_array_reshape -type cyclic -factor 16 -dim 1 "ldpcDecoder" Lq
set_directive_array_reshape -type cyclic -factor 16 -dim 1 "ldpcDecoder" Lr
set_directive_array_reshape -type cyclic -factor 16 -dim 1 "ldpcDecoder" LQ
set_directive_unroll -factor 16 "ldpcDecoder/cnUpdate/loop1"
set_directive_pipeline "ldpcDecoder/cnUpdate/loop2"
set_directive_unroll -factor 16 "ldpcDecoder/vnUpdate/loop1"
set_directive_pipeline "ldpcDecoder/vnUpdate/loop2"
```

Listing 5.7: Vivado HLS list of `Tcl` directives for hardware generation.

arrays `Lq`, `Lr` and `LQ` are defined to lay on BRAMs configured with two-port interfaces, to which a cyclic reshaping by a factor 16 is applied at the `ldpcDecoder` kernel level. Furthermore, loops `loop1` and `loop2` are unrolled by a factor of 16 and pipelined for a target II of 1, respectively, for both the `cnUpdate` and `vnUpdate` kernels.

5.4.4 Experimental results

Herein, we discuss the loop-annotated decoder applied to the non-binary LDPC decoding case. To this end, we benchmark dataset V (Table 4.3) by applying the methodology discussed in Section 5.4. The decoding algorithm utilized is the FFT-SPA divided onto two dataflow regions, as illustrated in Figure 5.20, and mapped to nested loop structures as shown in Figure 5.19. We discuss how the different optimizations carried out through code annotations, generally speaking `#pragmas`, but also, code refactoring that leads to better decoding throughput performances. The Xilinx 2014.2 infrastructure was used to generate hardware for the VC709 FPGA board—F5 in Table C.1. The template architecture was defined using the appropriate MIG[264] and clock wizard interfaces and assembled in a Verilog project using Vivado, which also instantiated a number of LDPC decoder sub-systems packaged through the Vivado HLS tool. The hardware utilization levels reported are relative to the fitted design and the decoding latency was extracted using the XSim simulator tool to infer the cycle accurate behavior of the LDPC accelerator[8,9].

Mapping the FFT-SPA to HLS C From equations (2.43), (2.43) and (2.54), in Algorithms 2.9 and 2.12, we can observe that each expression is applied to certain subsets of data, $C(v)$ or $V(c)$ within larger sets $\mathbf{m}_{cv}(x)$ or $\mathbf{m}_{vc}(x)$ at the node level, and at the *pmf* width when traversing an edge, i.e., at the edge-level. Essentially, all the enumerated functions in the isomorphic mapping of the factor graph to hardware blocks (Figure 5.17) will be instantiated in the FFT-SPA case. While under a *single-instruction multiple-thread* (SIMT)-like architecture[170], expressing the non-binary LDPC code dimensions could be efficiently performed by linearizing all dimensions to a one dimensional execution grid of threads (or work-items), as performed under the non-binary wide-pipeline LDPC decoder[6] discussed in Subsection 5.5.2. Translation of all the dimensions onto a single iterator would not be appropriate to finely control the level at which parallelism is exposed. Thus, the tool is explicitly instructed to keep the double- and triple-nested loop structures (Figure 5.19).

The main issue with defining a single-loop is the ability to apply optimizations at the appropriate level—at the GF(2^m) dimension, or node degree level, or even at the number of nodes required to be covered. Thus, the first loop structure in the snippet in Figure 5.8 will generate a bit-true C synthesized accelerator under Vivado HLS. However, the tool will not be able to pick up the optimizations targeted at each dimension level. Even if the optimization is correctly picked up, the tool will incur in too long C-synthesis times that can be overcome by refactoring of the C code to the second loop in Figure 5.8.

```
//flat loop unsuitable for Vivado HLS optimizations
for(int i = 0; i < edges*q; i++){
  int e = i/(d_v*q);          //get VN id
  int g = i%q;                //get GF(q) element
  int t = (i/q)%d_v;          //get d_v element
}

//nested loop suitable for Vivado HLS optimizations
for(int e = 0; e < edges; e++)
  for(int g = 0; g < q; g++)
    for(int t = 0; t < d_v; t++)
```

Listing 5.8: Loop structures suitable and unsuitable for Vivado HLS optimizations.

Hence, in order to allow the correct application of the optimizations discussed next, we label the loop nests according to following nomenclature. 1) The outermost loop, iterating over the total number of edges in the LDPC code is labeled as **E**, 2) the loop iterating over the GF(2^m) dimension q=2^m is designated by **GF** and 3) by **LOGGF** the one iterating over m, with 4,5) the one iterating over d_v/d_c designated as $\mathbf{D_v}/\mathbf{D_c}$. Under this nomen-

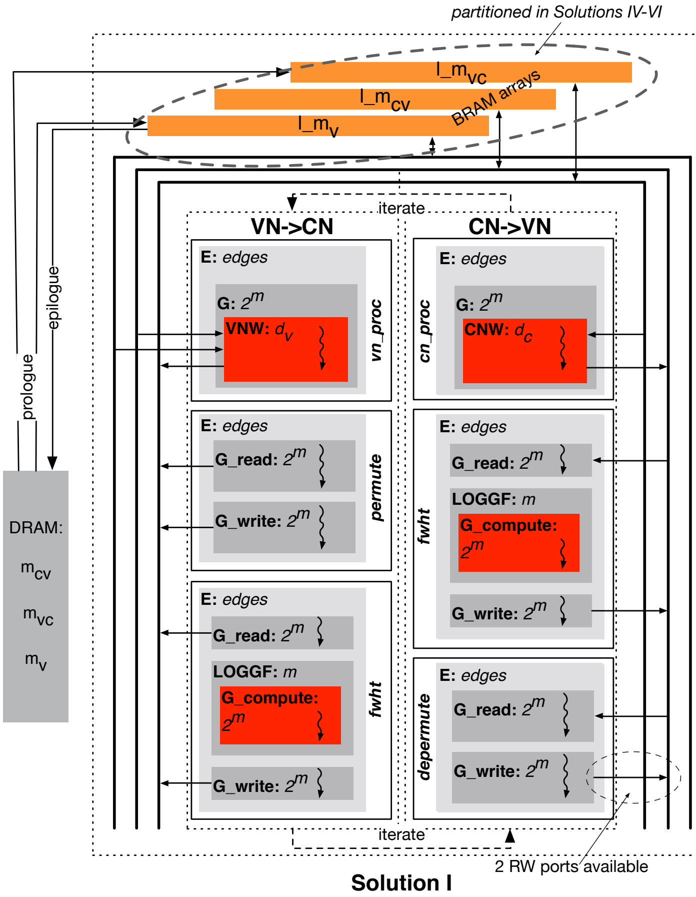


Figure 5.22: LDPC decoder architecture base version (Solution I). In each kernel, the loop label and trip count sizes are shown. Also show in the datapath from the DRAM space to the BRAM spaces inside the processing system.

clature, we can visualize the constructed hardware topology of the decoder as shown in Figure 5.22, which is the base version for the non-binary loop-annotated LDPC decoder. It is a naive version whose generated hardware is bit-true with regards to the C code fed into the HLS tool as the decoder description, but will not see applied the necessary optimizations to maximize the decoding throughput and minimize the decoding latency.

The base decoder provided by Solution I already applies the front-end, processing, back-end division discussed in Section 5.4 (Figure 5.18). Data is streamed onto the VC709 FPGA board through the PCIe interface using *direct memory access* (DMA) transfers to the memory space in the DRAM banks. To maximize the bandwidth obtained from the two 4GB banks in the board, we define data to be streamed onto the board through one memory bank and to flow out of it through another. Then, the streamed data is moved onto the appropriate BRAM arrays which lie on-chip for improved bandwidth and closer spatial proximity to the computation units[a]. Considering that the algorithm reads *pmfs* $\mathbf{m}_v(x)^{(i)}$ and $\mathbf{m}_{cv}(x)^{(i)}$ and writes *pmfs* $\mathbf{m}_{vc}(x)^{(i)}$ in the VN processing (`vnUpdate` kernel)—likewise, reads *pmfs* $\mathbf{m}_{vc}(x)^{(i)}$ and writes $\mathbf{m}_{cv}(x)^{(i)}$ under the CN processing (`cnUpdate` kernel)—each of the *pmf* vectors are defined in the BRAM-allocated arrays `l_mv`, `l_mvc`, and `l_mcv`, respectively for $\mathbf{m}_v(x)^{(i)}$, $\mathbf{m}_{vc}(x)^{(i)}$ and $\mathbf{m}_{cv}(x)^{(i)}$. The discussion concerning each kernel optimization follows next.

VN and CN processing Under FFT-SPA decoding, the VN and the CN processing kernels, `vnUpdate` and `cnUpdate`, perform Hadamard multiplications, or pointwise multiplications. In Solution I, seen in Figure 5.22, each kernel is composed of a triple-nested loop structure **E–GF–D$_\mathbf{v}$/D$_\mathbf{c}$**. In the **E** loop it computes over different messages, with trip count $edges{=}N{\times}d_v$. In the **GF** loop it operates over different probability values (*pmfs*) and in **D$_\mathbf{v}$/D$_\mathbf{c}$** it uses data read from different arrays. Hence, the optimization for these kernels will be to leverage the parallelism from all three loop bodies (Listing 5.9).

```
//nested loop loop structure of vnUpdate
E:for(int e = 0; e < edges; e++){
  GF:for(int g = 0; g < q; g++){
    Dv:for(int t = 0; t < d_v; t++){
      //computation follows
}}}

//nested loop loop structure of permute
E:for(int e = 0; e < limit; e++){
 GF_read:for(int g = 0; g < GF; g++)
    //load data into temporary buffer
  GF_write:for(int g = 0; g < GF; g++)
    //permute and store back to memory
}
```

Listing 5.9: Nested loop structure of `vnUpdate` and `permute` (under the `cnUpdate` kernel a replacement is made on the **E** trip count $d_v{\leftarrow}d_c$).

Permutation and depermutation The permutation and depermutation kernels, `permute` and `depermute`, apply the permutation of probabilities within the *pmf* dimension and are a double-nested structure in the decoding system. In the **GF_read** loop, data is loaded, shuffled according to the non-binary element in the parity-check matrix. defining the permutation or depermutation into a local copy. Then, the second loop, **GF_write**, will write data back contiguously to the correct BRAM location. Since the shuffling is performed in-place, as shown in Listing 5.9, the available parallelization potential will be limited by it, in spite of the memory saving of BRAM units.

[a]While we have not included a module to stream data to the DRAM banks, a PCIe or 10G interface can be included to perform DMA of data to the DRAM. This can be achieved in under 3000 LUTs, without a foreseeable impact on the timing of the LDPC accelerator, as these interfaces would have different clock domains.

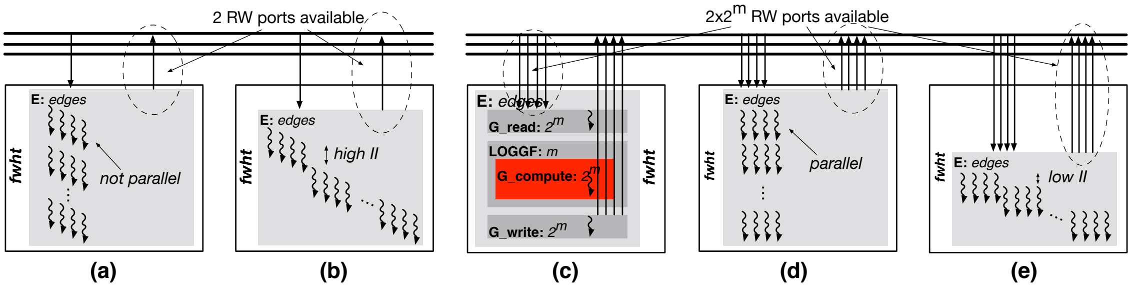


Figure 5.23: Expected behaviour of the FWHT kernel iteration scheduling for Solutions: a) II, b) III, c) IV, d) V and e) VI. Solutions II, III, V and VI (a),b), d) and e))) have all the inner loops of **E** unrolled, and loop **E** pipelined, respectively. Solutions II–III (a) and b)) access BRAM arrays through a double Read/Write (RW) port per array, while Solutions IV–VI (c) and e)) access them via 2^m double RW ports. As a consequence, true parallel execution of unrolled iterations and minimum IIs for pipelined execution is only achieved through the higher bandwidth exposed by 2^m double RW ports of Solutions V and VI (d) and e)). The former is an example of how higher bandwith available without scheduling optimizations yields no improvement of performance and contributes only to a lower efficiency of design.

FWHT processing The transform kernel, `fwht`, implements the FWHT, a special case of the FFT where the twiddle factors W_N^n are always -1 or 1, thus only additions and subtractions are executed in the butterfly computation. In the loop nest shown in Figure 5.22, **E** iterates over all the *pmfs* whose transforms are computed. Since performing the radix-2 factorization of the FWHT in-place would entail tremendous pressure accessing BRAMs, we utilize a temporary array as a scratchpad to hold the working data, even though the transform result is stored in-place. Loops **GF_read** and **GF_write** copy the data to and from this scratchpad in a prologue and epilogue stage of the processing. The **LogGF** loop cycles through the FWHT stages entailed by a radix-2 factorization, and **GF** iterates over each transform element. Here, we achieve parallelism among the different messages—transform batches—in **E** and also **GF**, among different message elements, that can be exploited during optimizations.

```
E:for(int e = 0; e < edges; e++){
  G_read:for(int g = 0; g < q; g++){
    //load data into temporary array
  }
  LogGF:for(int c=0;c<m;c++){
    GF:for(int g = 0; g < q; g++){
      //perform Radix-2 computation
  }}
  G_write:for(int g = 0; g < q; g++){
    //store data back to memory
}}
```

Listing 5.10: Nested loop structure of `fwht`.

Architecture Optimization Guidelines The unoptimized decoder in Solution I achieves only a modest performance since the decoding operations are performed sequentially.

This is because the tool does not automatically apply necessary optimizations to leverage the available parallelism. Hence, to achieve high performance, one needs to explicitly direct the tool to apply the necessary optimizations. Moreover, one needs to carefully consider the hardware implications of the specific optimization and often apply one or more optimization together in order to achieve the intended result. The optimizations carried out, and described next, can be visualized in Figure 5.23.

Loop unrolling tries to schedule multiple iterations in parallel to leverage parallelism and improve processing throughput. In our unoptimized decoder, as aforementioned, we have data-parallelism in the **E**, **GF**, $\mathbf{D_v}$ and $\mathbf{D_c}$ loops. Additionally, we can also unroll the **LogGF** loop to remove the control flow overhead associated with the loop structure, which can be useful when m is small. Solution II and V are generated by performing unrolling on the loops **GF**, $\mathbf{D_v}$, $\mathbf{D_c}$ and **LogGF**. In Solution II, however, since there is insufficient memory ports available to BRAM, we anticipate the data accesses to limit potential performance gains. In Solution V, with additional memory ports made available, we expect that iterations can be fully executed in parallel. Although **E** exposes data-parallelism, we do not unroll this loop since it is the outermost loop and unrolling it would in-turn unroll all the nested loop levels, creating a design that would not fit even on the largest FPGAs available.

This is of particular interest for $\mathbf{VN/D_c}$ loops and for **GF** loops, since there are no data dependencies between each probability elements in the same *pmf* for iterations on both the former and latter. Thus, the unrolling level is a parameter whose ultimate performance is driven by the number of available BRAM ports. The BRAM partitioning factor can thus be used to realistically drive the unroll factor to the level where no more operations will be scheduled. Increase of the unroll factor will result in extra logic consumed at little to no gain in latency of the unrolled loop due to port starvation.

Loop Pipelining Loop pipelining tries to improve loop execution performance by having multiple loop iterations execute on the same hardware. *Initiation interval* (II) is a metric that signifies how soon the loop structure can initiate the execution of a new iteration after having begun the execution of the previous one, (II=1 in the optimal situation). In our decoder, we utilize pipelining to exploit the parallelism that has remained untapped in the **E** loop, when its innermost loops are unrolled, or to exploit parallelism of the innermost loops. As with loop unrolling, the II resulting of pipelining the loops is limited by the number of memory ports that serve the BRAM-allocated arrays from which data is fetched. Unlike loop unrolling, pipelining in Vivado HLS is an optimization that constrains all inner loops to be unrolled prior to pipelining.

A question that remains is what is the most efficient way to combine unrolling and pipelining. Given the nested-loop structure of the LDPC decoder, educated assumptions can be made regarding the best approach. For a decoder whose innermost loops are pipelined, unrolling of the outermost loop will generate an accelerator composed of small pipelined cores. In its turn, this creates extra overhead due to several units managing their own pipelines. While this is not directly a limiting factor to the performance obtained, higher utilization of resources by control units will reduce the slack for better routing and higher clock frequencies. Whereas pipelining of the outermost loop, with the innermost loops unrolled, will generate a single core per decoding kernel which aggregates control logic in a single pipeline. This way, the FPGA logic resources are devoted in a larger fraction to arithmetic and exploitation of parallel instructions and less to logic control. This comes with increased slack in terms of routing and clock and gives margin to replicate more blocks of the LDPC decoder as explained next. Given that Vivado HLS optimizations are directives and the decoding kernels loop-nested structures, testing between both cases is a simple matter of interchanging the unroll and the pipeline directives, as shown in Listing 5.11 for the `vnUpdate`.

```
#a) pipeline outermost and unroll innermost
set_directive_pipeline "vnUpdate/E" -II 1 -rewind
set_directive_unroll   "vnUpdate/GG"
set_directive_unroll   "vnUpdate/Dv"
#b) unroll outermost and pipeline innermost
set_directive_unroll   "vnUpdate/E" -factor U
set_directive_pipeline "vnUpdate/GF" -II 1 -rewind
```

Listing 5.11: Pipeline and unroll optimizations `Tcl` directives for `vnUpdate`. In a), complete unrolling is instructed and pipeline II is tentatively set at 1. In b), unrolling by a U factor is instructed, and pipeline II is tentatively set at 1.

Array Partitioning In order to benefit from the unrolling and pipelining optimizations, we must provide sufficient bandwidth to the design. However, the default strategy of the tool is to sequentially allocate all the data elements into a BRAM unit until a new one is needed. This implies that contiguous data accesses often need to be served by the same BRAM which has only a limited bandwidth from the single or, sometimes, double read-write (RW) port. To alleviate this issue, we instruct Vivado HLS to instantiate dual-ported BRAM memories and, additionally, to partition each BRAM array with a 2^m cyclic factor to expose 2×2^m ports per data array. It partitions each array into 2^m new ones, where contiguous elements of the original one, are spread across the multiple BRAMs as discussed in Figures 5.20 and 5.21. The partitioning enables us to achieve an II=1 for the most complex loops in the design. While array partitioning is useful, it comes with the overhead of computing the indexes where an index i in the original array must be mapped to a 2-D address $(x,y) = (mod(i,2^m), \lfloor i/2^m \rfloor)$, with x the BRAM bank and y the index of i in x bank. Nevertheless, it is also a directive optimization that recomputes the indexes automatically for the developer and breaks the array into several BRAM banks, as seen in Listing 5.12.

```
#store l_mcv of top-level fftspa in 2-port BRAM
set_directive_resource -core RAM_T2P_BRAM
   "ldpcDecoder" l_mcv
#partition array l_mcv cyclically by a factor of 4
set_directive_array_partition -type cyclic
   -factor 4 -dim 1 "ldpcDecoder" l_mcv
```

Listing 5.12: `Tcl` directives that define and partitio an array on BRAMS: `l_mcv` is defined over BRAM units with two R/W memory ports and cyclically partitioned across its first dimension by a factor of four.

Floating- vs Fixed-point In FPGA design, we are not constrained by micro-architecture defined data types, such as single-precision floating-point, and can configure the datapath to use the most convenient one given the application requirements. For our application, we have found that *Q*8.7 fixed-point representation used for the messages exchanged in the FFT-SPA, with the intermediate operations performed in *Q*16.13, lead to simpler synthesized circuits and reduced latency of fixed-point arithmetic—$QX.Y$ standing for $X-Y$ sign and magnitude bits, and Y for decimal bits. This is a design optimization that must be done after carefully considering the characteristics of the specific application, but whose code refactoring can be performed with the inclusion of the `ap_cint.h` library and the `typedef` definition (Listing 5.13).

```
#include<ap_cint.h>
//data is stored in llr type variables
//computation is performed in llr_ type variables
//use floating-point
typedef float llr;
typedef float llr_;
//use Q8.7 fixed-point
typedef ap_fixed< 8, 1, AP_RND_INF, SC_SAT > llr;
typedef ap_fixed< 16, 3, AP_RND_INF, SC_SAT > llr_;
```

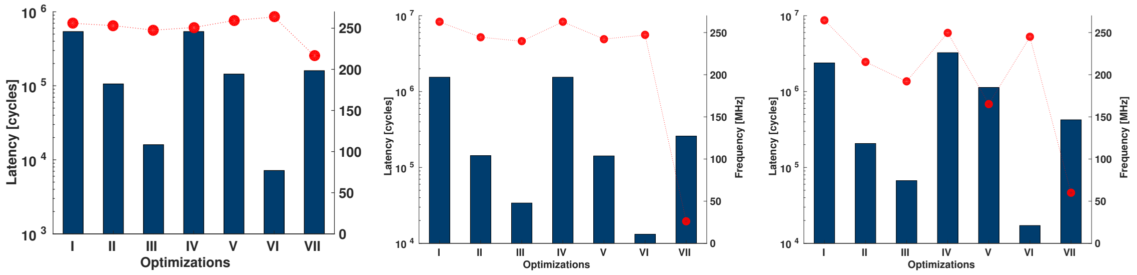
Listing 5.13: Code refactoring performed for the decoder design with fixed-point. The types `llr` and `llr_` are redefined as *Q*8.7 and *Q*16.13 types.

Evaluation of the Decoder Solutions We evaluated the decoder at different design points, Solutions I-VII, describe in Table 5.4, using the code defined in dataset Va–c). During the decoder development, the C-synthesis provides preliminary results to drive the design space exploration. The functional correctness of this synthesized design is then ascertained through RTL co-simulation, which provides a fairly accurate estimate of

Table 5.4: Solutions tested and corresponding optimizations.

Solution	Description of the solution architecture optimizations
I	Base version without C-directives
II	I + Full unrolling of inner loops **LogGF**and **GF**
III	II + Pipelining of outer loops E to II=1
IV	I + Cyclic partition of all BRAM arrays by a factor of 2^m
V	IV + Full unrolling of inner loops **LogGF** and **GF**
VI	III + IV (Unrolling, pipelining and partitioning)
VII	IV + Pipelining of inner loops **LogGF** and **GF** to II=1 and unrolling of outer loop **E** by a factor U=2^m

the overall decoding performance in clock cycles. Finally, before integrating the decoder into the high-level system architecture, we *place and route* (P&R) the decoder design standalone to obtain more accurate values for the hardware utilization and clock frequency. This enables us to estimate how many decoders can be instantiated in the high-level system architecture. Now, after performing P&R on this complete system, we compute the decoding throughput for 10 decoding iterations from the post-P&R clock frequency of this system, the number of decoders instantiated and the decoding latency based on the co-simulation.



(a) GF(2^2) **(b)** GF(2^3) **(c)** GF(2^4)

Figure 5.24: Loop-annotated decoding kernels latency (bars, left axis), and clock frequency of operation (points, right axis) of each solution for a) GF(2^2), b) GF(2^3), and c) GF(2^4).

Base Version The LDPC decoder base version provided by Solution I exploits no parallelism and, therefore, has low resource utilization and achieves a very modest throughput, well within the Kbit/s range. Moreover, the number of clock cycles taken by this design roughly doubles for each increment of m. This version was used for algorithmic validation and served as a baseline to evaluate the other design optimizations.

Loop Unrolling Solutions II, V, VI and VII are the cases that employ loop unrolling, which leads to a reduction in the overall latency of the decoder design, independent of any other optimizations carried out, as seen in Table 5.6. Naturally, unrolling is best

Table 5.5: FPGA utilization for the standalone LDPC decoder IP core.

FPGA	GF(2^2)							GF(2^3)						
Util.[%]	I	II	III	IV	V	VI	VII	I	II	III	IV	V	VI	VII
LUTs	0.76	1.48	2.98	1.07	1.52	4.62	7.67	0.64	2.17	5.04	1.13	5.20	10.4	37.4
FF	0.34	0.71	1.42	0.48	0.81	1.89	2.73	0.28	1.16	2.48	0.53	2.52	3.94	7.69
DSP	0.06	0.44	0.44	0.14	0.44	0.44	0.33	0.06	0.89	0.89	0.06	0.89	0.89	0.66
BRAM	0.31	0.24	0.24	0.41	0.41	0.41	0.41	0.44	0.48	0.68	0.82	0.82	0.82	0.85
	GF(2^4)							GF(2^3) (floating-point)						
LUTs	0.84	3.95	9.80	1.70	10.4	13.5	41.5	0.65	N/A	N/A	1.48	11.7	17.3	N/A
FF	0.42	2.25	4.86	0.97	4.95	5.14	12.9	0.29	N/A	N/A	0.51	3.30	7.56	N/A
DSP	0.44	1.78	1.78	0.22	2.00	1.11	1.33	0.06	N/A	N/A	0.06	1.78	1.78	N/A
BRAM	1.36	0.85	1.09	1.36	1.36	1.36	2.78	0.78	N/A	N/A	1.63	2.72	1.63	N/A

applied in conjunction with other optimizations. Both II and V have limited potential to reduce the decoder latency, as these solutions only expose parallelism to the inner loops. Solution II, counterintuitively, has lower decoding latency than V due to the fact that the re-indexing caused by the cyclic partitioning interferes with the in-place permutations carried out by the `permute` and `depermute` kernels.

Pipelining the computation produces designs that achieve the lowest latency—Solutions III, VI—and also Solution VII. Naturally, pipelining also increases the resource utilization of the FPGA. Among these two first design points utilizing pipelining, Solution VI achieves better performance since the array-partitioning exposes additional memory ports to serve the iterations inside loop **E**. After partitioning the arrays by a 2^m-factor, `vnUpdate` and `cnUpdate` can be optimized to 1 and 2 cycles of II and the `fwht` can achieve an II of 1 cycle. The `permute` and `depermute` kernels have data dependencies that cause its minimum II to grow with the field dimension. An interesting observation in Table 5.6 is that the clock frequency estimates for the most complex Solution after C-synthesis can turn out to be, in some cases, grossly estimated, than that obtained after P&R, a potential pitfall in relying on only C-synthesis estimates for evaluation.

Loop Pipelining Solutions VI and VII, that combine unrolling with pipelining, must be analyzed separately. As the outermost loops **E** are prevented to be fully unrolled by the tool, which imposes a limit to the trip count of loops to be unrolled, only low unroll factors could be set $(U{=}2^m)$, in order for the tool to synthesize the decoder accelerator in a reasonable timespan. Furthermore, as seen in Figure 5.25, Solution VII is a non-optimal Pareto point, using a high number of LUTs for a low reduction in decoder latency.

Table 5.6: LDPC *HLS IP Core* decoder latency and clock frequency.

Sol.	C-Synth'd Design		RTL Co-Sim.	*HLS IP core* P&R'd
	E Lat. [Kcycles]	Clk [MHz]	Lat. [Kcycles]	Clk [MHz]
	GF(2^2)			
I	540	266	607	263
II	121	266	129	253
III	20	266	28	248
IV	508	269	691	251
V	157	266	165	259
VI	8	117	16	264
VII	180	29	187	216
	GF(2^3)			
I	3096	266	1232	262
II	182	266	198	244
III	42	266	58	239
IV	3096	266	1581	263
V	284	266	226	242
VI	26	57	30	247
VII	299	19	315	26
	GF(2^4)			
I	4768	266	2651	265
II	285	266	315	215
III	83	266	114	192
IV	6512	266	3499	249
V	2268	266	1355	165
VI	34	25	50	244
VII	526	117	558	127

Array Partitioning Array partitioning increases the data bandwidth to the computation units by exposing more data that can be consumed in parallel. However, it also comes with a non-negligible cost of re-indexing that consumes resources and increases latency. This effect is visible in Solutions I-IV, and a more pronounced effect in Solutions II-V because of the unrolling that was previously applied, as seen in Figure 5.25. However, when this is applied in conjunction with pipelining, we obtain a 42–75% reduction in latency, as seen while moving from Solution III to VI, but not for the Solution VII.

Replication of Compute Units As seen in Table 5.5, our individual decoding units are fairly small. Moreover, as previously discussed, our high-level system architecture facilitates using multiple decoders to achieve higher throughput. Therefore, in our final design, we utilize multiple HLS IP cores to develop a design that targets an FPGA LUT

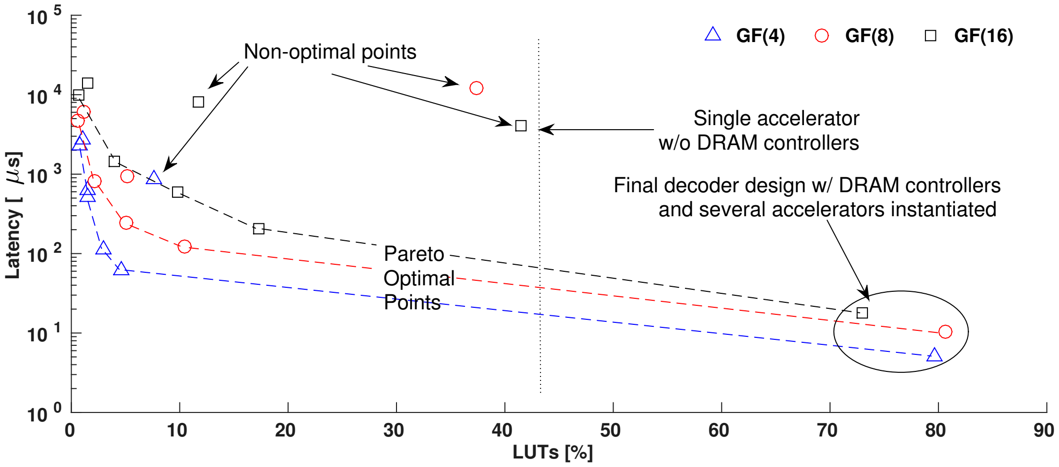


Figure 5.25: Pareto plotting of the design space for the LDPC accelerator in latency (μs) vs. LUT utilization (%). On the rightmost side, the design points correspond to the final decoder with replicated accelerators (14, 5 and 3, respectively for GF($2^2, 2^3, 2^4$)). Latency is per instantiated accelerator.

utilization of 80%. We depend on the resource utilization estimates produced from applying P&R on the standalone *HLS IP core* to guide this step. Using this approach, we were able to instantiate $K = \{14, 6, 3\}$ decoders for $m = \{2, 3, 4\}$. In this multi-decoder design, due to the large design size, we observed a drop in post-P&R clock frequency of operation—dropping by $\{12.4\%, 16.0\%, 6.94\%\}$—compared to the a single decoder design. But, this is well compensated by the improvement in decoding throughput due to the multiple kernels.

Relation to RTL Approaches A handful of publications address the complex design space exploration of LDPC decoder architectures on FPGAs. The use of HLS is equally uncommon, not only for the case of LDPC codes in particular, but also for other signal processing applications. An 802.11n LDPC decoder was designed using Vivado HLS achieving 13.4 Mbit/s throughput for a Spartan 6 LX150T device, operating at 122 MHz and for a frame length of 648 symbols[266].

Complex non-binary LDPC decoder architectures found in the literature for FPGA devices are usually developed at RTL level. Sulek *et al.* have exploited the use of DSP blocks of the FPGA to perform the multipliers used in the CNs and adders in the VNs computation reporting 6 Mbit/s of throughput for code (480,240) with column weight $d_v = 2$ in GF(2^5)[267]. Spagnol *et al.* mixed-domain RTL-based decoder for the GF(2^3) Mackay code obtains 4.7 Mbit/s on a Virtex 2 Pro FPGA[268]. Boutillon *et al.* developed a decoder architecture for a GF(2^6)-LDPC decoder based on the EMS algorithm reporting a decoding throughput of 2.95 Mbit/s for an occupied area of 20% of a Virtex 4 FPGA.

Although their architecture scales with minimal adaptation of the design to higher order GF(q), with $q \geq 2^{12}$, no throughputs are reported for q other than 2^6 [269].

Zhang *et al.* developed a layered partial-parallel architecture, achieving 9.3 Mbit/s throughput at 15 iterations for a GF(2^5) $(744, 653)$ length non-binary code of rate 0.88, with a frequency of operation of 106 MHz on a Virtex-2 Pro[270]. Emden *et al.* study the scalability of the non-binary decoder with a growing GF(2^m) and evaluated their design for a Virtex 5 FPGA, for m=$\{2, 4, 8\}$. Their achieved throughputs range from 1.6 up to 33.1 Mbit/s which illustrate the complexity of the algorithm, as the decoder scales with 11× more area and a throughput 95% inferior[271].

Table 5.7: Comparison of non-binary LDPC decoders: decoding throughput, FPGA utilization and frequency of operation*.

Decoder	m	K	LUT [%]	FF [%]	BRAM [%]	DSP [%]	Thr. [Mbit/s]	Clk [MHz]
This work	2	1	14	7	0.5	0.5	1.17	250
		14	80	35	6	6	14.54	219
	3	1	21	9	0.9	0.9	0.95	250
		6	81	34	5	5	4.81	210
	4	1	30	13	2	2	0.66	216
		3	73	32	5	5	1.85	201
Zhang *et al.* [270]	4	1	48 (Slices)		41	N/A	9.3	N/A
Emden *et al.* [271]	2		N/A				33.16	100
	4						13.22	
	8						1.56	
Spagnol *et al.* [268]	3		13	3	1	N/A	≤4.7	99
Boutillon *et al.* [269]	6		19	6	1	N/A	2.95	61
Scheiber *et al.* [266]	1		14 (Slices)		21	N/A	13.4	122

* Note that differences in technology nodes, FPGA vendor and generation are not considered in this tabulation.

5.5 Wide-pipeline LDPC Decoder

In this section, we discuss the targeting of LDPC decoders when the underlying accelerator architecture generated using a HLS approach is a wide-pipeline accelerator. In particular, our approach is modeled using the commercial Altera OpenCL compiler[177] and also the academic SOpenCL tool[201,254]. Both tools generate FPGA accelerators following a wide-pipeline architecture that can be used as streaming accelerators under the OpenCL context. Thus, this HLS tool is based on the OpenCL supported C programming language, and, as a consequence, the ability to well describe the algorithms is limited by the underlying implementation of the OpenCL programming model. The overview of each OpenCL HLS tool is given on the following subsections, along with the design space exploration performed for each of the developed LDPC decoders.

5.5.1 Altera OpenCL LDPC Decoder

There are many challenges behind the generation of hardware taken as an OpenCL description into an FPGA accelerator. Expressing parallelism at the granularity level of the work-item is particularly useful, since all available parallelism can be exposed to the HLS compiler tool[201,254]. Notwithstanding, mapping of the datapath of each work-item under such fine-granularity is an unrealistic approach to generate hardware from an OpenCL description. Synthesizing and maintaining a hardware accelerator per work-item would lead to tremendous overheads in logic utilization. However, coarsening of the computation from the work-item granularity leads to a better compromise where the exposed parallelism can still be exploited, but at the same time, logic required to handle the parallelism is kept at a minimum. In particular, FPGA designs in the literature have shown that pipelined execution excel in the reconfigurable computing field. Thus, in general, a wide-pipeline approach redefines the OpenCL execution grid as a loop structure that is triple-nested.

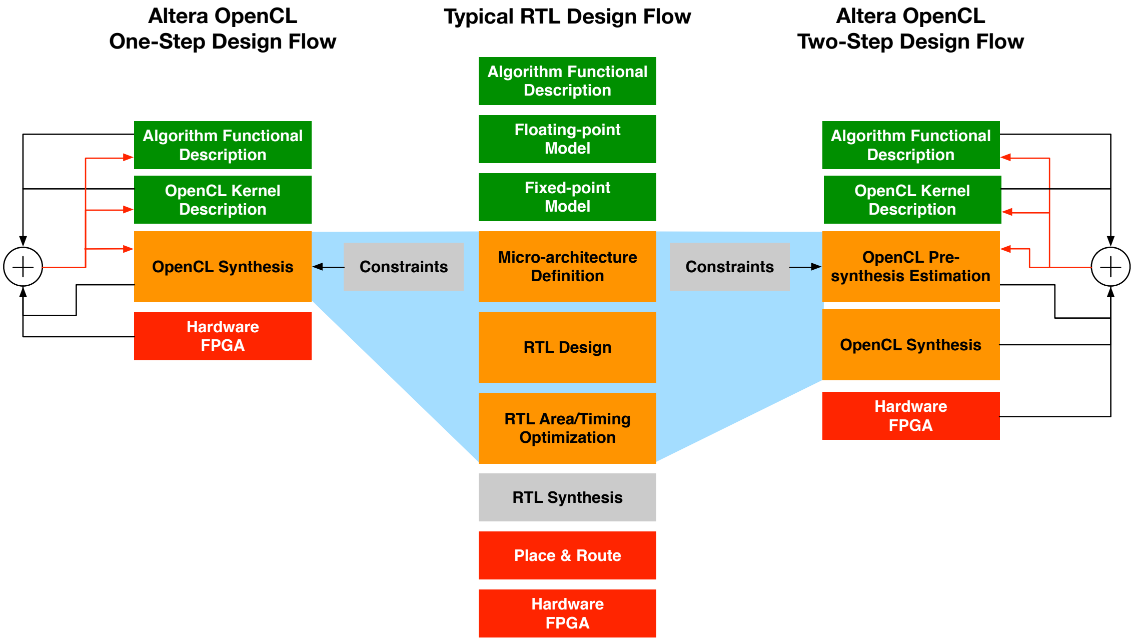


Figure 5.26: Altera OpenCL design flow for hardware development: using one-step flow which reports an accurate performance and utilization report after a bitstream is generated; and using a two-step flow which reports a pre-synthesis report based on conservative estimates for throughput and logic utilization.

The Altera OpenCL compiler is a commercial HLS tool that generates a wide-pipeline accelerator from an OpenCL kernel description[177]. The design flow of this tool modifies the traditional RTL design flow as observed in Figure 5.26. A functional algorithm description provided in the OpenCL supported C language is given as input. After which compilation ensues based on the one of two modes. The simplest and quickest method to obtain a functionally correct wide-pipeline accelerator is to pursue the single-step flow, where the OpenCL kernel is taken straight through until an accelerator bitstream is generated. At this stage, a report is generated, based on the placed and routed hardware accelerator whose information can be used to iterate on the design cycle. Similar to the RTL design flow (Figure 5.6), modifications can be brought to the algorithm functional description. Likewise, optimizations can be made on the OpenCL kernel description and on the set of optimization flags passed on to the OpenCL synthesis process. The two-step flow allows a faster design cycle since a performance report is available before OpenCL synthesis. Arguably will this report reflect the accuracy of a report based on placed and routed circuits. One of the consequences is that this pre-synthesis report is grossly overestimated, leading to reports that can prevent the designer to instruct higher levels of parallelism to be driven.

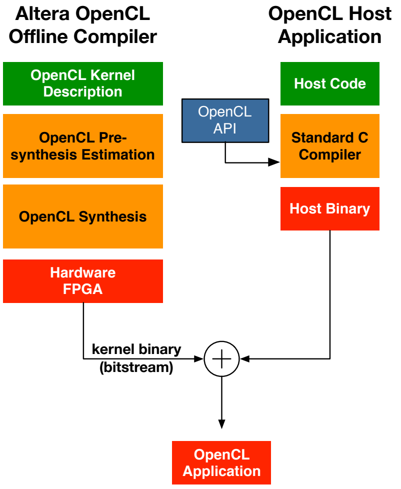


Figure 5.27: Altera OpenCL host application design flow. The OpenCL API links to a C program

Finally, the tool acts as an offline compiler under the OpenCL programming model, as observed in Figure 5.27. Instead of calling the compiler for a given set of kernel inputs at runtime, as is usually performed with GPUs and CPUs, as discussed in Chapter 4, the compiler is called independently from the OpenCL application that loads a pre-compiled binary, in this case a bitstream, with which the FPGA is configured for execution of the kernels on hardware[177].

Parallelism in the Pipeline First, the 3-dimensional execution grid along dimensions 0, 1 and 2, will see its execution serialized. Work-items will no longer be scheduled using the SIMT execution model of GPU architectures, nor will they follow the coarsening of granularity to the work-group level entailed with CPU execution[272]. To illustrate with a simple case the hardware generation that is followed by the Altera OpenCL compiler, we take the vector addition kernel `vectorAdd` defined in Listing 5.14 as an example. Parallelism in the generated accelerator is explored inside the pipeline that defines the work-items datapath. As observed in Figure 5.28 this takes the following form. Work-items are initiated inside the kernel pipeline at a rate that is designated as the *initiation interval* (II). Under the particular case of the Altera OpenCL tool, the hardware designer has no parameter with which the tool can be adjusted for an intended II value. Instead, the tool procures the minimization of the II to having one work-item initiated per clock cycle in each kernel pipeline if no other parallelism directive is given. This case is seen in Figure 5.28a), where there is a work-item in each pipeline stage of the vector addition datapath. On the other hand, while the designer cannot instruct the tool to attempt a higher II, i.e., to have a slower pace of work-item dispatched for execution, one can instruct the tool to effectively lower the II to less than a single clock cycle. This procedure is shown in Figure 5.28b) and c) for two distinct cases. In b) two work-items are initiated in the pipeline simultaneously because every arithmetic and logic control instruction synthesized has the ability to process two work-items in parallel. Effectively, rework of operations to a 2-way *single-instruction multiple-data* (SIMD) level is applied to the kernel pipeline. An equivalent rate of work-items initiation is obtained with the accelerator shown in c). In this case, the same work-item throughput is granted by duplication of the *compute unit*s (CUs) that compose the kernel pipeline shown in Figure 5.28a).

```
// OpenCL kernel
__kernel void vectorAdd(__global int *A,      // input/output vector A
                        __global int *B)      // input vector B
{

  int tx = get_global_id(0);

    A[tx] = A[tx] + B[tx];

}
```

Listing 5.14: OpenCL kernel vector addition example.

Lowering the II through vector processing, through k-way SIMD techniques or through the replication of CUs, achieves in different fashions the same objective of increasing the accelerators throughput by elevating the number of work-items active in the kernel pipeline. However, if the OpenCL kernel has divergent branches of computation in its datapath, the former is inherently more complex. That way bandwidth within each SIMD operation is lower since it may not be guaranteed that the vectorized work-items will be follow similar datapaths. Nevertheless, while the latter method can overcome this issue, since there is no limitation to whether work-items are required to follow the same data-

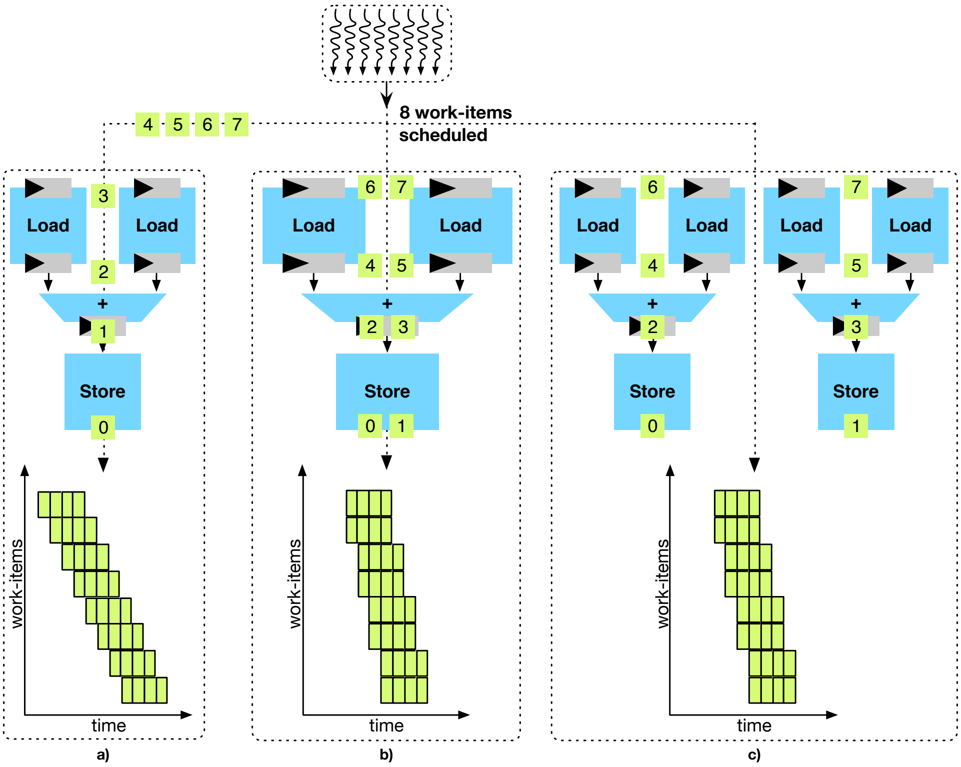


Figure 5.28: Parallelism in the vector addition wide-pipeline accelerators: *a)* initiates a single work-item at a time; *b)* initiates two work-items at a time using 2-way SIMD processing; and *c)* duplicates the CU in *a)* to achieve the initiation rate of *b)*. The time chart shows the scheduling of work-items vs. the processing time.

path, it comes with a higher logic utilization overhead, since not only are the arithmetic units replicated, but also is CU structure pertaining to control.

The throughput T_{wk} expressed as work-items initiated in the pipeline per time unit can be written in the following way

$$T_{wk} = \frac{f}{II} \times d_{connectivity} \times d_{control\ flow} \times d_{global\ mem} \times d_{stalls}\ (\text{work-items/s}), \tag{5.5}$$

with f the clock frequency of operation and II the initiation interval value (5.6). The remaining factors of the expression are designated as derate factors and account for the fact that certain limitations are imposed to the work-items processing throughput brought on by different factors. This can be modeled by routing issues that result in too high critical paths, $d_{connectivity}$, the bandwidth provided by the external memory interface, $d_{global\ mem}$, the overhead that control flow can impose, $d_{control\ flow}$ and finally the stalls endured by the pipeline due to data dependencies, synchronization or fencing instructions, d_{stalls}. The initiation interval is computed as follows

$$II = \frac{1}{k_{SIMD} \times k_{CU}}\ (\text{work-items/clock cycle}), \tag{5.6}$$

where k_{SIMD} and k_{CU} are factors driving the throughput upwards, with the former the k-way SIMD processing levels of the pipeline and the latter the number of CUs replicated in the pipeline[177]. Naturally, the major limitation to k_{SIMD} and k_{CU} is the limited number of logic resources in the FPGA. Notwithstanding, the HLS tool strictly enforces SIMD processing to powers of two up to sixteen ($k_{SIMD} \in \{1, 2, 4, 8, 16\}$), and no maximum limit is imposed *a-priori* on $k_{CU} \in \mathbb{Z}^+$, though in the reported works, k_{CU} has been kept at one order of magnitude[6,7].

Finally, we can write t_{grid}, the time taken to complete the computation of all work-items in an OpenCL execution grid, as

$$t_{grid} = D + II \times N_{work-items} \text{ (clock cycles)}, \tag{5.7}$$

with D the kernel pipeline latency, also referred to as the pipeline depth, and $N_{work-items}$ the total number of work-items scheduled for execution. Analysis of (5.7) shows two corner cases. When the pipeline depth is much higher than the initiation interval factored in by the number of scheduled work-items ($D \gg II \times N_{work-items}$), t_{grid} becomes

$$t_{grid} \approx D \text{ (clock cycles)}, \tag{5.8}$$

and when the II factored in by the number of scheduled work-items is much higher than the pipeline depth ($II \times N_{work-items} \gg D$), t_{grid} can be re-expressed as

$$t_{grid} \approx II \times N_{work-items} \text{ (clock cycles)}. \tag{5.9}$$

Considering that fine-grained algorithm descriptions are pursued for OpenCL-based kernels, it is easily observed that the most likely corner case is when the number of work-items greatly exceeds the pipeline depth. Under this light, is is clear that fine-grained OpenCL descriptions, which expose parallelism at a very fine granularity, and are beneficial to the tool mapping the computation onto circuits, are at odds with the maximization of the obtainable throughput. The finer-grained the higher the number of work-items required to be initiated in the generated kernel pipeline. Furthermore, coarsening the level to which parallelism is exposed can lead to a more complex circuit not able to guarantee the fastest attainable IIs, nor the highest clock frequency of operation.

Memory Model and Template Architecture The OpenCL memory model is mapped to the OpenCL-supported FPGA devices as illustrated in Figure 5.29. The global memory space is mapped to the DRAM units available as external memory to the FPGA chip but that lie in the FPGA board. Since supported FPGA boards come with dual-bank DRAMs, memory accesses can be optimized by the utilization of two distinct physical memory

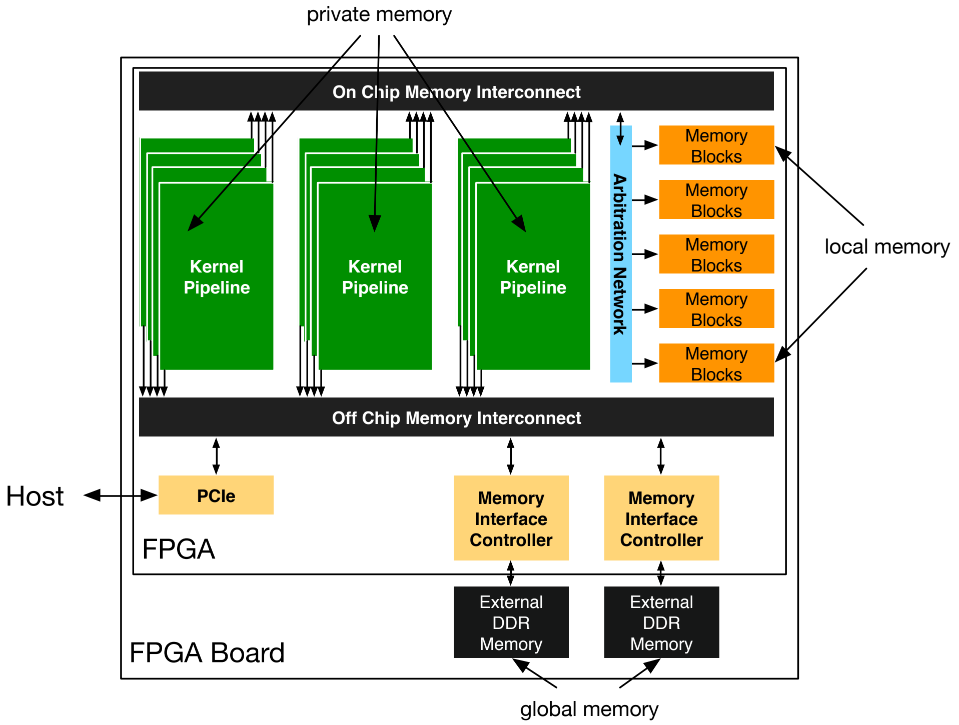


Figure 5.29: Altera OpenCL memory model: *private memory* is synthesized within the kernel pipeline logic; local memory is assigned to BRAMs inside the FPGA, and so is constant memory; and global memory is allocated on the DRAM memory that lies on the FPGA board.

addressing spaces, so as to simultaneously operate the two memory interface controllers. The host system has access to this memory space, a feature required by the OpenCL specification. Constant and local memory spaces are allocated on the FPGA BRAMs, with the former utilizing the BRAMs as ROM, and the latter providing read-write access to all work-items within a work-group. While all the work-items are initiated continuously in the wide-pipeline accelerator, the lifetime of variables in the local memory are still limited to the work-group span[177,273]. Finally, private memory is synthesized in the kernel pipeline logic. Access to memory spaces lying on-chip, constant and local memory, are made through an on chip memory interconnect and accesses made to off chip locations, global memory, are made through an off chip memory interconnect. Absent from Figure 5.29 is a partial reconfiguration module which receives the bitstream with which the FPGA area available for computation, the one corresponding to the kernel pipelines in the figure, is reconfigured[177]. This module reduces the logic available for utilization by the OpenCL kernels. Moreover, due to the partial reconfiguration of the FPGA chip, the template architecture is not optimized to the fullest by the synthesis process. Needless resources are not trimmed away in the generation of hardware. In fact, only the OpenCL kernels are optimized and unnecessary signals are removed from the generated circuits, and certain controllers of the template architecture will lie on the FPGA chip regardless of their underutilization.

Moreover, a disadvantage stands out clearly with regards to flexibility of the imposed memory model. Assuming that all applications benefit from the defined streaming of data from the global memory to computation units and back to global memory simplifies the complexity of the HLS tool. However, it doest not particularly benefit applications, such as the LDPC decoding, which need data to be streamed from the host computer system for computation in the FPGA, and also require that data stays in the FPGA for the duration of the decoding process. The inability to refine the memory model so as to move data from the off-chip DRAM into the on-chip BRAMs adds undesired contention to the external memory interface controllers, and yields a poorer design choice for maximizing the available bandwidth.

Pipelined TpN LDPC Decoder Our design space exploration concerning the OpenCL wide-pipeline architecture is limited to fine-grained parallel expressions, as imposed by the Altera OpenCL compiler. Thus, our approach is the definition of LDPC decoders under the fine-grained parallelism exposed by the TpN approach. As explained in Chapter 3, since the concept of a thread does not exist under the OpenCL programming model, instead, a work-item should be read instead of thread in TpN[177].

Using this model, we can have as a design methodology goal the exploitation of the OpenCL cross-platform capabilities and define two separate kernel instances for the CN and the VN processing, as done in CPU and GPU representations[182]. However, the hardware generation driven by multiple kernel definitions means that multiple pipelines are generated and share the memory interconnect resources of the template architecture illustrated in Figure 5.29. The execution flow for the TpN approach limits the decoding schedule of the LDPC decoder to the TPMP and under this approach each processing phase is executed exclusively. A higher overhead exists, naturally, since more than a single kernel connect to the on-chip and off-chip memory interconnection networks, but there will be no bandwidth contention. Since kernels within the accelerator do not execute simultaneously, the CN and the VN processing cannot overlap in time for coherent decoding, only the CN or the VN request data transactions at a time.

The execution flow of the TpN approach entails other consequences. Each kernel requires the initiation of M and N work-items, respectively for the CN and the VN processing, and their pipelines never overlap in time. In other words, the wide-pipeline accelerator pipelines the work-items of the CN processing and the ones of the VN processing, but not the work-items of the two different processing phases. This behavior is illustrated in Figure 5.30. Under this model, the desired behavior for a multi-kernel

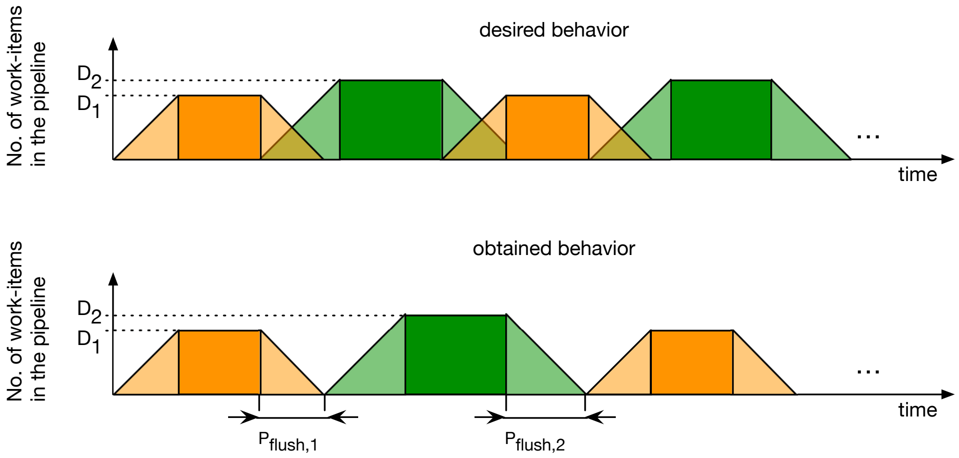


Figure 5.30: Altera OpenCL pipeline desired and obtained execution behaviors. The desired behavior case sees the low throughput phases—the rising throughput prologue and the diminishing throughput epilogue—only once in the OpenCL kernels execution. However, the obtained behavior is different, with each pipeline fully flushed before the execution of the next kernel. Thus, the trailing work-items of the terminating kernel never overlap with the first work-items in the initiating kernel. As a consequence a latency term P_{flush} is added to the execution time of multi-kernel designs.

design such as the TpN decoder, is that the prologue and epilogue of each pipeline are diluted by the epilogue and prologue of adjacent kernels respectively. Naturally this cannot be accomplished for the first and last kernel calls, but between consecutive kernels calls, the overlapping of execution grids should occur, as seen in the top of Figure 5.30. However, this is not the case, with additional latency being added to the decoding time in the form of a flushing penalty P_{flush}. As a consequence we modify the processing time of an execution grid (5.7) to account for multiple grids and this penalty as

$$t_{dec} = (D_1 + D_2 + II \times (N + M)) \times N_{Iter} + \sum_i P_{flush,i} \text{ (cycles)}, \tag{5.10}$$

with M and N the number of work-items scheduled for execution by the TpN decoder and $P_{flush,i}$ the flushing penalty between the flushing of kernel i and the initiation of kernel $i+1$, and D_i the depth, or latency, of each pipeline.

Our discussion of the limitations brought on by the flushing of pipelines in a multi-kernel scenario has thus far neglected the fact that the there is a near-zero potential for node execution overlapping under the TPMP decoding schedule. Notwithstanding, the majority of standardized LDPC codes possess RA structures that introduce a connection between the PNs of the code with CNs with the same indexes[183]. Thus, the trailing VNs (PNs) under execution are connected only to the bottommost CNs and CN execution starting on the topmost nodes should not see incoherent message-passing in the Tanner graph. Herein, an opportunity exists for exploring overlapped processing of the work-items involved in the VN and the CN kernel. Equivalently, there could be an overlap in time of the processing of the CN and the VN kernels, if a proper rescheduling of nodes is performed such that the trailing work-items in the CN processing never overlap with the first work-items in the VN processing (pertaining to the execution of *information node*s (INs)). Hence, instead of assigning an identity function to mapping work-items to nodes, we modify this mapping such that execution of the first VN does not coincide with VNs with which the last CNs executing are connected. In other words, VNs to work-items mapping maintains the identity function, while the CN to work-item mapping is reworked.

Furthermore, to be able to extract overlapping execution of work-items in the CN and VN processing phases in the wide-pipeline LDPC decoder, we need to part from the multi-kernel scenario into a single kernel scenario. Due to the OpenCL programming model this is easily accomplished by a mere refactoring of the previously separate kernel instances into a single kernel definition along with appropriate work-item control flow. Additionally, in order to perform fully pipelined execution, instead of defining how many decoding iterations are issued by how many execution grids are enqueued for execution on the accelerator by the host, there will be a single kernel call, enqueueing $N_{iter} \times (M + N)$ work-items *a-priori*. Then, inside the kernel, work-items falling in the first $M + N$ work-items lie in the first decoding iteration span, of which the first M work-items are responsible for the CN processing, the last N for the VN processing and the next $M + N$ work-items fall in the span of the second iteration. This deeply-pipelined execution flow is depicted in Figure 5.31 By refactoring the multi-kernel TpN decoder

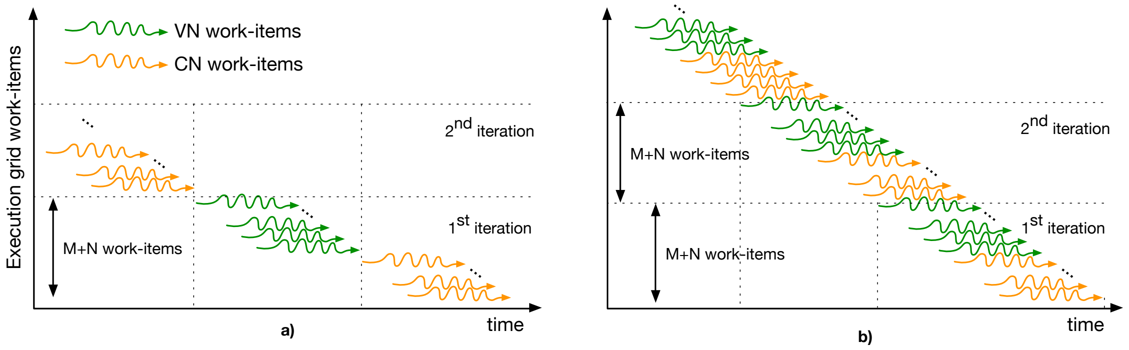


Figure 5.31: Altera OpenCL TpN execution and work-item scheduling: following *a)* a multi-kernel strategy, where an execution grid per iteration per processing stage is issued, and thus, completely flushed, and using *b)* a deep-pipeline approach where all work-items are scheduled at once and the single-kernel pipeline is never flushed.

to a single-kernel deep-pipelined approach, as seen in Listing 5.15, we are able to fully pipeline the execution of the TpN decoder. Furthermore, this way we are able to have a

```
// Multi-kernel TpN approach
__kernel void cnUpdate(__global int *Lq,    // L(q) messages
                       __global int *Lr);   // L(r) messages

__kernel void vnUpdate(__global int *Lq,    // L(q) messages
                       __global int *Lr,    // L(r) messages
                       __global int *LQ);   // L(Q) messages
```

```
// Single-kernel deep-pipeline TpN approach
__kernel void deepTpN(__global int *Lq,    // L(q) messages
                      __global int *Lr,    // L(r) messages
                      __global int *LQ)    // L(Q) messages
{
  unsigned int tid     = get_global_id(0);
  unsigned int nodeID  = tid % (N + M);
  unsigned int iter    = tid / (N + M);

  if(iter<Niter){
    if(nodeID < M)
      cnUpdate(Lr,Lq,nodeID);
    else
      vnUpdate(Lq,Lr,LQ,nodeID);
  }
}
```

Listing 5.15: OpenCL kernel containers for the TpN approaches. The core computation performed at the CN and VN level (`cnUpdate` and `vnUpdate`) are kept the same, but their OpenCL kernel qualifier (`__kernel`) is removed in favor of a C function, called by the `deepTpN` deep-pipeline single-kernel approach.

decoding latency given by

$$t_{dec} = (D + II \times (N + M)) \times N_{Iter} \text{ (cycles)}, \tag{5.11}$$

instead of the one obtained in the multi-kernel scenario (5.10).

The Non-binary Decoding Case The aforementioned discussion between the multi-kernel and the deeply-pipelined single-kernel approach is to a certain extent limited to the binary decoding case. On the one hand, this is due to the Tanner graph structure. Binary LDPC codes, especially the standardized ones, have structured Tanner graph properties, making it possible to organize the nodes execution schedule so as to not interfere with the coherent consumption and production of data elements[7]. On the other hand, since the same BER performance can be obtained with much shorter non-binary codes, the pressure to structure the Tanner graph adjacencies are not so impending as in the binary case. To a certain extent, in the non-binary case the field dimension works as an expansion factor. The same code can be expanded or shortened in length by driving the order of GF(2^m)[192,274] without a higher overhead in nodes' adjacencies mapping. Most of the non-binary LDPC codes found in the literature are built with methods that leave little slack for the decoder design to reshuffle the nodes execution order[275–277].

$$\mathbf{H} = \begin{bmatrix} \alpha & 0 & 1 & \alpha & 0 & 1 \\ \alpha^2 & \alpha & 0 & 1 & 1 & 0 \\ 0 & \alpha & \alpha^2 & 0 & \alpha^2 & 1 \end{bmatrix}$$

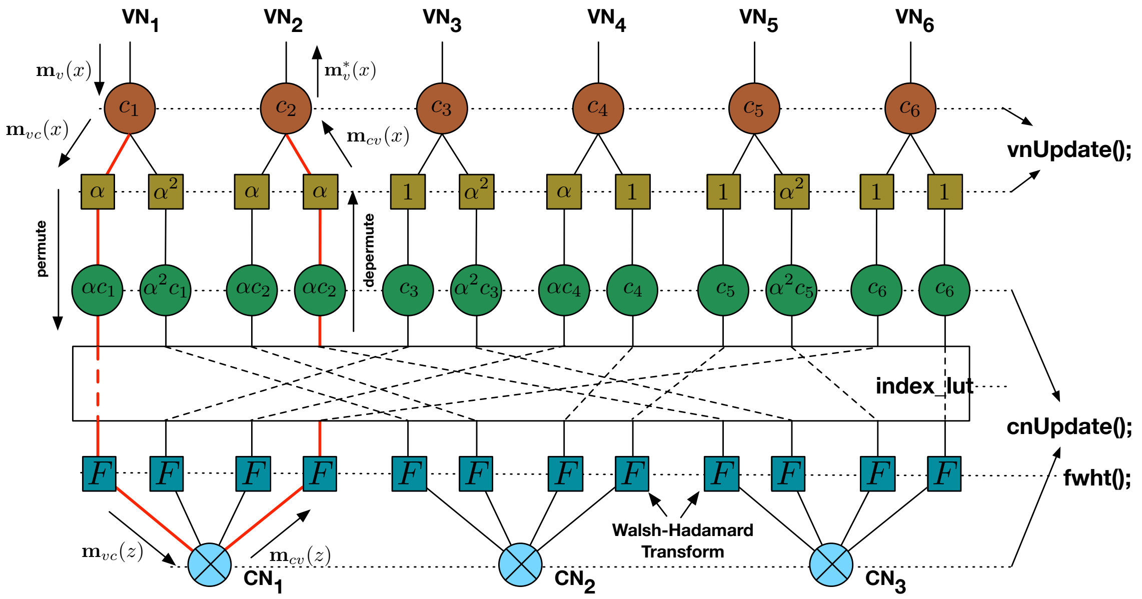


Figure 5.32: Altera OpenCL isomorphic mapping to a multi-kernel approach. Therein, permutation and depermutation functions are moved to the node-level and the operational transform subsides at the edge-level, allowing the incorporation of other edge-level functions. The multi-kernel approach then allows for distinct execution grids for each kernel case (`cnUpdate`, `vnUpdate` and `fwht`).

As a consequence, we are limited to the multi-kernel design approach for the non-binary LDPC decoding case. The same dataflow regions as those discussed for the loop-annotated decoder in Section 5.4 apply, although the isomorphic mapping of the Tanner graph onto hardware blocks differs (c.f Figure 5.17). However, explicitly defining the node- and edge-level operations in a single dataflow kernel does not configure what we foresee as the best approach to extract parallelism at a fine-level under an OpenCL environment. For that matter, different execution grids are better for certain operations than others, while some features of the edge traversing message can be fully absorbed into a node-level kernel. For instance, permutation of messages that traverse the edges can be merged onto the CN and VN kernels, or fully merged onto one of them exclusively[11]. However, we cannot forgo of the fact that, under the case that the FWHT is applied at the edge-level, its performance peaks for a different execution grid than the work-item configuration utilized for the CN and the VN update kernels[14]. The majority of times, the most suitable dimensions of the execution grid depend on the factorization that work best for a particular transform size, which for LDPC codes over GF(2^m) is a 2^m-point transform[13,239].

Considering these limitations, we are able to isomorphically map the non-binary LDPC decoding case as illustrated in Figure 5.32. Therein, the permutation and depermutation operations are moved from the edge-level to the node-level kernels defined by the CN and the VN update kernels `vnUpdate` and `cnUpdate`. Only the operational transform is left as an edge-level function. Due to its particular properties under fine-grained parallel OpenCL description, requiring an execution grid dimension loosely independent of the grid dimensions deployed for CN and VN update, and also due to the fact that the majority of non-binary Tanner graphs cannot support nodes reordering compatible with the deeply-pipelined approach, the non-binary decoder accelerator is defined over the multi-kernel approach. All surveyed algorithms in Chapter 2 can be supported by it, other features such as domain conversions can be inbuilt in the FWHT kernel, or replace it altogether, at the edge-level. The drawback of this approach, is that a pipeline flushing penalty is introduced between each kernel execution, thus adding at most three distinct pipeline flushes for each decoding iteration (5.10). Although, the upside of this approach is that different configurations for (k_{SIMD}, k_{CU}) can be obtained for each kernel singlehandedly.

Data-parallelism Knowing, beforehand, that the non-binary case on account of the multi-kernel approach will incur in heavy penalties due to the flushing of the wide-pipeline kernels on the accelerator, we try and mitigate the effect by increasing the data-parallelism level. This can be performed in one of two ways, *i)* we can simply add more codewords to the execution grid by initiating more work-items in the execution grids and keep adding words to memory with a stride defined by a codeword. The benefit is not so much in actual acceleration, since there are no more words actually being decoded in parallel but rather waiting to having their execution work-items initiated onto the pipeline accelerator. The prologue and epilogue regions (when the pipeline is not fully occupied by work-items), which are responsible for the flushing overhead (Figure 5.30), take a smaller proportion of the execution time and thus, flushing penalties become lower in magnitude when compared to the actual time taken by each kernel when the pipelines are fully occupied. This strategy is equivalent to that discussed for GPU decoders in Chapter 4 and comes with the downside of driving latency upwards. Since one of the motivations we took to move onto hardware design was its potential to allow for real-time decoding systems, driving latency for the sake of diluting the flushing penalties will most likely render a decoder system inoperable in real-time conditions. On the other hand, *ii)* we can increase the workload by packing more works onto a vector datatype so that each work-item performs more computation.

However, a limitation arises when the LDPC decoding algorithms require fixed-point computation. Despite the support of IEEE-compliant floating-point[177], in most occasions floating-point can be deprecated in the design in favor of fixed-point arithmetic.

The 1.1 OpenCL specification employed, does not define arbitrary precision datatypes, making fixed-point representations dependent on explicit design of the multiplication and division instructions, and making them limited to the bitwidths of the supported datatypes, at least in the bitwidth of the container type. Thus, if we use the supported datatypes with regards to the LDPC decoding algorithms arithmetic nature, floating-point utilization is reserved to algorithms applying multiplication or division, and fixed-point to the ones applying additions and subtractions only[11,182]. Under the supported datatypes, this limits the number of floating-point codewords that can be packed in a vector type to 16 in a `float16` and under 8-bit fixed-point representation to 64 codewords in a `int16` vector.

5.5.2 Experimental results

The aforementioned discussed methodology for the development of the wide-pipeline decoder has been tested considering the scenarios II, IIIa–d) and Va–c) (c.f Table 4.3) under different configurations that capture LDPC codes and decoding algorithms' features. The first, dealing with scenario II is a multi-kernel TpN approach where there is a synchronization point between each phase of the LDPC decoding process[6]. Under scenarios IIIa–d), we consider the case of the deeply-pipelined work-items where they are instantiated by the same execution grid onto the LDPC decoder accelerator[7]. Finally, we consider the multi-kernel case when the decoder has had non-binary decoding kernels instantiated, following a *thread-per-edge* (TpE) work-item granularity. The HLS compiler tool utilized was the Altera OpenCL 13.0SP1 using the Nallatech 385 N FPGA board with a Stratix V D5 chip (F4 in Table C.1). We make a distinction between the binary LDPC decoding case (datasets II and III) and the non-binary case (dataset V), and thus we discuss them separately in the following paragraphs, before drawing conclusions to the wide-pipeline accelerator, after the discussion surrounding the wide-pipeline approach conveyed by the SOpenCL tool (Subsection 5.5.4).

A note must be given regarding the logic utilization levels, based on the pre-synthesis (c.f. Figure 5.26), placement and routing, and fitting reports[177]. If we were willing to present equivalent logic utilization metrics, such as those reported for the dataflow approach or the loop-annotated methodology, we would not be able to do so straightforwardly. Whereas we can report the relative utilization of BRAM and DSP resources, the method used by the Altera tools to report the utilization level of resources within the FPGA chip is different. For the one, we cannot refer to LUT utilization, but only to *adaptive LUT* (ALUT). However, it is unknown how many ALUTs can be allocated by a design, since each LUT provides for two ALUTs. On the other hand, the basic CLB module of the FPGA is an *adaptive logic module* (ALM), yielding a combined metric of LUT and FF utilization, although we can not explicitly state if the utilization of ALMs is being driven by LUT or by FF consumption. Notwithstanding, the reports convey information regarding the usage of *logic element*s (LEs) and FFs separately, even though the concept of a LE has become increasingly blurred as the FPGA chip families evolved[177,278]. In the metrics presented next, we use LE as a proxy to LUT and ALM as a metric for the overall CLB utilization[278]. Furthermore, the discussed utilization levels omit the architecture hardware blocks that do concern the kernel pipelines, i.e., the on- and off-chip interconnections and the memory controllers to the external DRAM interfaces are not accounted for (Figure 5.29).

Table 5.8: FPGA logic resource utilization, clock frequency, throughput and latency at 10 decoding iterations for benchmarked LDPC decoders. Also, decoding throughput is measured relative to each core, and the minimum decoding iterations allowed by the external interfaces are tabulated.

		Scenario					
		II		IIIa)	IIIb)	IIIc)	IIId)
	Parallelism (k_{SIMD}, k_{CU})	(1,1)	(1,2)	(1,7)	(1,7)	(1,6)	(1,6)
Resource Util. (%)	Logic Utilization	53.30	77.64	64.88	70.54	59.19	64.12
	LEs	28.13	41.20	43.35	47.38	39.42	42.87
	FFs	22.69	35.87	44.22	47.54	39.42	42.27
	BRAMs	42.75	66.24	81.48	84.61	72.59	75.27
	DSPs	1.82	3.65	2.20	4.40	1.89	3.77
Performance	Clock (MHz)	240.00	157.00	212.00	223.0	204.00	203.00
	Thr. (Mbit/s)	16.0.0	21.00	98.70	103.9	81.40	81.0
	Thr./Core (Mbit/s/core)	16.0.0	10.50	14.10	14.80	13.60	13.50
	Min. Dec. Iter.	0.02	0.03	3.00	2.10	1.20	0.90

Binary Multi-kernel and Single-kernel Deeply-pipelined Approach The accelerator developed using the dataset II considers the following scenario. The MSA decoding algorithm is employed and the granularity to which parallelism is exposed is defined at the TpN level. This guarantees that there is no redundancy of memory operations and, also, that data can be block fetched from memory since the QC-LDPC Wi-Fi code of dataset II allows for coalesced memory accesses in blocks of z_f LLR messages[1,131]. The place and routed design characteristics of this decoder accelerator are summarized in the first two columns in Table 5.8.

As observed, we have been able to drive the generation of more than a single CU, but the utilization level lead to a maximum of 2 CUs instantiated only. Logic utilization (expressed in utilized ALMs) does not double, nor does the utilization of FFs or BRAMs, though there is a doubling of utilized LEs. DSPs utilization also doubles even when there the applied arithmetic is mainly composed of additions and subtractions, although the computation of the indexes of LLRs requires the multiplication and modulo division.

While some of the effects of doubling the number of CUs in the design are not observed in the same increase factor of utilization, some are directly related to it. The former is due to the template architecture to which the kernels pipelines are hooked up. As seen in Figure 5.29, the higher the number of kernels in the accelerator design, the higher the logic overhead due to a higher number of ports required in the on-chip interconnection network to feed all the required data to each kernel pipeline.

To improve the throughput achieved with the multi-kernel wide-pipeline accelerator, the deeply-pipelined methodology allows the forgoing of the flushing of work-items in each kernel pipeline. However, this is made at the expense of rescheduling of the nodes execution such that data produced by certain nodes does not overlap with its consumption by their adjacent nodes. To this end, we can perform a remapping of work-item index to Tanner graph node index, as discussed next.

Node Rescheduling Considering that an in-order execution of the workgroups exists under the Altera OpenCL implementation, we can guarantee the execution order of each node in the Tanner graph. Unlike GPU engines, we are not, thus, limited to enforcing a global synchronization only through the execution of two separate execution grids as so happens in the multi-kernel approach just discussed. We can exploit the in-order execution to our benefit by merging the CN and the VN kernels and remapping the CNs and the VNs order of execution so that a memory hazard does not happen. In particular, if we assume an identity mapping of the work-item index to Tanner graph index, we must focus our attention to connectivity of the first CNs/VNs to the last VNs/CNs. Under the QC-LDPC codes class, this analysis is easier to perform, since the parity-check matrix **H** is obtained from the expansion of the protograph **F**. We associate the nomenclature $\mathrm{CN}_i^{'}/\mathrm{VN}_i^{'}$ as the set of CNs/VNs with indexes in the range $\{i \times z_f, (i-1) \times z_f - 1\}$.

Analysis of the base matrix for scenario II, $\mathbf{F_{wifi}}$ (5.12) shows that the dense columns $\mathrm{VN}_{0,8}^{'}$ make it impossible to reschedule the node execution order.

$$\mathbf{F_{wifi}} = \left[\begin{array}{lcccccccccccccccccccccccc}
 & VN_0^{'} & VN_1^{'} & VN_2^{'} & VN_3^{'} & VN_4^{'} & VN_5^{'} & VN_6^{'} & VN_7^{'} & VN_8^{'} & VN_9^{'} & VN_{10}^{'} & VN_{11}^{'} & VN_{12}^{'} & VN_{13}^{'} & VN_{14}^{'} & VN_{15}^{'} & VN_{16}^{'} & VN_{17}^{'} & VN_{18}^{'} & VN_{19}^{'} & VN_{20}^{'} & VN_{21}^{'} & VN_{22}^{'} & VN_{23}^{'} \\
CN_0^{'} & 57 & \infty & \infty & \infty & 50 & \infty & 11 & \infty & 50 & \infty & 79 & \infty & 1 & 0 & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\
CN_1^{'} & 3 & \infty & 28 & \infty & 0 & \infty & \infty & \infty & 55 & 7 & \infty & \infty & \infty & 0 & 0 & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\
CN_2^{'} & 30 & \infty & \infty & \infty & 24 & 37 & \infty & \infty & 56 & 14 & \infty & \infty & \infty & \infty & 0 & 0 & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\
CN_3^{'} & 62 & 53 & \infty & \infty & 53 & \infty & \infty & 3 & 35 & \infty & \infty & \infty & \infty & \infty & \infty & 0 & 0 & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\
CN_4^{'} & 40 & \infty & \infty & 20 & 66 & \infty & \infty & 22 & 28 & \infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 & 0 & \infty & \infty & \infty & \infty & \infty & \infty \\
CN_5^{'} & 0 & \infty & \infty & \infty & 8 & \infty & 42 & \infty & 50 & \infty & \infty & 8 & \infty & \infty & \infty & \infty & \infty & 0 & 0 & \infty & \infty & \infty & \infty & \infty \\
CN_6^{'} & 69 & 79 & 79 & \infty & \infty & \infty & 56 & \infty & 52 & \infty & \infty & \infty & 0 & \infty & \infty & \infty & \infty & \infty & 0 & 0 & \infty & \infty & \infty & \infty \\
CN_7^{'} & 65 & \infty & \infty & \infty & 38 & 57 & \infty & \infty & 72 & \infty & 27 & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 & 0 & \infty & \infty & \infty \\
CN_8^{'} & 64 & \infty & \infty & \infty & 14 & 52 & \infty & \infty & 30 & \infty & \infty & 32 & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 & 0 & \infty & \infty \\
CN_9^{'} & \infty & 45 & \infty & 70 & 0 & \infty & \infty & \infty & 77 & 9 & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 & 0 & \infty \\
CN_{10}^{'} & 2 & 56 & \infty & 57 & 35 & \infty & \infty & \infty & \infty & \infty & 12 & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 & 0 \\
CN_{11}^{'} & 24 & \infty & 61 & \infty & 60 & \infty & \infty & 27 & 51 & \infty & \infty & 16 & 1 & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & 0
\end{array}\right] \tag{5.12}$$

For instance, the transition of the VN processing to the CN processing could be easily dealt with by execution $\mathrm{VN}_{0,8}^{'}$ first, and then proceeding with the remaining $\mathrm{VN}_{i \setminus \{0,8\}}^{'}$. However, this would draw memory hazards to the other transition phase, when the last

CNs executing are doing so at the same time the first VNs commence execution. Consequently, dataset II has been profiled using the multi-kernel approach[6], while datasets III, which can also be benchmarked in the multi-kernel approach, were profiled under the deeply-pipelined accelerator applying the remapping next discussed[7].

$$\mathbf{F_{wimax}} = \left[\begin{array}{ccc|cccccccccccccccccccccccc}
 & & & VN_0' & VN_1' & VN_2' & VN_3' & VN_4' & VN_5' & VN_6' & VN_7' & VN_8' & VN_9' & VN_{10}' & VN_{11}' & VN_{12}' & VN_{13}' & VN_{14}' & VN_{15}' & VN_{16}' & VN_{17}' & VN_{18}' & VN_{19}' & VN_{20}' & VN_{21}' & VN_{22}' & VN_{23}' \\
CN_0' & \rightarrow & CN_0' & \infty & 94 & 73 & \infty & \infty & \infty & \infty & \infty & 55 & 83 & \infty & \infty & 7 & 0 & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\
CN_1' & \rightarrow & CN_1' & \infty & 27 & \infty & \infty & \infty & 22 & 79 & 9 & \infty & \infty & \infty & 12 & \infty & 0 & 0 & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\
CN_2' & \rightarrow & CN_2' & \infty & \infty & \infty & 24 & 22 & 81 & \infty & 33 & \infty & \infty & \infty & 0 & \infty & \infty & 0 & 0 & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\
CN_3' & \rightarrow & CN_3' & 61 & \infty & 47 & \infty & \infty & \infty & \infty & \infty & 65 & 25 & \infty & \infty & \infty & \infty & \infty & 0 & 0 & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\
CN_4' & \rightarrow & CN_4' & \infty & \infty & 39 & \infty & \infty & \infty & 84 & \infty & \infty & 41 & 72 & \infty & \infty & \infty & \infty & \infty & 0 & 0 & \infty & \infty & \infty & \infty & \infty & \infty \\
CN_5' & \rightarrow & CN_5' & \infty & \infty & \infty & \infty & 46 & 40 & \infty & 82 & \infty & \infty & \infty & 79 & 0 & \infty & \infty & \infty & \infty & 0 & 0 & \infty & \infty & \infty & \infty & \infty \\
CN_6' & \rightarrow & CN_6' & \infty & \infty & 95 & 53 & \infty & \infty & \infty & \infty & \infty & 14 & 18 & \infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 & 0 & \infty & \infty & \infty & \infty \\
CN_7' & \rightarrow & CN_7' & \infty & 11 & 73 & \infty & \infty & \infty & 2 & \infty & \infty & 47 & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 & 0 & \infty & \infty & \infty \\
CN_8' & \rightarrow & CN_8' & 12 & \infty & \infty & \infty & 83 & 24 & \infty & 43 & \infty & \infty & \infty & 51 & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 & 0 & \infty & \infty \\
CN_9' & \searrow & CN_{11}' & \infty & \infty & \infty & \infty & \infty & 94 & \infty & 59 & \infty & \infty & 70 & 72 & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 & 0 & \infty \\
CN_{10}' & \rightarrow & CN_{10}' & \infty & \infty & 7 & 65 & \infty & \infty & \infty & \infty & 39 & 49 & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 & 0 \\
CN_{11}' & \nearrow & CN_9' & 43 & \infty & \infty & \infty & \infty & 66 & \infty & 41 & \infty & \infty & \infty & 26 & 7 & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & 0
\end{array}\right] \tag{5.13}$$

A simple reordering of the protograph to nodes' execution schedule can be performed unraveling all out-of-order consumptions of data and, thus, eliminating all memory hazards, for the remapping in the second column in $\mathbf{F_{wimax}}$ (5.13). Therein, CNs in the range CN_9' are swapped with CN_{11}', enabling the deeply-pipelined approach for all z_f in the *worldwide interoperability for microwave access* (WiMAX) standard. This approach has been empirically verified, with the identity mapping giving rise to memory hazards, and the optimized mapping not.

There are limitations to the aforementioned remapping. First one is its *ad hoc* nature with a limited methodology applied to solving any memory hazards arising from a deeply-pipelined execution of nodes. While it is fairly easy to formalize the data dependencies, we can only do so at a programming level that lies above the actual movement of data in the kernel datapath. As a consequence, we can only empirically verify that the decoder behaves well in the remapped case, but produces out-of-order memory accesses that make for non-coherent data consumption, thus compromising the whole decoding process, in the non-remapped case. Two approaches could solve this problem, but entail increasing the decoding latency. The first, requiring an analysis of the generated circuits, would require the introduction of stall cycles in the pipeline in the most critical cases so that coherent consumption of data can occur. Another approach would entail raising the number of words that are being decoded simultaneously, not necessarily by raising the data-parallelism, but by initiating more work-items in the pipeline. This way, and by interleaving execution of nodes so that consecutive work-items update consecutive nodes of different codewords, an appropriate number of codewords to be simultaneously decoded can be put forward that ensures that no memory hazards occurs. As said, this would increase latency, the latter most certainly beyond the point admissible for real-time decoding. Furthermore, the last strategy depletes the FPGA ability to enqueue more work-items, since a new factor contributes to increasing their number, in addition to the nodes and the iterations, several codewords.

Logic Utilization and Throughput Analysis As discussed previously in Subsection 5.5.1, the logic utilization of the wide-pipeline accelerator is limited in the design flow process, illustrated in Figure 5.26, by what is reported at the pre-synthesis report. Given that at that stage, no synthesis of circuits has occurred, there is a tendency to overestimate the consumed logic that then limits the number of CUs, or the SIMD level, that can be defined on the wide-pipeline accelerator to further boost the initiation of work-items[177]. As a consequence, the accelerators designed for scenarios II and III, show logic utilizations levels that are fairly low after placement and routing, even though the pre-synthesis reports utilizations close to 90% of LEs. This causes the maximum number of CUs of the TpN multi-kernel approach to peak at only 2 and for the deeply-pipelined accelerator, we are able to define 6 and 7 CUs. The reason for not defining an equivalent SIMD level, is related to the single codeword approach taken for the decoders. Due to this, it is harder for the compiler to automatically vectorize processing. In fact, carried out experiments showed that a potential SIMD levels higher than 1 were definable, but the throughput of work-items did not increase by the same factor due to the compiler inability to vectorize processing. As a consequence, the accelerators incur in a higher overhead for instantiating more CUs. We could argue that BRAMs utilization is what actually is limiting the feasibility of the instantiation of more CUs. However, BRAM utilization is expected to increase slightly with each CU and its utilization can be considered fairly independent from it. Another interesting observation is the optimizations are more effective for powers of 2 dimensions. Scenario IIIc) sees a lower utilization of logic resources when compared to b) and d). This is due to z_f=64 being a power of 2, making any multiplications and modulo divisions employed by the indexing of messages exchanged in the Tanner graph be defined by simpler shift operations. The simplicity of the operations defined by the MSA and the indexing of the Tanner graph is patent on the low level of DSPs utilization.

However, power of 2 expansion factors for QC-LDPC codes in use with the IEEE Wi-Fi and WiMAX standards are the exception and not the rule. In any case, the enhanced result of the optimizations under theses cases is not observed at the pre-synthesis report, but it rather unfolds as a consequence of the synthesis of the generated circuits. Thus, we have not been able to explore this feature for improved parallelism, as the number of CU instantiated is in line with the remaining decoders tested.

An interesting result, highlighted by the clock frequency of operation obtained, is the ability to capitalize on an increased number of CUs in order to improve the decoding throughput. For scenario II, which defines a two-kernel accelerator, when multiple CUs

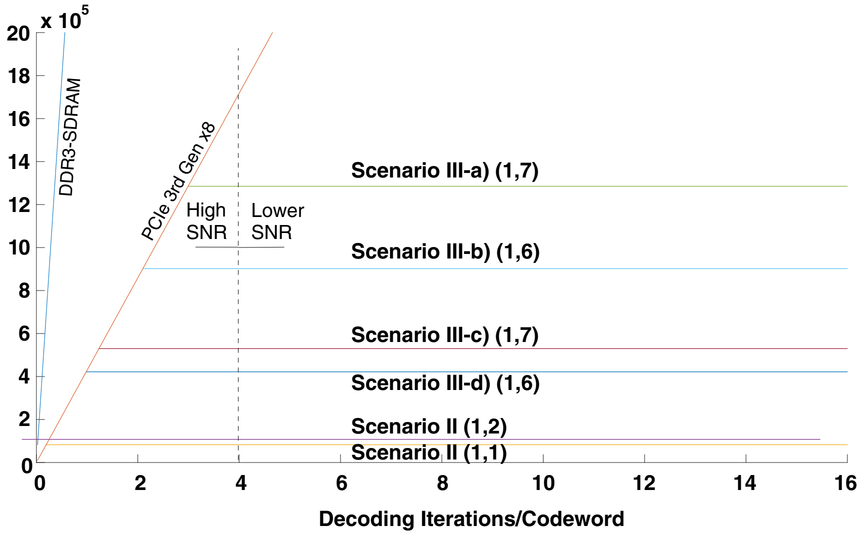


Figure 5.33: Altera OpenCL roofline analysis of the LDPC decoder for the considered scenarios II and IIIa–d). As observed throughput is at such low levels, when compared to the available memory bandwidth in the DRAM interface that the data required to decode can be produce in under a tenth of a single decoding iteration, while the PCIe bus can deliver data at most after 3 decoding iterations.

are instantiated leads to routing congestions shown in the form of a much lower clock frequency of operation than the target 250 MHz. On the other hand, these routing issues are not observed for the deeply-pipelined single-kernel approach. Therefore, we can speculate on the ability of the wide-pipeline accelerator approach ability to place and route a multi-kernel design without incurring heavy penalties when the complexity of the design is driven up by multiple CUs. Compounded by the fact that the accelerator has seen a penalty yield in the decoding latency due to flushing of each kernel pipeline between the CN and the VN processing, and also between the VN and the CN, it is clear that multi-kernel strategies should be left for accelerator architectures that do not introduce an *a-priori* latency penalty, especially per kernel launch as it will impact on iterative kernels case, such as LDPC decoding. Thus, a deeply pipelined single-kernel strategy is the most suitable approach for the wide-pipeline accelerator, granting higher clock frequencies of operation, non-stalling computation and the ability to improve the number of CUs initiating work-items in the accelerator[7].

Roofline Analysis The roofline analysis of the developed LDPC decoders is useful in order to determine how constrained by the bandwidth of the external interfaces they are[105]. The utilized FPGA board, under the decoder architecture discussed in this subsection, poses bandwidth constraints at the PCIe bus in the interaction of host system to FPGA accelerator, and inside the FPGA board due to the off-chip DRAM interfaces. Those constraints are drawn in Figure 5.33 on the left-hand side[6].

Bandwidth of the external interfaces and maximization of the throughput mean that very high throughputs are, naturally, more limited by lack of bandwidth than lower throughput decoders. In this particular case, we are able to obtain moderate throughputs, the lowest for the considered scenario II using the TpN approach. In this case, we can observe that both the DRAM and the PCIe interface can deliver data in under a single decoding iteration. In a sense, not being constrained by the bandwidth of the external interfaces is a positive feature of the LDPC decoder accelerator. However, in practice this means that either the bandwidth of the DRAM and the PCIe interfaces are over engineered for the accelerator, or being the former fixed and not adjustable or swappable by other components, means that the accelerator is under performing for the board capabilities target. Hence, the decoder considered under scenario II is under performing with regards to the board capabilities. On the other hand, for the deeply pipelined approach considered under scenarios III, the accelerator shows a higher level of constraints induced by the bandwidth of the interfaces to external memory. In this case, operating at the highest throughputs (measured as decoding iterations per second) can only be performed after 3, 2.1, 1.2 and 0.9 decoding iterations, respectively for IIIa–d).

As discussed previously in the dataflow decoder case, this roofline analysis allows us to draw insights regarding the utilization of the LDPC decoder accelerators under different SNR conditions. Whereas previous works found in the literature consider the BER performance with regards to decoding throughput, decoding iterations or its dependence to the SNR conditions, we draw a dependence of the SNR conditions with the throughput achievable. If the operating SNR condition leads to a number of decoding iterations lower than the identified turning points, the decoder will not be able to comply with the peak decoding data rate for which it has been designed[6].

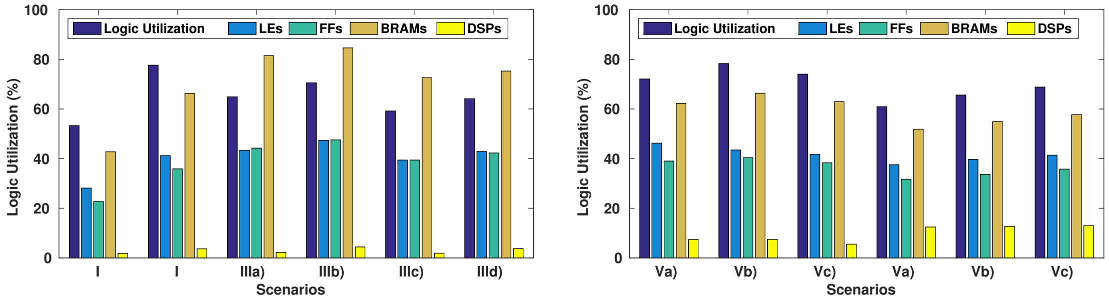
Non-binary Multi-kernel Approach The non-binary wide-pipeline accelerator must be analyzed under a slightly different light than the binary decoders. In this case, we benchmark dataset V, for low orders of the binary extension field, respectively $\{2^2, 2^3, 2^4\}$ for the considered scenarios Va–c). The logic utilization of the LDPC decoder accelerator, as a whole, is tabulated in Table 5.9, considering scenarios Va–c) using data-parallelism level of 1 codeword$*$ and 4 codewords†. Raising of the data-parallelism level took into account not the expansion of the number of work-items in the execution along with an increasing number of decoding words, but instead the packing of 4 codewords onto a vector type. Considering that under the OpenCL programming model there is only defined support for integer and floating-point types, under the C canonical types, the FFT-SPA benchmarked is defined over floating-point representation. Hence, the quad-codeword approach sees the loading and storing of `float4` types to DRAM. In a sense, we can designate this as manually defining a SIMD-level of 4, since the decoding kernels apply arithmetic operations at a bitwidth of 128 bits with a word bitwidth of 32 bits. In addition, the OpenCL compiler has been instructed to automatically find the best combination of (k_{SIMD}, k_{CU}) for the three kernels complying to an early estimation of less than 85% utilized resources.

Table 5.9: FPGA logic resource utilization, clock frequency, throughput and latency at 10 decoding iterations for benchmarked LDPC decoders. Also, decoding throughput is measured relative to each core, and the minimum decoding iterations allowed by the external interfaces are tabulated.

		Scenario V*		
		a) m=2	b) m=3	c) m=2
	Parallelism (k_{SIMD}, k_{CU}) `cnUpdate,FWHT,vnUpdate`	$\{(2,1),(2,1),(2,1)\}$	$\{(4,1),(2,1),(2,1)\}$	$\{(2,1),(2,1),(2,1)\}$
Resource Util. (%)	Logic Utilization	72.09	78.31	74.02
	LEs	46.22	43.47	41.70
	FFs	39.01	40.39	38.32
	BRAMs	62.26	66.34	62.96
	DSPs	1.82	3.65	2.20
Perf.	Clock (MHz)	163.07	188.96	193.16
	Throughput (Mbit/s)	1.08	0.82	0.68
		Scenario V†		
	Parallelism (k_{SIMD}, k_{CU}) `cnUpdate,FWHT,vnUpdate`	$\{(2,1),(1,1),(2,1)\}$	$\{(2,1),(1,1),(2,1)\}$	$\{(2,1),(1,1),(2,1)\}$
Resource Util. (%)	Logic Utilization	60.92	65.61	68.83
	LEs	37.54	39.70	41.38
	FFs	31.69	33.66	35.76
	BRAMs	51.84	54.92	57.70
	DSPs	4.40	1.89	3.77
Perf.	Clock (MHz)	206.52	216.07	203.5
	Throughput (Mbit/s)	3.36	1.73	0.98
	* - 1 codeword; †- 4 codewords			

Counter-intuitively, the level of logic resource utilization lowers with an increasing number of decoded words, with a consistent less than 10% resources drawn for the `float4` operations than for the `float` ones. The exception to this trend is the increased number of DSP units which increases, but again with an exception for scenario Vb), seeing a lower DSP usage. Simultaneously, the clock frequency of operation increases with the increase in data-parallelism. Since there is a replication of the applied operations by a factor of 4, the fitting tool has a better opportunity at coming up with a better routing solution than when the same set of instructions is applied to less data.

Comparison of the utilization of logic resources in the FPGA between the benchmarked binary and non-binary cases show that utilization is similar levels for both approaches, as illustrated in Figure 5.34. Though binary decoders required almost an extra 20% of BRAM in their accelerator design, the consumed ALM units lie in the same $[60\%, 80\%]$ utilization range. As a consequence, considering the overwhelming difference in numerical complexity of the MSA and the FFT-SPA decoding algorithms, discussed in Chapter 2, the decoding throughput of the non-binary decoding case is much lower than that obtained for the binary accelerators. In particular, the single codeword accelerator sees a peak throughput of Va) 1.08 Mbit/s dropping to Vc)0.68 . However, the quad-codeword approach, which would asymptotically yield a speedup of 4×, conveys a throughput of Va) 3.98 Mbit/s dropping to Vc) 0.98. Visualization of the speedups in clock and throughput is illustrated in Figure 5.35, and if we consider the following expression combining the speedups, and slowdowns, of the two data-parallelism levels as



(a) Binary scenarios

(b) Non-binary scenarios

Figure 5.34: Binary LDPC wide-pipeline decoder logic utilization: LEs+FFs; BRAMs; and DSPs in percentage.

$$T_4 = T_1 \times S_{clock} \times S_{data} \times \frac{1}{S_{latency}}, \tag{5.14}$$

where T_4 is the throughput of the quad-codeword accelerator, T_1 is the throughput of the single-codeword decoder, S_{clock} is the clock increase factor from the quad- to the single-codeword case, S_{data} the increase in data-parallelism and $\frac{1}{S_{latency}}$ the overall increase in latency, we can evaluate how the tool is able to vectorize the processing of the codewords. For overall increases in throughputs of $\{3.11, 2.11, 1.44\}\times$, with data increasing by fourfold and the clock frequencies improving by factors of $\{1.27, 1.14, 1.05\}\times$, the equivalent latencies have increased by factors of $\{1.63, 2.16, 2.92\}$. Clearly, while there is still a positive net gain when driving up the data-parallelism levels by packing more data onto wider words, there are diminishing returns on the obtained throughput due to an overall latency increase of the kernels pipeline. Naturally, some of this increase is the responsibility of the lower levels of SIMD (k_{SIMD}) obtained for the kernels' pipeline. Considering that each kernel is executed by an equivalent number of work-items we can look at the multi-kernel approach as a single-accelerator with k_{SIMD} equal to the harmonic mean of the number of the different k_{SIMD} levels in each kernel compounded by the number of work-items executed by each kernel.

$$k'_{SIMD} = \frac{3\times N\times 2^m}{\frac{N\times 2^m}{k_{SIMD,CN}} + \frac{N\times 2^m}{k_{SIMD,FWHT}} + \frac{N\times 2^m}{k_{SIMD,VN}}} \tag{5.15}$$

In (5.15), k'_{SIMD} is an equivalent SIMD level for the LDPC decoder as whole, the number of executed work-items is $N\times 2^m$ and $k_{SIMD,i}$ is the k_{SIMD} implemented by the tool for kernel i. Replacing in (5.15) by the corresponding values from Table 5.9 for each scenario in V, we can calculate the latency increase factors due to the decrease in parallelism levels of the kernel pipelines (due to lower k_{SIMD}) as $\{1.6, 1.3, 1.3\}\times$. While for scenario Va) this solely explains the latency increase, for Vb) and Vc) there is still an effective increase of 1.62× and 2.19× kernel pipeline latency increase.

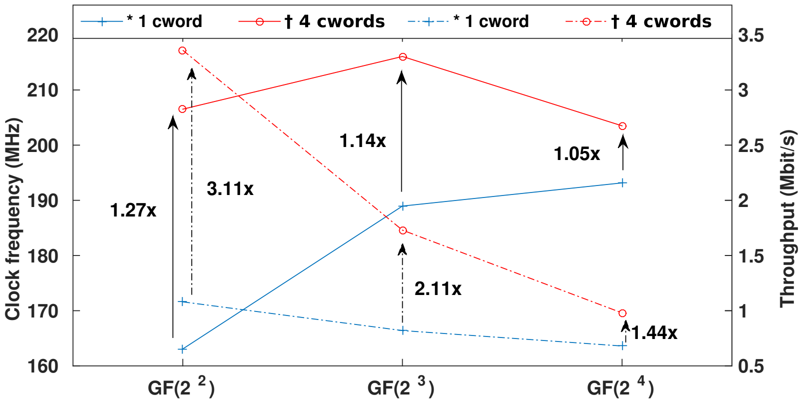


Figure 5.35: Non-binary clock frequency and throughput tradeoff with data-parallelism. Solid lines stand for clock frequency of operation (MHz) while the dash-dotted stands for achieved throughput (Mbit/s), also shown the clock frequency (solid) and throughput (dash-dot) increase factors.

OpenCL Limitations One of the benefits of using OpenCL, which is not having to worry with the accelerator micro-architecture components, such as the DRAM memory interface, DMA blocks and clock configuration, is also one of its pitfalls. In particular, the flexibility of design with an FPGA is greatly hindered by the logic division of the OpenCL memory model and how data therein retained has a limited lifetime that makes sense under CPU and GPU execution, but less so in a reconfigurable substrate. Whereas in the previous section we discussed how the BRAMs could be configured so that data would not have to flow through DRAM each time, the OpenCL programming model [b)] is lacking this functionality. Considering that under the OpenCL programming model, the local memory is mapped to BRAMs, instead of a fixed configuration that is observed for the GPU engine, it could be fine-tuned to a particular kernel/application. However,

[b)] OpenCL 2.0 has begun to incorporate functionalities oriented towards FPGAs than those provided by version 1.1, the version when OpenCL FPGA debut took place.

this still does not address the fact the data therein has only the lifetime of the kernel that called it. As a consequence data must necessarily flow through the DRAM-mapped global memory space.

In addition to this, the wide-pipeline engine, although well-suited for a range of applications that require one time calling per computed workload. is not the best approach for the LDPC decoder whose execution features are substantially different. The iterative nature of the algorithm dealing with error-correction makes it impossible to know beforehand how many decoding iterations are required. Although an average value can be estimated, even under high SNR operating conditions the decoder system will execute the maximum number of iterations for codewords that contribute with errors to the BER at the error-floor region. Thus, frequent calling between the host and the device is needed. Again, this is a legacy drawn from the way that computers are built and to the strategy that FPGA manufacturers supporting OpenCL chose to introduce their hardware. The majority of the FPGA supported models are included in boards that are connected through the PCIe interface for power and data I/O, just like a GPU device.

5.5.3 SOpenCL LDPC Decoder

SOpenCL is an academia led-effort compiler tool, based on the *LLVM compiler infrastructure* (LLVM), that generates a synthesizable HDL accelerator from an OpenCL kernel description[201,254]. The compiler applies a number of source code transformations to the OpenCL kernel descriptions, in order to allow efficient Verilog generation. The source code transformations are applied at the front-end level, while architectural transformation are applied at the back-end.

Front-end The front-end consists of a number of stages applying code transformations. *i) logical thread serialization*, whereas OpenCL kernels perform computation by an up to tri-dimensional execution grid composed of work-items divided onto workgroups, in the absence of synchronization points in the code, computation in the wide-pipeline accelerator will be serialized. Thus, code will be transformed so that the OpenCL kernel body is expressed by a triple-nested loop. An example is illustrated in the `oclKernel` kernel in Listing 5.16a) whose transformed code is re-expressed as shown in Figure 5.16b). *ii) barrier elimination* where barriers and other synchronization points in the kernel description are allowed by OpenCL within the work-group granularity. Therefore, in its triple-nested loop equivalent, all iterations prior to the barrier must complete execution before the next iterations can be dispatched. Essentially, by performing loop fission, the SOpenCL front-end guarantees that the whole computation body is performed prior to *iii) variable privatization*. At his stage loop fission performed to re-express computation in the triple-nested loop while respecting the enforced barriers brings about challenges regarding the lifetime of variable that need to be kept before and after the barrier. In this case, semantics dictates that such variables be overwritten by other iterations, leaving the computation largely incoherent. As a consequence, such variables are privatized.

```
// SOpenCL function container
__kernel void oclKernel(/*...*/){

  int tx = get_global_id(0);
  int ty = get_global_id(1);
  int tz = get_global_id(2);


  oclKernel_stage1(tx,ty,tz,/*...*/);
  barrier(CLK_LOCAL_MEM_FENCE);
  oclKernel_stage2(tx,ty,tz,/*...*/);


}
```

a) SOpenCL kernel example.

```
// Transformed C function container
void cKernel(/*...*/){

  for(int tx = 0; tx < Nx; tx++)
    for(int ty = 0; ty < Ny; ty++)
      for(int tz = 0; tz < Nz; tz++)
        cKernel_stage1(/*...*/);

  for(int tx = 0; tx < Nx; tx++)
    for(int ty = 0; ty < Ny; ty++)
      for(int tz = 0; tz < Nz; tz++)
        cKernel_stage2(/*...*/);

}
```

b) SOpenCL kernel post-front-end transformations.

Listing 5.16: SOpenCL kernel source transformations: *a)* computation in the original `oclKernel` is performed by work-items across an execution grid that synchronize between `oclKernel_stage1` and `oclKernel_stage2`; *b)* the resulting C function after the front-end transformations are applied replaced the execution grid with triple-nested loops, and reworked the barrier through loop fission.

After these transformations have been performed, the re-expressed OpenCL kernel is given to the back-end for generation of synthesizable HDL.

Back-end The output of the SOpenCL front-end is put through the LLVM and an LLVM *intermediate representation* (IR) of it is generated so that standard compiler optimizations can ensue. The process next described is the so called back-end of the compiler and is composed of a series of steps, illustrated in Figure 5.36, that are described next. *i) bitwidth optimization* is an automated procedure aiming the minimization of the number of bits required to represent each operand[279]. In particular, it draws on code constructs such as array bounds, loop iteration counts and type casts to trim any bitwidth that had been defined as unnecessarily wide. The scope of this optimization is at the integer and fixed-point arithmetic, boolean operations and bit manipulations, that can be found across the instructions issued by the LDPC decoder. *ii) predication* allows for conditional execution of instructions. Typically, a 1-bit predicate dictates the execution based on certain conditions, allowing the removal of the majority of branches in the code. This strategy

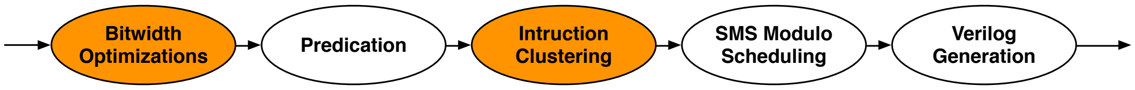


Figure 5.36: Sequence of SOpenCL backend code transformations and optimizations. Highlighted are the optimization steps of particular interest to LDPC decoding.

greatly simplifies the task of efficient instruction scheduling and hardware generation, as a branching functionality is replaced by select instructions managed by predicate variables. *iii) code slicing* will split each SOpenCL kernel onto three stages 1) *input stream kernel* consisting of all load instructions or instructions participating in load address computation, 2) *output stream kernel* that is similar to the above, only for write instructions and 3) *computation kernel* defines the core of the accelerator. It comprises the instructions received from the Input Stream Units and produce data to the output stream units. Since the SOpenCL defines in-order datapaths, computation units just push or pop data, according to their consumption/production patterns, without the need for a matching address. *iv) instruction clustering* sweeps the DFG of the operations in the algorithm at hand

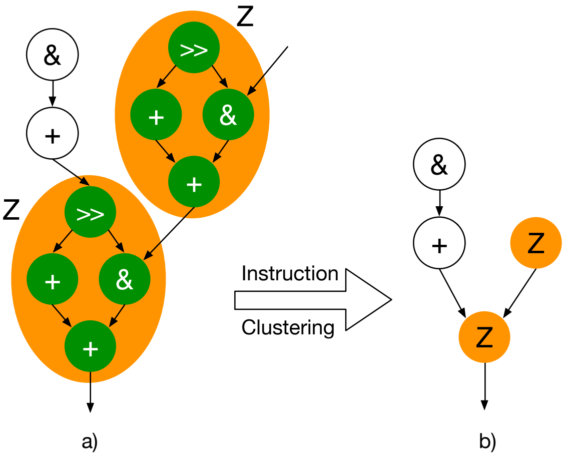


Figure 5.37: Instruction clustering of arithmetic operations. The partial DFG in *a)* has a regular pattern of instructions that is clustered to a macro-instruction *Z*, utilized in the redrawn DFG in*b)*.

finding computation patterns, i.e., combinations of instructions that appear repetitively, that can be replaced by a single macro-instruction encompassing all the computation in the pattern. This process is illustrated by the DFG in Figure 5.37 and follows a grammar-based approach[12]. The implementation is tuned to the underlying FPGA design so that the macro-instruction can be implemented in a single LUT. If, however, computation ensues from one LUT to another, then there is a need to register the result passing between LUTs. Finally, *v) swing modulo scheduling* provides instruction scheduling exploiting *instruction level parallelism* (ILP) in loops by overlapping execution of iterations[280]. This leads to parallelism within a pipeline, as each SMS'd loops initiate iterations at a certain rate, designated as II and quantifies the timing difference in clock cycles between the initiation of each consecutive iteration. Minimization of the II delivers in the highest possible parallelism for any given loop, the most contributing factor to high throughputs. This design feature is a property of the wide-pipeline accelerator, whose discussion is set in the beginning of this section.

Template Architecture The generation of a synthesizable HDL accelerator is based on a template HDL architecture, which is illustrated in Figure 5.38. This way, the compiler is able to leverage on a working setup that is optimized and tuned to the underlying OpenCL kernel at hand. The datapath is composed of a network of FUs that produce and consume data elements using FIFO channels to the streaming units. To this end a multiplexer tree and buffers is generated to allow the correct production-consumption patterns to be realized. The required control logic to operate it is spatially located close to their corresponding FUs and other logic. The rationale is avoid long critical paths on the FPGA with the distributed control logic model.

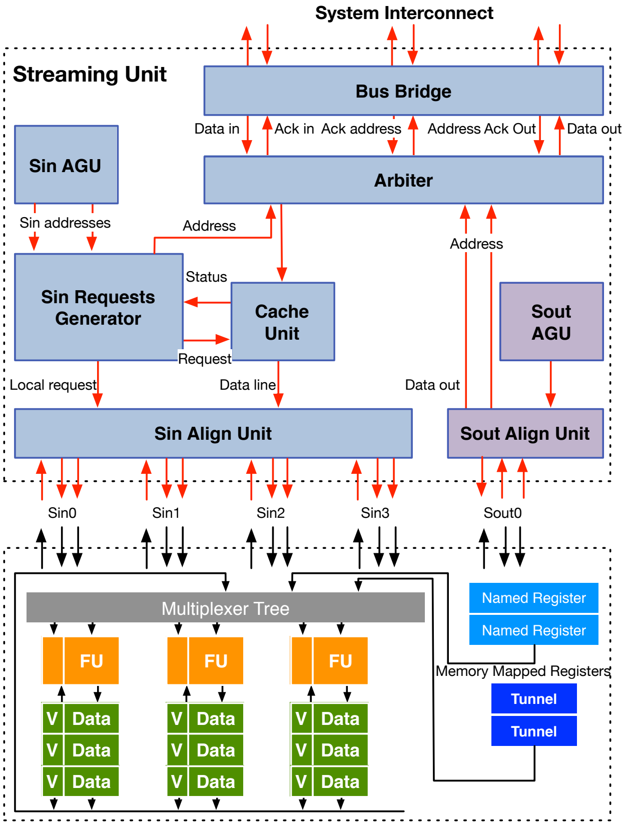


Figure 5.38: SOpenCL template architecture.

The template architecture accommodates a system interconnect—in reality a Xilinx *processor local bus* (PLB)—to handle all memory transactions from and to the main memory addressing space. Orchestration of data elements transactions is performed by the *address generation unit* (AGU) that generates addresses to prefetch data and delivers them to the address request module. The generation of the AGU is guided by identifying which code part is responsible for data I/O, and thus, the AGU provides addresses to all the elements within an input stream. The data requests generated then by the generator are coalesced by the Sin AGU to the width of the underlying system interconnect. There is competition from other stream units for the system interconnect to provide the required memory accesses. Thus, a cache unit allows to withhold using the system interconnect if data is available in it, improving the overall accelerator bandwidth and contributing to lowering the contention on the interconnect[201]. The caching system employed explores both temporal and spatial locality to reduce the latency of memory accesses. Its dual-port design allows the simultaneous serving of the arbiter and the Sin align unit. The alignment unit retrieves data either from the cache unit or incoming from the data channel, eliminates any existing gaps in order to optimize bandwidth and keeps them ordered in the datapath. As soon as enough data is ready to fully consume the bitwidth of the PLB connection, a write request is given to the arbiter. At this point, the stream units access to the PLB are regulated in round-robin scheduling.

5.5.4 Experimental results

The SOpenCL-based accelerator considers scenarios I and IV to evaluate how efficiently the wide-pipeline accelerator is generated[12]. Furthermore, the SOpenCL-generated Verilog project has been synthesized, and put through placement and routing on a Xilinx Virtex-6 LX760 FPGA device, using the Xilinx ISE 12.4 toolset. We consider the LDPC codes in scenarios I and IV (c.f Table 4.3) under SOpenCL accelerator generation.

In detail, the considered scenarios I and IV are expressed by OpenCL kernels defining one work-item per node in the Tanner graph, i.e., the `CN_kernel` and the `VN_kernel`, whose kernel containers are coded in Figure 5.17, spawn M and N work-items. Due to the front-end optimizations and the SMS back-end optimization carried out, the execution grid structure is replaced by a triple-nested loop structure, thus there will be M and N iterations scheduled for execution in the synthesized CN and VN pipelines. Thus, according to (5.6), the total latency taken by the wide-pipeline decoder in clock cycles as

$$t_{dec} = D_{CN} + M \times II_{CN} + D_{VN} + N \times II_{VN} \ (cycles), \tag{5.16}$$

with D_{CN} and D_{VN} the generated pipeline depths, and II_{CN} and II_{VN} the IIs achieved, respectively, for the CN and VN pipelines. By combining the clock frequency of operation f obtained by the placed and routed design we can write the total latency in seconds as

$$t_{dec} = \frac{D_{CN} + M \times II_{CN} + D_{VN} + N \times II_{VN}}{f} \ (s), \tag{5.17}$$

and finally, the obtained decoding throughput will come as

$$T_{dec} = \frac{Words \times N}{Iter \times t_{dec}} = \frac{Words \times N \times f}{Iter \times (D_{CN} + M \times II_{CN} + D_{VN} + N \times II_{VN})} \ (bit/s), \tag{5.18}$$

```
// SOpenCL CN kernel container
__kernel void CN_kernel(__global uint4 *Lq,      // L(q) messages
                        __global uint4 *Lr,      // L(r) messages
                        __global int *Lr_idx)    // Indexes to write L(r)
{
  int CN_id = get_global_id(0);

  CN_update(Lq,Lr,Lr_idx,CN_id);

}
```

```
// SOpenCL VN kernel container
__kernel void VN_kernel(__global uint4 *LQ,      // L(Q) messages
                        __global uint4 *Lq,      // L(q) messages
                        __global uint4 *Lr,      // L(r) messages
                        __global int *Lq_idx)    // Indexes to write L(q)
{
  int VN_id = get_global_id(0);

  VN_update(Lq,Lr,Lr_idx,CN_id);

}
```

Listing 5.17: SOpenCL CN and VN kernel containers.

where *Words* are the number of codewords concurrently decoded by the accelerator and *Iter* the number of issued decoding iterations.

Bitwidth and II Optimizations The bitwidth of operands that are mapped onto FPGA logic resources dictates some of the pressure of procured resources to place and route the synthesized LDPC decoder. To aid the SOpenCL tool perform this optimization better we explicitly slice the *most significant bit*s (MSBs) that are not utilized. Since OpenCL is based on the C89 programming language standard, there are no custom types for arbitrary integer or fixed-point precision. Instead, we have to rely on the conventional data types and bitwise mask to the desired bitwith. To this end, a parameter `b` is introduced in the kernel descriptions (illustrate in the code snippet in Figure 5.18) which sets the defined bitwidth of the LLR messages in the kernel. Also, a generic version has been synthesized which forgoes the bitwidth optimization step in the SOpenCL back-end.

```
int Lq;     // Container for 4 LLRs
uint4 Lq4; //Container for 16 LLRs

// Bitwidth defined as b={1,2,3,4,5,6,7,8} bits
int Lq0 = (Lq >> 24) & (1 << b);
int Lq1 = (Lq >> 16) & (1 << b);
int Lq2 = (Lq >>  8) & (1 << b);
int Lq3 = (Lq >>  0) & (1 << b);
```

Listing 5.18: Unpacking and bitwise operations in the SOpenCL kernels. Parameter `b` defines the bitwidth representing an individual LLR message that dictates how the bitwidth optimization is driven for the LLR arithmetic operations.

Table 5.10: SOpenCL CN pipeline for the considered scenario I LDPC kernel implementation and architectures configurations $II = \{1,2,8\}$, and varying bitwidth optimizations for $\{5,6,8\}$ bits and a Generic bit precision approach.

	Bitwidth	Slices	Flip-flops	LUTs	Freq. (MHz)	Latency (cycles)	Exec. Time (ms)
II=1	8 (no BW opt.)	12061	42718	39594	100	102	0.481020
	8	11600	41892	38759	101	102	0.476257
	6	11647	35948	33914	103	106	0.467049
	5	10369	33639	32861	107	106	0.449589
	Generic	24108	101960	80115	91	106	0.528637
II=2	8 (no BW opt.)	25453	64311	92096	88	103	0.546625
	8	21424	54872	81526	97	103	0.495907
	6	23632	61035	78884	95	110	0.506421
	5	19374	61052	65192	88	110	0.546705
	Generic	28432	67307	73212	63	110	0.763651
II=8	8 (no BW opt.)	33213	54749	78266	50	210	1.284200
	8	27556	57582	58788	53	210	1.211509
	6	27008	56745	64104	50	231	1.284620
	5	26894	54868	64083	51	231	1.259431
	Generic	36954	58121	79682	51	231	1.259431

Furthermore, to assess how the performance of the generated LDPC decoder is affected by varying the II and how efficiently are the FPGA logic elements utilized by pipelines with different IIs, this parameter was set to $II{=}\{1,2,8\}$ for optimization by the SMS step. The obtained results for scenario I are tabulated in Tables 5.10 and 5.11.

The analysis of the obtained results for the scenario I under consideration allows some insight into how hardware is generated and what are the most suitable combinations of design parameters. First, we analyze how the II affects the clock frequency of operation and the latency of the generated CN and VN pipeline datapaths.

Impact of II Optimizations Counter-intuitively, since a separate FU for each primitive instruction in the datapath is required, the design choice that minimizes the number of utilized logic resources is for $II{=}1$. However, since the kernel code is punctuated by simple arithmetic operations between a variable and a constant, the assignment of FU inputs to constants makes room for copious opportunities for the synthesis tool to reduce the number of utilized logic elements. On the other hand, as soon as a higher II is set ($II{>}1$), the FUs are driven by a multiplexer tree which prevents this optimization from happening. Furthermore, the multiplexer tree adds routing congestion, and, thus, the placer is not able to come up with a routing solution as good as the one available for when $II{=}1$. The clock frequency of operations are better for lower IIs, especially in the

Table 5.11: SOpenCL VN pipeline for the considered scenario I LDPC kernel implementation and architectures configurations $II = \{1, 2, 8\}$, and varying bitwidth optimizations for $\{5, 6, 8\}$ bits and a Generic bit precision approach.

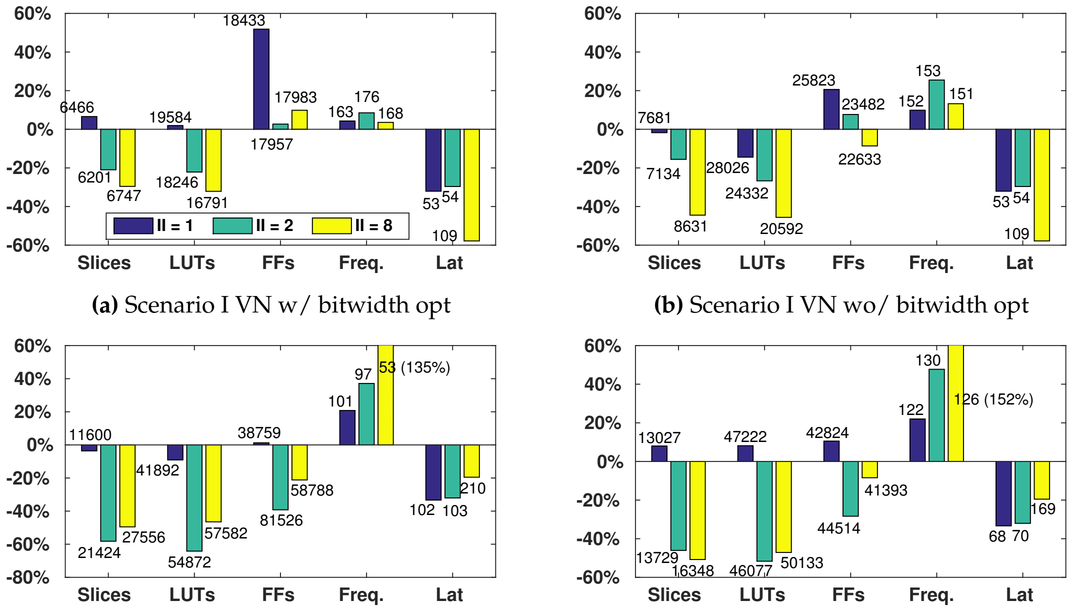
	Bitwidth	Slices	Flip-flops	LUTs	Freq. (MHz)	Latency (cycles)	Exec. Time (ms)
II=1	8 (no BW opt.)	7681	28026	25823	152	53	0.158243
	8	6466	19584	18433	163	53	0.147564
	6	5891	17746	17001	175	57	0.137469
	5	5515	16132	16509	182	57	0.132181
	Generic	10572	35865	37056	164	61	0.146713
II=2	8 (no BW opt.)	7134	24332	23482	153	54	0.157216
	8	6201	18246	17957	176	54	0.136670
	6	5996	17663	17385	171	58	0.140690
	5	5665	17269	17077	166	58	0.144928
	Generic	8226	27190	27891	164	62	0.146720
II=8	8 (no BW opt.)	8631	20592	22633	151	109	0.212642
	8	6747	16791	17983	168	109	0.191125
	6	7032	17524	18697	163	120	0.197055
	5	6731	17227	18384	172	120	0.186744
	Generic	9963	23946	26683	132	127	0.243386

CN datapath case, whereas in the VN there is no significant penalty incurred. Also, another interesting effect observed for both the CN and the VN datapaths, is that when $II > 2$ the insertion of pipeline registers in the multiplexer tree as a way to reduce the critical path delay and improve routing will lead to roughly a doubling of the pipeline latency when the II is eight clock cycles. This behavior is not observed, however, when II=2, as there are no registers introduced to the pipeline.

Thus, the highest II case is negatively affected in a three-fold. Not only is their *a-priori* performance worse than lower IIs due to lower rate of initiation of work-items in the pipeline (5.16), but also because the latency is negatively affected as well, and, finally, because the clock frequency of operation also degrades.

Impact of Bitwidth Optimizations Bitwidth optimization is a particularly subject in the reconfigurable computing field which is not matched by the OpenCL standard capability to define arbitrary precision variables. In fact, the supported C89 standard defines the conventional datatypes of the C programming language with the addition of several vector data types that are suitable for maximizing the bandwidth of memory transfers involving wide buses, as discussed in the previous chapter. In the considered scenario I, not only is the bitwidth defined by the bitwise mask (c.f the code snippet in Figure 5.18), but also several LLR messages, corresponding to several codewords, are loaded are required unpacking. The operations required to do so have an impact on the quality of the generated hardware. The packing and slicing operations are costly for II values greater than a single clock cycle. Not only is the multiplexer tree density increased, but also, there could be a need for more FUs, adding further multiplexer trees to the design. Again, counter-intuitively, the logic utilization is higher for smaller bitwidths. As a consequence, it is harder to efficiently place and route these design configurations.

Two other cases have been considered. When a generic bitwidth is applied, i.e., the bitwidth is not a known *a-priori* but depends on a runtime parameter, all FUs are polarized to the greater bitwidth involved (8 bits). This configuration is the most area demanding, especially when the IIs are low. This is due to the fact that all compile-time optimizations are not performed and that a tremendous number of masking operations will need to be applied in the datapath. Secondly, to illustrate the importance of this type of optimizations, they have been turned off for the best considered case which is the 8 bit decoder. Whereas the performance difference may not be so significant with regards to decoding latency, it is with regards to logic utilization, especially for higher values of the IIs in the CN and VN datapaths.



(a) Scenario I VN w/ bitwidth opt

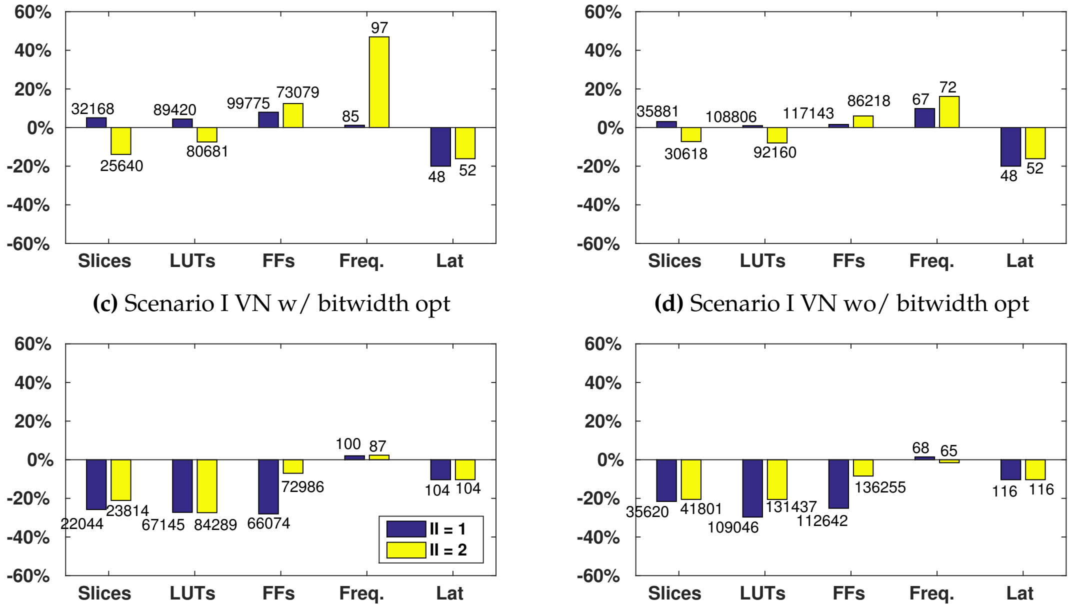
(b) Scenario I VN wo/ bitwidth opt

(c) Scenario I CN w/ bitwidth opt

(d) Scenario I CN wo/ bitwidth opt

Figure 5.39: Impact of instruction clustering on the logic utilization for the considered Scenario I a)–d) (lower is better for Slice, LUT and FF utilization, and Latency, higher is better for Frequency). The baseline comparison is to the case where no clustering optimization is performed. The bar label stands for absolute number of resources utilized by unoptimized decoder. On the left column the 8-bit decoder solutions are bitwidth optimized while on the right-hand column they are not.

Impact of Instruction Clustering Clustering of instructions across the DFG of both the CN and VN datapath is observed to be a powerful area utilization optimization tech-



(c) Scenario I VN w/ bitwidth opt

(d) Scenario I VN wo/ bitwidth opt

(e) Scenario I CN w/ bitwidth opt

(f) Scenario I CN wo/ bitwidth opt

Figure 5.40: Impact of instruction clustering on the logic utilization for the considered Scenario IV c)–f) (same legend as in Figure 5.39, only $II=\{1,2\}$).

nique. Herein, the scenarios assumed are I and II (c.f. Table 4.3) and the baseline version for comparison is the decoder version not performing instruction clustering, as assumed before. As seen in Figures 5.39 and 5.40, instruction clustering has a clear positive effect on the clock frequency of operation. On Scenario I, the VN kernels are the ones benefiting the most from this optimization, in particular for II=8 clock cycles, unfolding improvements of up to 152%. Furthermore, the computational performance of the generated pipeline is also improved by the fact that the latency is reduced by at least 40% for all Scenario I cases. In this case, VN kernels are the ones improving the most, again for II=8. Naturally, if the number of loop iterations scheduled for execution, which match the number of work-items in the OpenCL kernel description, is high, then latency improvement have a lower impact than higher clock frequency of operation of the pipeline.

For the considered Scenario IV case, a SOpenCL compiler limitation arises as the tool was not able to converge to a solution when instruction clustering optimizations are turned on. This is due to the irregular LDPC structure which adds further complexity to the OpenCL kernel descriptions. Herein, we observed highest improvements come for the VN kernels with regards to clock frequency of operation and not more than a 20% reduction in pipeline latency is observed. As the design grew more complex, we can see diminishing returns for instruction clustering optimizations measured in clock frequency of operation and pipeline latency. Roughly speaking, we can see the clock frequency of operation as a consequence of logic utilization, since lower levels of the latter typically imply less complex routing solutions, and, thus, the critical path is shorter allowing the former to increase.

In this regard, the impact of instruction clustering on logic utilization is extremely positive, with all solutions experiencing at least a 20% decrease in logic required to implement the considered scenarios. While instruction clustering requires more FFs, this does not lead to an overall greater utilization of slices. This stems from the fact that a single slice is composed of four LUTs and eight FFs, which tells us that a greater utilization of FFs within the utilized slices occurs. Because after placement and routing a greater number of FFs within a smaller number of slices are being utilized, this means that computation is spatially more contained within the FPGA, which in its turn leads to higher clock frequencies of operation, due to shorter critical paths. After all, spatially, computation occurs in a lessen area of the FPGA chip. Instruction clustering is not very efficient at reducing the logic utilization when the II is at its fastest rate because there are no multiplexers structures to optimize in this case. However, when the rate goes down, i.e. for $II=\{2,8\}$, plenty of opportunity exists to optimize these structures and, consequently, optimize each variable lifetime.

The aforementioned increase in FF utilization rate can be explained by the effect that instruction clustering has on chain patterns in the DFG it replaces by so-called *macro-FU*s (MFUs). Instead of requiring a single register prior the instruction, each variable will need as many registers as required to keep them and to retain the data from preceding and proceeding loop iterations in the MFU. In Figure 5.41, we can observe that the result of subtraction is required only at the final addition instructions. When no clustering is applied, the result in $N0$ is registered and then fed in to $N7$, while in the MFU stemming from the instruction clustering optimization, $N0$ whose output is available at $T=0$ will need four registers to be kept alive until this result is consumed at $T=4$.

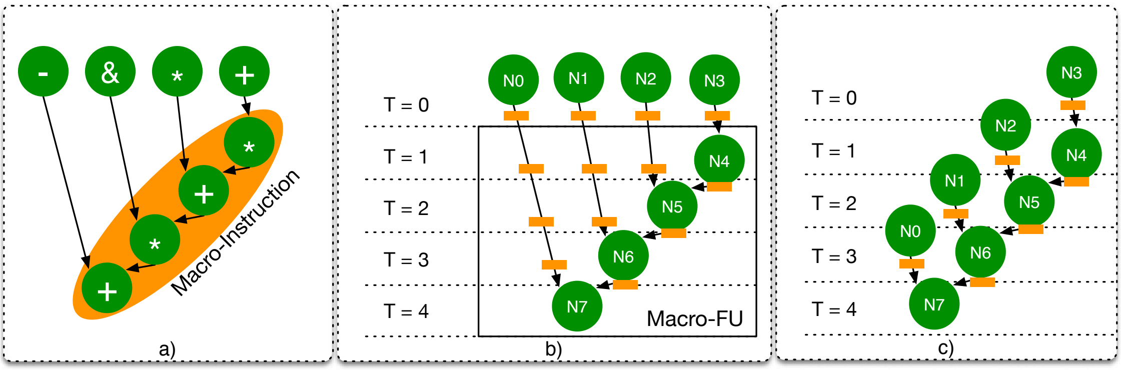


Figure 5.41: Instruction clustering effect on the lifetime of inputs. In *a)* a partial DFG is optimized taking into account the highlighted macro-instruction. In *b)* the scheduler output and FU implementation with the macro-instruction is shown. In *c)* the equivalent non-clustered version schedule is shown.

Generated Datapath and Memory System The SOpenCL generates a functional datapath as wide-pipeline to each kernel given as an input, to which a memory system is hooked that manages traffic between the FPGA and the host system via a PCIe interface[201,254]. The generated memory system is illustrated in Figure 5.42, and can assume one of two configurations. In detail, the memory system handles all data transfers between the accelerator, resident in the FPGA fabric and the host computer via a PCIe interface. The accesses to the host are directed to its RAM, while FPGA memory accesses are directed to the FPGA BRAMs. The benchmarked system provides a PCIe v2.1 8x interface, thus, it offers a bandwidth of 4000 MB/s in either host to FPGA or FPGA to host directions. In the FPGA accelerator, the memory system should be able to deliver all required bandwidth to the generated datapaths, otherwise, computation must be stalled

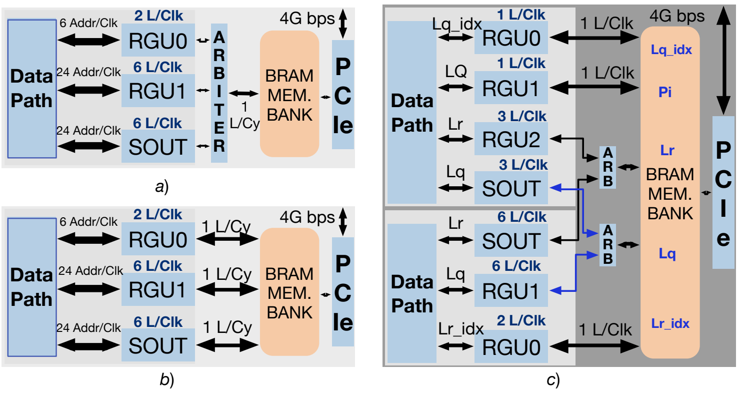


Figure 5.42: Block diagram of the SOpenCL-generated LDPC decoder showing the CN and VN kernels *a)* connecting via a single BRAM port, *b)* via three ports, one per I/O stream, and *c)* both the CN and the VN kernels are shown instantiated and interconnected.

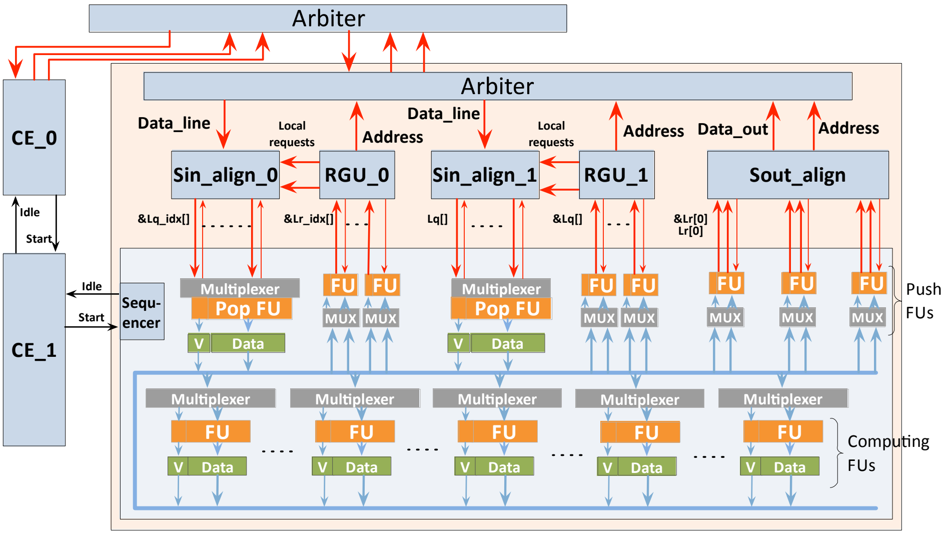


Figure 5.43: SOpenCL generated CN datapath. A custom number of FUs can be instantiated, based on which the generation of the *Sin_align* and the *RGU* units is driven to provide the correct servicing of data.

so that the loading and storing of the required data elements completes. We limit the analysis of the required bandwidth to the case where the II in the wide-pipeline is at the fastest rate (II=1) since it is the design producing and consuming data at the fastest pace, it is the one where it is most pressing to keep with its bandwidth requirements. Furthermore, we can restrict our analysis to the kernel with the most complex datapath, in this case the CN kernel datapath, seen in Figure 5.43. At this configuration, the CN datapath pipeline, when fully occupied, keeps 106 loop iterations active whose computation is active throughout 392 adders, 210 shifters, 369 logic units, 434 comparators, and 994 1-bit predicate manipulation units. Such datapath consumes 120 bytes of data every clock cycle and produces 96 clock cycles to the memory system. To try and meet this bandwidth requirements, the SOpenCL tool provides two memory sub-systems types that connect the generated datapaths to the accelerator memory space defined in the FPGA BRAMs.

The SOpenCL accelerator utilizes as many BRAMs units required to allocate the desired memory space in a concatenated fashion. This space can then be accessed in one of two ways. Illustrated in Figure 5.42a), a single-port to the BRAM memories is defined, while in Figure 5.42b), the system is implemented as a distributed memory system. The necessary memory access rate required by the CN accelerator is also illustrated in the Figure and it requires for the II=1 accelerator the following accesses in parallel. 6 addresses/cycle (A/Clk) for indexing of the Tanner graph (`Lq_idx` and `Lr_idx`), 24 A/Clk for the $L(q)$ LLR messages (`Lq`) and $L(q)$ (`Lr`). The *request generator unit* (RGU) and Sout Align blocks coalesce the addresses into 2 lines/cycle (L/Clk), 6 L/Clk and 6 L/Clk, respectively, utilizing a 128-bit data bus. The bus width is a consequence of the packed 16 codewords scheme utilized, making each LLR datatype to contain 16 messages with 8-bits. Due to the 128-bit single bus, data is stalled for 14 cycles in teh pipeline for each computation address generation cycle. However, in the distributed memory configuration, one 128-bit bus is assigned to each RGU and Sout Align modules, making the stall time to be shortened to 6∼14 cycles. Conditions for preventing the stalling in the pipeline unfold a need for 768-bit bus for the handling LLR messages data requests and a 512-bit bus for handling those of the indexing of the Tanner graph.

Clearly, the most suitable configuration is the distributed one as it provides more bandwidth to fulfill the generated wide-pipeline datapath requirements. Another feature is that due to the distributed nature of the FPGA BRAM units in the chip is more suitable than the single-port memory bank which would restrict bandwidth and draw extra stall cycles. The distributed memory system, interconnected via appropriate data arbiters to the wide-pipeline datapaths, is shown connected to the CN and the VN pipeline datapaths in Figure 5.42.

5.5.5 Operational Transform FFT/FWHT

In Chapter 2 we have discussed the role of the FFT applied to non-binary LDPC and in Chapter 4 we have discussed the constraints that need to be overcome for the realization of fast and efficient FWHT computation in programmable GPU architectures[13]. Herein, we discuss the particular case of the FFT acceleration on reconfigurable devices using a wide-pipeline approach. We do so based on the following assumptions. Of all the supported compiler infrastructure, the wide-pipeline OpenCL tools available, allow for the generation of a complete architecture, i.e., not only is the accelerator generated, but the remaining modules, such as clock, PCIe interface and respective memory interconnects, are also instantiated, making it available for use in a similar fashion that a GPU accelerated FFT would be. Secondly, a conservative throughput estimate for the FWHT, which is of interest to the work developed in this Thesis for the Fourier-based non-binary LDPC decoding, can be inferred from the FFT accelerator. Furthermore, literature covers the FFT extensively[238] for programmable computer architectures such as CPUs[281] and GPUs[282], and also in reconfigurable computing devices[283–285]. However, in the latter case, majority of the surveyed approaches use RTL-based projects to realize the FFT accelerator or specialized, and often proprietary and closed-source, IP cores, whereas, in our case, we are interested in exploring how an HLS-based model can be employed, namely in what ways the challenges of devising an efficient accelerator can be overcome. To this end, we explore the Altera OpenCL compiler to generate a wide-pipeline accelerator.

OpenCL FFT for Reconfigurable Devices The computation of the *discrete Fourier transform* (DFT) through the FFT algorithm allows for a a numerical complexity initially scaling with the transform length N of $O(N^2)$ to $O(N\log_2(N))$ while allowing for better numerical accuracy[238]. Despite this less demanding complexity when compared to the DFT case, the FFT still performs a high volume of memory and arithmetic operations, and usually very high throughputs are required, as for instance, in the non-binary LDPC case or other communication systems[13]. The need for high number of arithmetic resources and high throughputs makes the development of the FFT particularly challenging. Furthermore, due to the several radix combination or configurations available for implementation for each N-length transform FFT, further adds to the design space complexity. Compounded by the fact that data elements in the intermediate stages of computation will flow through memory locations available to each work-item within a work-group, then our design must be able to overcome the fact that work-items have been initiated in the wide-pipeline accelerator at a rate that may produce a considerable time lag to elements produced by the preceding and the trailing work-items, which are then consumed in reversed order. In other words, efficient working out the required synchronization mechanism to share data among work-items will determine the efficiency of the developed accelerator. This challenge is illustrated in Figure 5.44.

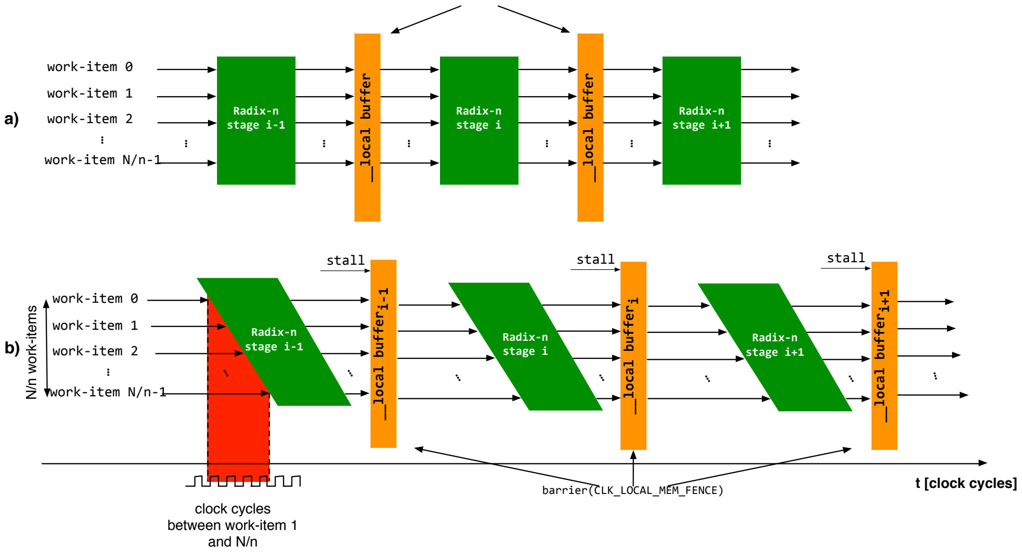


Figure 5.44: OpenCL workgroup execution flow of the wide-pipeline N-length FFT assuming an arbitrary radix-n implementation. *a)* compares the expected flow for a SIMT execution model. In that case, synchronization mechanism issued by the `barrier(CLK_LOCAL_MEM_FENCE)` can be issued in parallel for all work-items within a work-group. However, for the wide-pipeline in *b)* the work-item 0 lies at least $N/n - 1$ clock cycles ahead of the trailing work-item, and, thus, the synchronization mechanisms stalls the pipeline completely throughput a number of cycles.

Mapping the Design Space to Circuits The instructed data granularity of the FFT is towards the finest granularity possible for a particular radix-n implementation, which in the past lead to significant performance boosts for programmable GPU devices[239]. Stretching the OpenCL cross-platform capabilities one step further, we take FFT OpenCL-descriptions that based on the literature will yield high FFT throughputs. However, the underlying SIMT execution model, whose features are well captured by the OpenCL programming model, assumes in the best case scenario that all work-items are running in parallel, as in Figure 5.44a). In this case, the data exchange happening between work-items, after each radix-n stage of the FFT, can be performed concurrently, or at least concurrently at the warp or wavefront granularity[188,221]. This is certainly not the case of the OpenCL wide-pipeline accelerator seen in Figure 5.44b). In that case, as work-items are initiated at a given rate in the wide-pipeline accelerator, their execution is offset in time by the difference between their numerical identifier times the II. As a consequence, the execution of several synchronization points in the design requires that the work-items execution is stalled. In the naïve-most case, all work-items are stalled until the trailing work-item reaches the synchronization point in the pipeline, after which work-items are again re-initiated through the remaining stages of the pipeline. In essence, a procedure much alike the loop fission defined in SOpenCL compiler[201,254]. However, a closer inspection of the memory access patterns between the several radix-n stages reveals that in most cases, the worse case waiting time, where work-item 0 is stalled until the last work-item has not reached the synchronization point, does not occur. Provided the compiler tool has the ability to do so, then the penalty incurred by each synchronization operation does not have to be necessarily the number of work-items involved in the computation of the FFT factored in by the II rate.

The proposed FFT accelerator is not based on a particular combination of radix-n factorization units which direct data between each other based on the transform length N to be computed. Instead, because of the OpenCL description, we proposed several configurations which are shown to perform well for SIMT-based architectures[13,239]. Under this configuration a particular length N will see the initiation of N/n work-items onto its FFT wide-pipeline accelerator. The rationale of the keeping of data granularity at fine levels, where each work-item is responsible for serving the input to each radix factorization block, prevents additional synchronization concerns with elements being fed as inputs to the different radix stages.

```
__kernel
__attribute ((num_simd_work_items(SIMD_LEVEL)))
__attribute ((num_compute_units(NO_CUS)))
__attribute ((reqd_work_group_size(WORKGROUP_SIZE,1,1)))
void fft_1024(__global float2 *in, __global float2 *out,__constant *twiddle){
  __local float lMem[1040], *lMemStore, *lMemLoad;
  float2 buffer16[16];

  loadData(in, buffer16);                //load 16 elements to buffer

  radix16(buffer16, twiddle);            //execute stage 0 radix-16 computation

  storeData(buffer16, lMemStore, stage00_idx); barrier(CLK_LOCAL_MEM_FENCE);
  loadData(buffer16, lMemLoad, stage01_idx); barrier(CLK_LOCAL_MEM_FENCE);

  radix16(buffer16+0, twiddle);          //perform stage 1 radix-16 computation

  storeData(buffer16, lMemStore, stage10_idx);  //swap data
  barrier(CLK_LOCAL_MEM_FENCE);
  loadData(buffer16, lMemLoad, stage11_idx);
  barrier(CLK_LOCAL_MEM_FENCE);

  //perform stage 2 4x radix-4 computation
  radix4(buffer16+0, twiddle); radix4(buffer16+4, twiddle);
  radix4(buffer16+8, twiddle); radix4(buffer16+12, twiddle);

  storeData(out, buffer16);              //store data back to memory
}
```

Listing 5.19: OpenCL kernel for the N=1024-point FFT. The employed `float2` stores the real part in component `x` and the imaginary part in component `y`. The n-way SIMD level is controlled by the `num_simd_work_items` directive, the number of CUs by `num_compute_units`, and the generated hardware is optimized for workgroups with a certain size through the `reqd_work_group_size`.

The employed method uses *decimation in time* (DIT) in order to divide two of the most critical operations performed, data shuffled between work-items and multiplication of data elements by their corresponding twiddle factors. Hence, first we assign data to the correct work-item. During initialization and termination of the kernel, this is an in-order procedure. Each work-item loads as many data elements as required by the first stage of the radix factorization. In the intermediate stages, where data elements must be coherently exchanged between work-items, a synchronization point issued by a barrier instruction enforces this coherence and work-items proceed to compute the butterfly stage ahead. Afterwards, due to DIT, the second-most complex operation is issued. Each work-item computes the set of twiddle factors by which the produced data elements are multiplied, or loads them from a ROM containing a set of precomputed twiddle factors.

Performing computation of twiddle factors on-the-fly allows the designer to prevent routing issues ensuing from the utilization of memory blocks that are distributed across the FPGA and are not necessarily close to the location where computation is defined on the FPGA. However, on-the-fly computation entails wasted bandwidth as twiddle factors are repeatedly computed and are not re-utilized when several transform batches are to be computed. Furthermore, since the twiddle factor is computed as $W_N^n{=}e^{\frac{-j2\pi n}{N}}$, its hardware implementation for an unknown transform size N entails the computation of $\sin(\cdot)$ and $\cos(\cdot)$ values, an operation most likely performed using DSP, a scarce resource on the FPGA, whose high utilization may incur severe routing issues as it requires DSPs potentially far from where computation spatially occurs. As a consequence, and considering the that acceleration via an OpenCL-based wide-pipeline requires integration of the FPGA accelerator onto a host computer system, computation of the twiddle factors can be performed offline by the host system, where sin and cos operations are virtually free (with regards to its influence on the performance attained by the FPGA accelerator), together with the remaining data management housekeeping tasks. As mentioned, the downside is the utilization of ROMs blocks, for synthesis of the constant memory space, that are not spatially close to logic units where computation occurs, leading to routing problems. The OpenCL kernel container for the FFT accelerator is shown in Listing 5.19 for the $N{=}1024$-point FFT, and it exemplifies the execution flow taken for the developed FFT flow.

In addition to the aforementioned decisions made regarding the design space exploration of the FFT on a wide-pipeline accelerator, other hardware generation parameters are explored.

i) number of CUs in the wide-pipeline accelerator;

ii) vectorized processing within each CU through $\{2, 4, 8, 16\}$-way SIMD;

iii) distinct DRAM bank to maximize global bandwidth for out-of-place computation;

iv) explicit BRAM usage is limited to local memory.

The combination of *i)* and *ii)* effectively lowers the II to less than a clock cycle per work-item. On the one hand, inclusion of multiple CUs per wide-pipeline divides the work-items schedule for execution between each CU. On the other, vectorized processing makes each work-item to be initiated simultaneously with k others for a k-way SIMD processing[177]. The main driving limitation of increasing the parallelism level through the generation of several CUs per pipeline and defining SIMD processing is the overhead in logic elements required. Also, if the FFT computation is performed out-of-place, i.e., the transform is stored in different memory location than the input signal, the input and the output memory arrays can be divided into separate DRAM memory banks. For a low number of FFT batches, the DRAM might not be accessing the input buffer at the same time it accesses the output buffer, and thus DRAM bandwidth can be fully allocated to each access. However, if a high number of batches are to be computed, both buffers will be accessed at the same time, thus data requests will race for memory bandwidth. Given the existence of only two buffers in the global memory region that are mapped to DRAM, we can explicitly define that the input buffer be allocated in a separate physical memory bank than the output buffer[177]. Naturally, for an in-place FFT wide-pipeline accelerator, *iii)* is not feasible. Finally, we are limited by the mapping of the OpenCL memory model to the physical FPGA resources. A more efficient usage of the FPGA BRAM is not possible because of the variables lifetime when they are defined as local memory, as opposed to their lifetime when allocated as global memory. Due to granularity involved, assigning the computation of one transform batch per work-group means that even if data is moved ahead of computation to the local memory, its lifetime span under the OpenCL programming model does not withstand the change of workgroup instantiated in the wide-pipeline. In other words, all fetching of data inside the kernel is available within the time and spatially boundaries of a workgroup.

Wide-pipeline FFT A driving design goal of the FFT accelerators herein presented was to preserve a true cross-platform capability. To that end, the wide-pipeline FFT herein described was optimized for FPGA execution based on a high speed GPU FFT kernel.

Despite the pipelined processing of work-items that compose the kernel execution grid, there is no pipelined processing of distinct execution grids. As a consequence, in order to keep the execution flow fully pipelined, all stages composing the radix-n DIT factorization of the N-point DFT are packed into the same kernel[273]. However, this objective is unattainable as between stages data must be exchanged between the executing work-items through the *local* memory space, a process requiring barriers that stall the work-items execution. This is shown in Figure 5.44b), for an II of one clock cycle. Depending on the DFT length N and on the stage computed, each work-item in the pipeline is stalled so that it can consume data produced later by a trailing work-item. However, this is preferable to the alternative of non-pipelined execution of OpenCL kernels for each stage, constrained to the use of *global* memory only where the whole execution grid must be flushed first so that the next stage may execute.

Moreover, whenever the same *local* buffer memory location is utilized in different stages of the kernel, which in the FFT case means as many times as there are stages minus one, there is a time growth of the wide-pipeline critical path. We have tested two ways to avoid this are: the use of circular buffering so that the *local* memory buffer used prior to a stage is always different from the one used after it (only two are necessary); or we can define *local* memory buffers that are used only once. Both strategies are possible due to the availability of a large number of BRAM blocks, whereas in multicores *local* memory is typically limited to 48KB.

5.5.6 Experimental results

Herein, we discuss the performance of the synthesized FFT accelerators using the wide-pipeline accelerator approach using the Altera OpenCL compiler in an identical experimental apparatus as in Subsection 5.5.2. Due to the focus of this work in LDPC decoding, the design space features tested, have been applied to the 256-point FFT, which is of particular interest in the case of FFT-SPA decoding of non-binary LDPC codes, due to its excelling performance[192]. The synthesized accelerators performance is tabuled in Table 5.12.

Table 5.12: Experimental results for $N = \{256, 1024\}$ radix-4 FFT algorithms run on FPGA, CPU and GPU. The throughput is expressed in mega work-item/s and in mega FFTs/s (M/s).

Kernel	N	Radix	SIMD/ /CUS	Tot. Logic (%)	Registers (%)	BRAMs (%)	DSPs (%)	Op. Freq. (MHz)	Mwork-item/s	MFFTs/s
Single buffer	256	4	1/1	71	31	43	12	215.14	108.95	1.38
Circular buffer				71	31	43	12	193.38	141.12	1.27
Dedicated buffer				70	31	43	12	226.75	172.23	1.38
Dedicated buffer pre-computed W_N				60	26	43	3	193.34	172.23	1.28
Same as upper w/ manual SIMD of 2 FFTs				85	36	53	5	160.90	136.87	2.13
Altera SDK[177]	256	4		76	41	35	6	212.13	190.00	3.99
	1024	4		87	40	45	8	204.7	190.00	1.27

The first observation to be held is the inability of the OpenCL compiler to drive a number of CUs greater than one, or for that matter, to vectorize the work-items processing in a SIMD execution. This is justified by an over-conservative policy of the compiler regarding utilized logic. The first estimate, obtained by the C-synthesis process is based on the overall sum of logic elements that each primitive consumes, and thus overestimates the logic resources actually need. After the synthesis process, the generated netlist sees shared utilization of logic elements by the different primitives, and, thus, logic utilization is actually lower than the first estimate. However, it is the overestimated utilization logic levels that defines whether synthesis will go through or not. Under our considered designs, we might have been able to drive the number of CUs or the SIMD levels to define IIs greater than a single clock cycle, since for the 1 CU and 1-way SIMD we could get utilization levels under 70%.

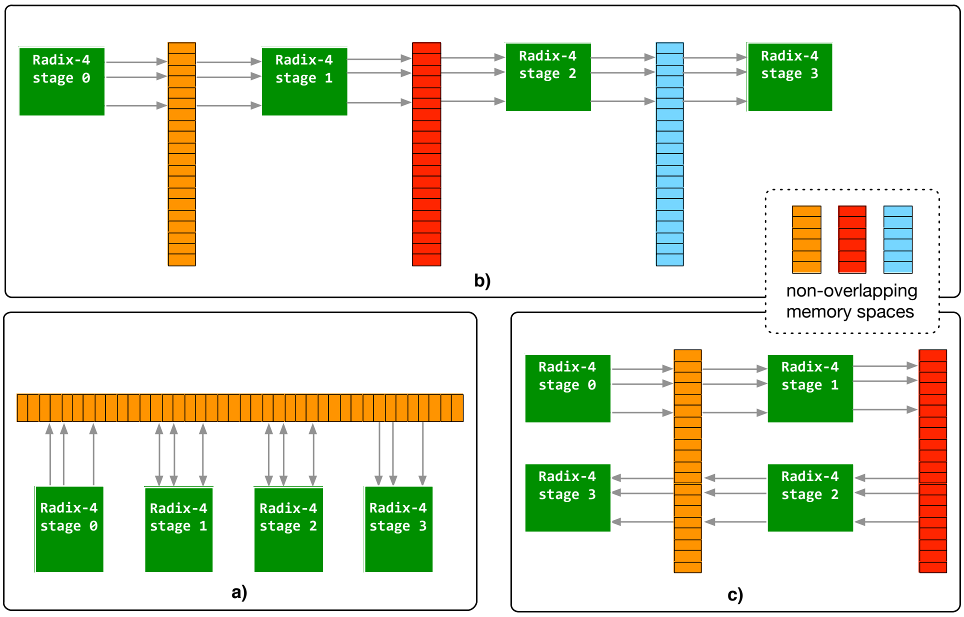


Figure 5.45: OpenCL FFT local memory buffer schemes employed: *a)* single buffer scheme where each stage produces data to the same data locations; *b)* dedicated buffer scheme, where each stage produces data to stage-exclusive data locations; and *c)* circular buffer scheme, where two buffers are used in rotation mode.

An interesting observation is the logic utilization of designs that utilize twice or three times more local memory space than what is utilized by the single buffer scheme. There is no variation on the logic utilization of the dedicated buffer scheme design when compared to the single buffer scheme. The only noticeable difference is when twiddle factors are offline computed by the host and are loaded from the constant memory space. In this case, the total logic utilization is driven down from 70% to 60%, with a four-fold decrease in DSP utilization as the computation of the sin and cos operations are no longer required. Given the slack obtained in utilized logic, we potentially define a SIMD level of 2-way, since each work-item in addition to its data elements is also given an extra batch to compute. This strategy proved feasible, with logic driven up to 85% utilization levels.

Clock frequency of operation undergoes a variation process with the buffer scheme employed, the need to compute twiddle factors and the SIMD level in the accelerator. The highest frequencies have been obtained for the single buffer and the dedicated buffer schemes. However, this does not necessarily translate to a higher throughput, both in work-items dispatched per second or in computed transforms per second, which is discussed below taking the derate factors into consideration (c.f. Table 5.13). For the offline computation of the twiddle factors, the same work-item dispatch rate is obtained, even for a clock frequency 33 MHz lower. Furthermore, clock frequency drops another 30 MHz for the case where two batches are computed per work-group, further lowering the number of work-items dispatched. Notwithstanding, the latter has the greater throughput measured in computed transforms per second, for the tested 1000 FFT batches workload, roughly 50% under the Altera OpenCL *software development kit* (SDK) version[177] throughput.

Derate Driving Factors The other main factors driving down the obtained performance, can be expressed in terms of derate factors, that lower the work-item initiation rate in the pipeline. In other words, while the II stays the same, at 1 work-item per clock cycle in the 1/1 (CU/SIMD level) configuration, the clock frequency of operation is lowered from the targeted 250 MHz to one factoring in the derate factors. 1) Control flow overhead in the FFT is limited, since there is not divergent branches in the datapath, and as such, it does not pose any issue, since it is 1 across all the targeted designs, including the ones provided in the Altera OpenCL SDK which we also benchmark. 2) Global memory bandwidth does not seem to come to play for our designed out-of-place FFT computation, since two different memory banks are assigned to each global memory array, i.e., input and output sit in physically different memory banks. This is not the case for the compared SDK versions, which are derated by a 0.92 factor. On the other hand, 3) the latter show no performance penalty on account of routing, whereas our designs are derated by 0.56∼0.68. As expected, when a single buffer scheme is employed, the tool has the ability to utilize resources that are spatially close. On the other hand, when two or more buffers need allocation routing issues arise given a not so spatially close location of the employed BRAMs to produce the local memory. Naturally, this could be avoided by synthesizing the local memory space onto LUTs. However, while this strategy is available in other HLS tools, it is absent from the Altera OpenCL compiler[177]. Finally, 4) while at a first glance the derate factor brought on by the stalling in the pipeline might be an indication of how well the code describes the algorithm for OpenCL wide-pipeline execution it must be interpreted taking other aspects in light. In the single buffer scheme case, the derate by pipeline stalls is 0.44, lowering the initiation rate of work-items in the pipeline to under half. To any degree are we able to determine whether this is due to work-items starving for bandwidth to local memory, or because, indeed, work-items are stalled in the synchronization points. Notwithstanding, the circular buffer and dedicated buffer schemes shed some light into why there are not stalls in the pipeline (their derate factors are 1 for pipeline stalling). Since connectivity issues prevent work-items from faster operation, there is no longer the need to stall the wide-pipeline. The total derate factors weigh 0.44∼0.68 in our designs.

Table 5.13: OpenCL FFT accelerator throughput derate factors: control flow overhead, lack of global memory bandwidth, connectivity issues, and pipeline stalls.

Buffer Scheme	N	Radix	SIMD/	Derate factors				
			/#CUs	Control Flow	Global mem. bandwidth	Connectivity (Routing)	Pipeline stalls	Total
Single buffer	256	4	1/1	1	1	1	0.44	0.44
Circular buffer	256	4	1/1	1	1	0.56	1	0.56
Dedicated buffer	256	4	1/1	1	1	0.68	1	0.68
Dedicated buffer pre-comp. twiddle fact.	256	4	1/1	1	1	0.68	1	0.68
Altera[177]	256	4	1/1	1	0.92	1	0.81	0.75
	1024	4	1/1	1	0.92	1	0.81	0.75

Comparison to CPU and GPU Libraries For the sake of comparison with CPU and GPU architectures, the 256-point FFT has also been benchmarked using the *fastest Fourier transform in the West* (FFTW) and the *CUDA FFT* (CUFFT) libraries, respectively. As observed in Table 5.14, the throughputs observed for the FFTW and CUFFT benchmarks are in line with the obtained FFT OpenCL accelerator on the FPGA. As a consequence, the power efficiency of the proposed methodology yields higher gains than those obtained by programmable architectures.

Table 5.14: FFTW and cuFFT libraries FFT performance for the 256-point FFT.

Kernel	N	Radix	Architecture	Op. Freq. (MHz)	MFFTs/s
CUFFT[282]	256	N/A	GPU	1.058	5.36
	1024	N/A			2.75
FFTW[281]	256	N/A	CPU	3.07	0.31
	1024	N/A			

Summary The methodology behind the proposed FFT accelerator was based on a strategy that is analogous to the one followed by the FWHT GPU kernel. While in the GPU engine the shared memory, herein local memory, poses the key challenge to attaining high throughput performance in the FPGA case it is not necessarily so. In fact, comparison with the radix-2 factorization FFT included in the Altera OpenCL SDK we can observe that forgoing its use bears more fruits than using it. In its core, this symbolizes that the potential of using OpenCL to drive the generation of hardware accelerators cannot be argued, but improvements are still required to address ways that allow the designer to configure a memory space without having the flexibility provided by the FPGA BRAMs (local memory) withhold. Despite this constraint, competitive FFT hardware accelerators can be generated based on the OpenCL programming model that achieve performances close to other FPGA-based alternatives.

5.6 Summary

In this chapter we discussed the different methodologies that allow us to obtain efficient LDPC decoder designs using distinct architectures, through different HLS tools, on reconfigurable computing devices[c)] In particular, we have shown how the dataflow approach is the most suitable one for obtaining very high decoding throughputs and low latencies, although at the cost of an increased development effort, with the designer having to focus on the balancing of the computation and data management through the kernel and manager concepts[6,10]. Through the use of a directive-based (annotations) HLS model, we proposed an isomorphic mapping of the Factor graph to hardware and how to direct the optimization space within that type of model[8,9]. Moreover, we propose an efficient deeply-pipelined method to extract high performance from the wide-pipeline decoder architecture, by introducing a time dependency (the corresponding decoding iteration) to the execution grid. By spawning work-items corresponding to different iterations, and by remapping the nodes of the Tanner graph appropriately, we have been able to push fourfold improvements under this approach when compared to a multi-kernel one[7,11]. Also, we have analyzed which contributions can be brought towards HLS compiler tools, through the evaluation of instruction clustering and bitwidth optimizations under the SOpenCL model[12]. Finally, we discuss how to realize efficient FFT accelerators for wide-pipeline approaches[14]. The pitfalls of an approach remotely close to that pursued for programmable hardware reveals that under the allure of the cross-platform capabilities of OpenCL lies a potentially for under-performing design. The roofline analysis pursued has made a connection between SNR channel conditions and with the bandwidth of the LDPC decoder, thus bounding it between memory and computing boundaries[6].

[c)] The work presented in this Chapter was communicated in a journal paper and in international conference[6–12,14]. In addition, the Altera OpenCL non-binary wide-pipeline decoder was submitted under the project "ONOFF: OpenCL-based Non-Binary LDPC Decoding for FPGAs" to the *Innovate Europe Contest 2012-2013* co-organized by Altera and the CNFM and was awarded the "Most commercially relevant application of an FPGA" prize.

6

Power-aware LDPC Decoders

Contents

The pursue of power and energy-efficient computer accelerators has been an ever driving design goal with researchers tackling how to make the most out of as many silicon devices could be crammed onto a chip. Naturally, challenges for energy-efficient computation on a *graphics processing unit* (GPU) processor are substantially different than those arising from a dedicated accelerator developed only for a particular task. With the number of transistors on a chip reaching the billion count, the majority of them cannot be active at the same time as it is impossible to dissipate all the heat produced[286]. Also, it has become a requirement for several accelerators that they be powered by batteries and operated in hand-held devices, thus greatly tightening the power budget available for a wide range of accelerators.

Researchers have tackled these challenges using multiple techniques, generally designated as dark-silicon, involving the turning on and off of idle components, or by keeping unused components in low-powered states, based on clock and voltage scaling, or *approximate computing* designs[286], whereupon strictly reliable computation is relaxed in favor of cheaper, and lower power components, but with a cost of introducing computation errors. Furthermore, optimization strategies favoring fast computation on fixed instruction set architectures, usually also optimize power-efficiency as the same algorithm is computed in less time, making the most out of the fixed set of logic resources available. In this case, the challenge of developing energy efficient systems is tackled most at the algorithmic level, whereas in the former methods, it is tackled at the system- or at the silicon-level (c.f. Figure 6.1).

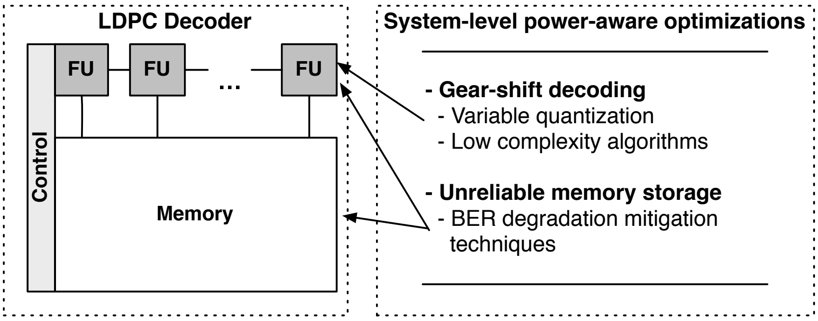

LDPC Decoder
FU
FU
...
FU
Control
Memory
System-level power-aware optimizations
- Gear-shift decoding
- Variable quantization
- Low complexity algorithms
- Unreliable memory storage
- BER degradation mitigation techniques

Figure 6.1: Power-aware optimizations introduced at the system-level. gear-shift decoding is an optimization to the FU operation. Unreliable memory storage lowers the energy requirements of the memory block while at the same time introducing errors that can be tackled at the FU- or memory-level accordingly.

If we consider that in digital circuits the switching power dissipation P is expressed by

$$P = \alpha \cdot C \cdot V_{DD}^2 \cdot f, \tag{6.1}$$

with α the activity factor, C the effective capacitance, V_{DD} the supply voltage and f the clock frequency of operation, designers can tackle the switching power dissipated at several levels. Algorithmic optimizations mainly target the number of arithmetic instructions per second or make use of data reuse, thereby reducing the activity due to memory accesses, minimizing α and f[287]. On the other hand, the quadratic dependency of the dissipated power with V_{DD} makes it highly appealing to try and lower it. However, aggressive voltage scaling will stumble upon the reliability wall, whereupon fully 100% reliable operation can no longer be guaranteed as components become electronically unstable. For instance, *static RAM* (SRAM), the most widely employed memory technology in dedicated accelerators, sees bit-cells no longer retaining data correctly. In addition, *embedded dynamic RAM* (eDRAM) technology, based on its density and low power requirements, has been gaining traction[288], and although it operates under lower V_{DD} levels, the limited *data retention time* (DRT) of the bit-cells requires refresh procedures for 100% reliable operation, in its turn increasing power consumption. Hence, the reliability wall can be stumbled upon by minimizing f on the memory accesses for eDRAM technology, such that refreshing not longer meets the minimum DRT of all bit-cells.

In this chapter we explore two approaches for the reduction of energy consumption motivated by power efficiency optimization techniques applied at the system-level[289–291]. Thus, they are suitable for dedicated *low-density parity-check* (LDPC) decoders. The first approach discusses gear-shift methods that can be introduced in the LDPC decoding algorithm in order to reduce the amount of computation, largely by the introduction of further sub-optimality, and therefore minimizing the energy required to successfully decode. gear-shift decoders balance a tradeoff between numerical complexity and affordable *bit error rate* (BER) losses[292,293]. The other method, weighs in on the fact that dedicated decoders are essentially composed of memory units. Thus, aggressive voltage-scaling techniques, including sub-threshold operation of the silicon, introduce unreliability as data is stored but cannot be ensured that it will be correctly retrieved[181]. Due to the errors introduced by the unreliable memory storage, suitable techniques are required to mitigate the ensuing BER degradation, and can be applied at the algorithmic-[294], system-[16] or at silicon-level[290].

6.1 Gear-Shift LDPC Decoders

Although the majority of the power budget of LDPC decoders is allocated to the memory block, and thus, alternative design techniques that minimize the energy consumed by the memory have the highest impact—including the aforementioned unreliable memory storage—power savings are also procured in the FUs at the system-level. In fact, this is one of the main motivations that lead to the introduction of the sub-optimal decoding algorithms described in Chapter 2. This includes algorithm modifications at the analytical-level—the *min-sum algorithm* (MSA) *check node* (CN) processing equations obtained from the *logarithmic sum-product algorithm* (LSPA), or the different formulations of the LSPA algorithm[262]—and also at the system implementation and optimizations therein—such as 1-D to 2-D indexed *lookup-table* (LUT)-based schemes[79,262].

With a design simplicity motivation in mind, most LDPC decoders perform the decoding of received words using a single algorithm. However, as observed by Gallager[21], binary message-passing decoders benefit from changing the decision threshold during the decoding process, with the performance and convergence threshold improving significantly. This concept is explored in so-called Gallager-B algorithm[21]. In practice, however, this principle of changing the decoding algorithm during the decoding process, designated as gear-shift, is not expected to improve on a code's threshold, but is rather employed to lower the decoding complexity[292]. This makes sense in a purely practical point of view, since an LDPC decoder will have to comply to a certain BER-*signal-to-noise ratio* (SNR) operation point in communication standards[34] with the least energy dispensed in the process.

Combining multiple decoding algorithms into obtaining a lower *gear-shif* decoding complexity should adhere to the following guidelines

i) the execution of most complex, and powerful, decoding algorithm should be kept to a minimum;

ii) the resulting overhead in decoding iterations should be well offset by the energy savings from executing lower complexity algorithms;

iii) the energy overhead in control logic in a dedicated accelerator should not offset the energy saved;

iv) minimization the BER degradation entailed by the application of lower complexity, and most times sub-optimal, decoding algorithms.

6.1.1 Gear-shift strategies

gear-shift decoding algorithms found in the literature combine several both soft- and hard-decoding algorithms for minimizing the decoding complexity. Some strategies deal with the problem of ensure lower complexity at the algorithm-level, regardless of an actual decoder implementation, while others make use of particular system-level traits[293] to pursue improved efficiency through gear-shift decoding.

Ardakani *et al.*, propose a gear-shift decoder which combines the soft-decoding *sum-product algorithm* (SPA) and MSA with the utilization of hard-decoding Gallager-B and -E algorithms[21]. The first problem arising from this combination is the domain-crossing when moving from the soft- to the hard-decoding domain. In particular, as observed in Table 6.1, albeit soft- to hard-decoding domain changes are deemed possible, even if some conversion steps depend on selected randomness (for instance, when the message magnitude is 0), hard- to soft-decoding transitions are not of straightforward compatibility. In hard-decoding algorithms there is no notion of how good is the estimate through the message magnitude, thus, one way to overcome this is to assign a fixed K constant value in the conversion[292]. By EXIT chart analysis, the authors prove that the decoding threshold obtained with the combination of the decoding algorithms is the decoding threshold of the best decoding algorithm, by consecutively moving from the most to the least complex decoding algorithm. Furthermore, the number of decoding iterations required is also reduced up to 30%, which is quite the opposite of an overhead, thus conveying a twofold improvement—better threshold and lower number of iterations required. Other authors, dealing with LDPC decoders that implement *pulse-width modulation* (PWM) in the system also propose gear-shift decoding for their design. This type of decoders relinquishes the use of buses to transmit messages of a certain bitwidth B_m. Instead, each bus is replaced by a single-wire, with the magnitude of the transmitted message given by the width of the pulse in clock cycles[295]. Unlike the previous case[292], complexity of the decoding algorithm is pursued with the minimum effort in mind, i.e., so long as the least complex decoding algorithm works, there is no need to execute a single iteration of the most complex one. Specifically, the *improved differential binary* (IDB) algorithm and the *offset min-sum algorithm* (OMSA) are combined in order to improve the energy efficiency of the decoder design while achieving the BER performance of the most powerful, and thus, more complex algorithm. In particular, the use of IDB brings single-bit memory requirements to store the *variable node* (VN) messages, thus with the appropriate clock-gating techniques, most of the memory can be turned off when the decoder executes the IDB algorithm. The overall results yield BER performances between that of the IDB and the OMSA, with substantially lower energy required to decode[295].

Table 6.1: Message domain change under SPA, MSA and Gallager-B and -E decoding of the LDPC decoder proposed by Ardakani *et al.*[292]. K is a selected constant value.

To From	SPA MSA	Gallager-E $\{-1, 0, 1\}$	Gallager-B $\{0, 1\}$
SPA MSA	no change	$\lvert m\rvert > 1, m \to \text{sign}(m)$ $\lvert m\rvert \leq 1, m \to 0$	$\lvert m\rvert > 0, m \to \frac{1-\text{sign}(m)}{2}$ $\lvert m\rvert = 0, m \to 0, 1$ randomly
Gallager-E	not straightforward $m \to K \cdot \text{sign}(m)$	no change	$\lvert m\rvert = 1,\ m \to \frac{1-m}{2}$ $\lvert m\rvert = 0,\ m \to 0, 1$ randomly
Gallager-B	not straightforward $m \to K \cdot (2m-1)$	$m = 0,\ m \to 1$ $m = 1,\ m \to -1$	no change

Another decoding strategy, which is not gear-shift decoding *per se* consists of building a decoder whose FUs implement different decoding algorithms for improved flexibility in face of SNR variations. When the SNR degrades, executing a more complex algorithm can be triggered by the degrading BER. Similarly, when SNR conditions improve a lower complexity one can also be triggered[296]. However, this strategy is limited to obtaining the BER performance and energy levels of each decoding algorithm, and not of the combination of the 3 implemented algorithms in the decoder.

6.1.2 MSA-based gear-shift decoder

While the combination of hard- and soft-decoding algorithm may be appealing, it requires domain conversions that consume resources that waste time and energy. Thus, we are motivated into combining decoding algorithms of the same decoding domain. Due to the MSA popularity, combining low decoding complexity, with the ability to be corrected into improving its BER performance, and thus, providing numerous BER-complexity tradeoffs in its *normalized min-sum algorithm* (NMSA), OMSA and *self-corrected min-sum algorithm* (SCMSA) variations, we are interested in pursuing a MSA-based gear-shift LDPC decoder.

Another reason to pursue this type of gear-shift technique is due to the waste of logic resources that occur when the least complex algorithm is operating. Cushon *et al.* report ∼90% hardware utilization for their gear-shift approach using the PWM OMSA and the IDB algorithms[295]. Furthermore, in *log-likelihood ratio* (LLR)-based domains, the message magnitudes begins the decoding process orders of magnitude below their final value. Thus, by varying the bitwidth B_m of the messages in the decoding process guarantees that only the required number of bits in the memory is being powered—for instance with clock-gating[295].

Taxonomy gear-shift decoding is a simple concept to grasp that makes no distinction to which complexity direction onto the decoder progresses when it commutes the decoding algorithm, regardless of any complex under the hood mechanisms which trigger the execution of the implemented algorithms. We introduce suitable designations for when the decoding complexity is increasing and for the case where it is decreasing. We designate both cases with regards to "gear changes", a "gear" a decoding algorithm, as in its original proposal[292], and the "gearbox" as the LDPC decoder as a whole. Under these designations we can then define two gearboxes.

***Accelerating* Gearbox** This strategy sees the decoding complexity increasing over time, i.e., only after a certain threshold is surpassed will the FUs commute from a lower complexity algorithm to a higher one. The rationale is based on a minimum effort heuristic. Decoding starts with the lowest complexity possible. If this algorithm fails to converge to a valid codeword within its allowed iteration window of operation, then the decoder will progress to attempt the decoding of the codeword under a higher complexity algorithm.

***Decelerating* Gearbox** This strategy sees the opposite trend in complexity over time. The motivation herein is to give the decoder a boost in its initial iterations. A higher complexity algorithm should accelerate the convergence of the codeword when in the initial iterations the LLRs have a low magnitude, and thus, reliability associated with each bit state is lower than in a posterior stage of the decoding process when LLRs are more reliable (higher magnitude).

It is worth noting, that under *extrinsic information transfer chart* (EXIT) chart analysis[292], the combination of decoding algorithms will always yield the BER performance of the best decoding algorithm case. However, such result can only be guaranteed for a very high number of decoding iterations issued. For the standardized codes case, the maximum number of allowed iterations is significantly lower (20 to 50). While accelerating and decelerating decoding would yield little to no difference for an infinite number of iterations, we expect to observe differences in obtained BER performance when the number of executed iterations is lower.

The MSA-based gear-shift LDPC decoder is based on the MSA and on the SCMSA. Thus, the high complexity decoder only activates the self-correction functionality in addition to the permanently executing MSA. In that sense the high complexity is still low, though the BER performance achieved is on a par with the LSPA[3]. The *region of interest* (ROI) of the MSA-based gear-shift decoder lies between the BER curves of the SCMSA and MSA, as shown in Figure 6.2. As aforementioned, our motivation is to run the minimum number of iterations in the SCMSA mode, while still reaching said BER performance, i.e., it is desirable for energy efficiency to run as many iterations under plain MSA mode. The SCMSA is a particular case. If the there is no sign change in a given $L(q_{nm})$ from the previous iteration $(i-1)$ to the current one (i), it behaves as the MSA, but with more logic resources in the datapath turned on. However, analytically executing the MSA while executing in SCMSA mode still means that the more complex datapath of the SCMSA has active. On the other hand, this creates an asymmetry on the attainable BER performance. While all the analysis in the literature assume that the performance of the highest complexity algorithm is better than the lower complexity counterparts, in this case, the SCMSA only sees higher complexity when sign changes occur in $L(q_{nm})$. Furthermore, as it violates the Gaussian approximation assumptions its behavior is not well-captured by either EXIT chart or *density evolution* (DE) analysis[83]. Thus, we are limited to an empirical analysis of the MSA-based gear-shift LDPC decoder BER performance.

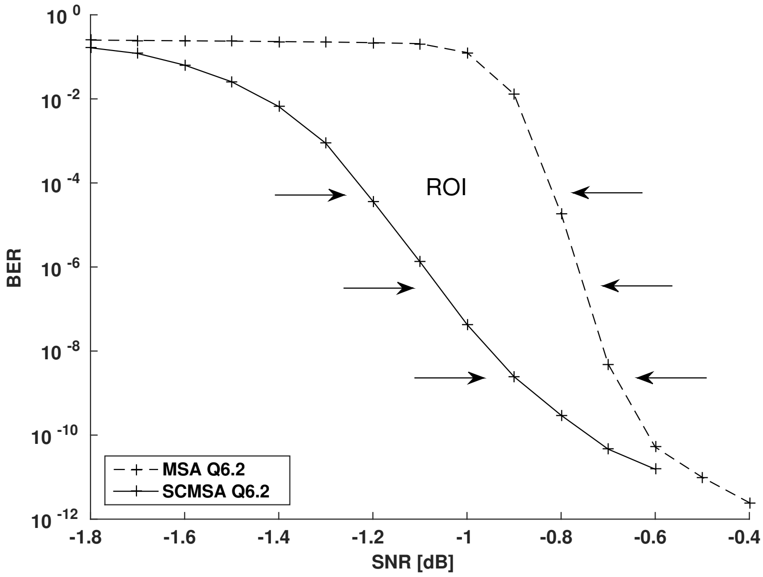
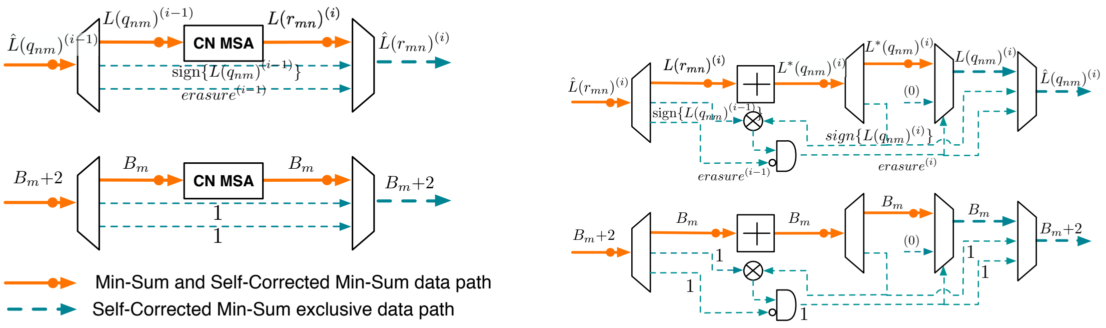


Figure 6.2: ROI of the MSA-based gear-shift decoder is located between the SCMSA BER curve and the MSA. The BER curves were plotted for the DVB-S2 rate $1/3$, $N = 64800$ bits for fixed-point quantization in $Qx.y$ format. Combining both algorithms in gear-shift decoding will result in a BER curve limited by the two.

6.1.3 Variable quantization bits and compact representation

Since only a limited number of bits in fixed-precision are required to reach within a negligible offset to floating-point equivalent BER[3,297], we are able to exploit a varying bitwidth, across the decoding procedure, in a two-fold approach. For the one, we know, heuristically, that in the first iterations the LLRs will not scale fast enough to require the maximum bitwidth $B_{m_{max}}$ defined in the decoder. Thus, a decoding iteration threshold can be fixed before which the LLRs magnitudes are clipped to a narrower bitwidth $B_m = B_{low}$, and in the trailing decoding iterations, i.e., after which the bitwidth employed expands to $B_m = B_{high}$.

Moreover, the usage of the SCMSA in most LDPC decoder architectures must be cleverly implemented, as the availability of messages from the previous iteration may not be available, considering that the majority write $L(q_{nm})$ and $L(r_{mn})$ in-place, the erasure bit and the sign must be preserved across iterations. Thus, as discussed in the efficient SCMSA LDPC decoder architecture[3] in Chapter 3, for the *two-phased message-passing* (TPMP) scheduling, the messages representation can be optimized as represented in Figure 6.3. By allowing $L(q_{nm})$ and $L(r_{mn})$ to have the following bit representation



(a) CN simplified datapath

(b) VN simplified datapath

Figure 6.3: MSA-based gear-shift decoder architecture datapath. In the upper row, the functionality is shown with regards to the messages circulating on each bus/wire, and in the bottom row the bitwidths of the buses are depicted.

$$\begin{aligned}
\hat{L}(q_{nm})^{(i)} &= erasure^{(i-1)} \mid sign\{L(q_{nm})^{(i-1)}\} \mid L(q_{nm})^{(i)} \\
\hat{L}(r_{mn})^{(i)} &= erasure^{(i-1)} \mid sign\{L(q_{nm})^{(i-1)}\} \mid L(r_{mn})^{(i)}, \\
erasure^{(i-1)} &= \begin{cases} 0, & L(q_{nm})^{(i-1)} = 0 \\ 1, & \text{otherwise} \end{cases}
\end{aligned} \tag{6.2}$$

where $\mid$ is the bitwise concatenation of the elements representation, and *erasure* is a signaling bit which is set to 1 whenever there has been an erasure in the previous iteration. This way, as detailed in Figure 6.3a), we can observe that the CN datapath does not suffer any changes in the datapath corresponding to computation as it will just slice, forward and concatenate the 2 *most significant bit*s (MSBs) in $\hat{L}(q_{nm})$ to $\hat{L}(r_{mn})$. In the VN datapath shown in Figure 6.3b), the same slicing of the 2 MSBs occurs, but they will be utilized in the datapath portion corresponding to the SCMSA to define whether or not an erasure is to be introduced at this stage. This architecture is slightly different than the one proposed previously in Chapter 2[3], as it entails a symmetrical data representation for both $L(q_{nm})$ and $L(r_{mn})$ messages. In this case, there is no need to evaluate in the CN functionality whether or not an erasure has indeed occurred, as this information is readily available in the AND gate in Figure 6.3b).

In the gear-shift decoder the hybrid representation in $\hat{L}(\cdot)$ of the erasure, sign and message $L(\cdot)$ means that all the datapath components dealing not only with the decoding algorithm functionality, but also with the blocks dealing with managing this overhead must be turned off for the decelerating gearbox, or are that are turned on for the accelerating gearbox.

accelerating and decelerating Transitions When accelerating the decoder starts in MSA operation, thus all the grayed out components shown in Figure 6.4 are turned off, and become active once the acceleration threshold t_a is reached. Afterwards, there is a bitwidth increase of 2 bits, regardless of the current bitwidth B_m of the words. In other words, although there is an increase of 2 bits per word, there is no improved precision due to more bits committed to the quantization of the LLRs. Furthermore, considering that if set to 1 the erasure bit forces the SCMSA datapath to produce the same results as the MSA would, and the fact that the sign bit is lagging behind one iteration, we can exploit the former to allow the sign bit to be correctly stored, but only one iteration after the SCMSA datapath has actually been activated.

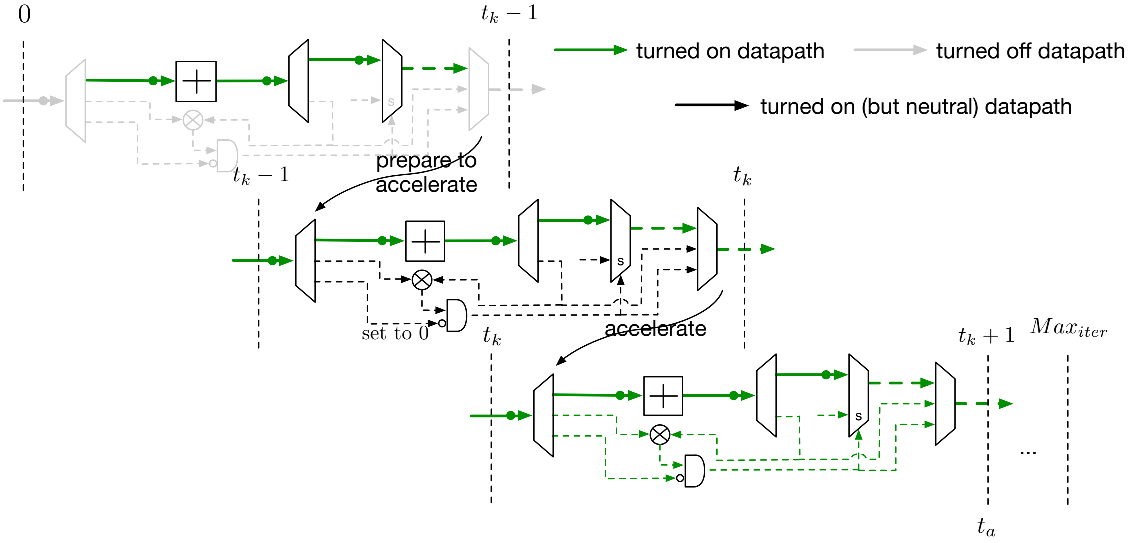
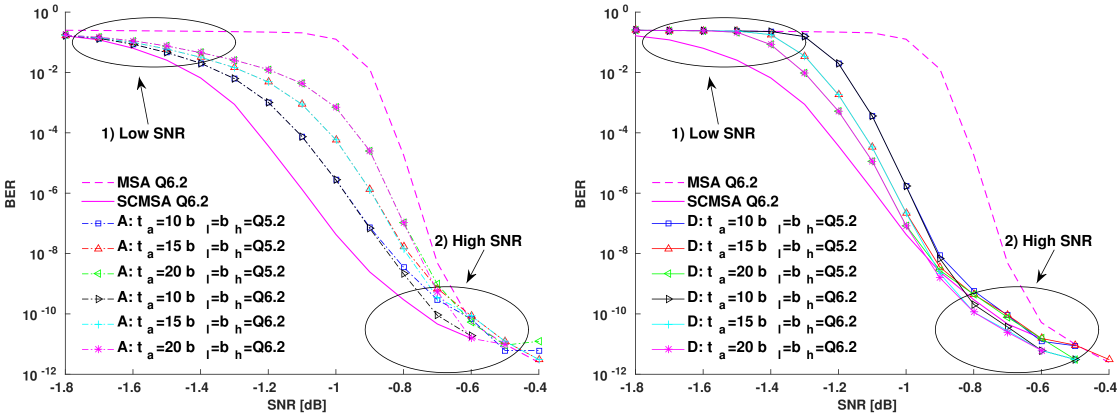


Figure 6.4: Datapath switching on transition of the *accelerating gearbox*. A latency of one decoding iteration is hidden by turning on of the SCMSA datapath one iteration in advance, though by setting the erasure bit to 1, the decoding algorithm is guaranteed to be the MSA and not the SCMSA. At the $(t_k - 1)$-th iteration the SCMSA is switch on, operates solely as MSA through the t_k-th iteration, in order to see sign$\{L(q_{nm})^{(i-1)}\}$ initialized to its correct value a the $(t_k + 1)$-th (t_a-th) iteration.

Hence, for an arbitrary t_k, the SCMSA will only become active at the $t_k + 1$-th iteration, even though its datapath has been activated two iterations in advance, i.e., at the $(t_k - 1) = t_a$-th iteration. In the t_a-th iteration, the algorithm is forced to execute the MSA by the erasure bit being set to 1. However, we refer to t_a as the algorithmic threshold, as under the hood, the decoder has switched on the full datapath an iteration earlier. Computationally, this means that if each individual operation in the SCMSA that come after the MSA datapath could be executed in a single clock cycle, the VN would see two clock cycles overhead. The accelerating transition is illustrated in Figure 6.4.

6.1.4 Experimental Results

The experimental validation of this architecture was performed for the DVB-S2 $N = 64800$ bits rate 1/3 LDPC code, running a maximum of 50 decoding iterations[61,241]. Furthermore, with the aforementioned modifications, in terms of FU and data representation and storage, the architecture is loosely based on the bit-parallel TPMP M-modulo decoder architecture[232]. We have set a gear-shift threshold of $t_a \in \{10, 15, 20\}$, for either the accelerating and decelerating approaches, and also, a bitwidth threshold $t_b = 20$ to commute between $b_l \in \{Q4.2, Q5.2\}$ and $b_h \in \{Q5.2, Q6.2\}$.



(a) accelerating gear-shift decoder

(b) decelerating gear-shift decoder

Figure 6.5: BER of the a) accelerating and b) decelerating gear-shift decoders for a fixed bitwidth b_l=$b_h \in Q\{5.2, 6.2\}$: 1) low SNR region and 2) high SNR region.

Accelerating Gearbox Performance The BER performance of the *accelerating gear-shift* decoder is shown in Figure 6.5a) for fixed-point representations of $Q5.2$ and $Q6.2$. Two regions are identified, 1) low SNR and 2) high SNR, since the BER performance of the gear-shift LDPC decoder shows clear convergence trends in those regions. In 1), the attained BER is that of the decoding algorithm executing in the trailing iterations. Naturally, this is due to the low SNR operation of decoder, which sees the number of decoding iterations required to converge close to the maximum number of iterations allowed. Thus, being the SCMSA the last executed algorithm, the BER curve enters the waterfall region, i.e., that between 1) and 2), closer to the SCMSA performance than to the MSA. As the SNR improves, the BER curves for $t_a \in \{10, 15, 20\}$ are spread between the SCMSA and the MSA curves. The better BER is achieved by $t_a = 10$, naturally, since this accelerating configuration executes more iterations of the most powerful algorithm. When the error-floor region is reached in 2), there is a non-distinguishable BER performance between both the SCMSA, MSA and *accelerating gear-shift* curves. In the accelerating decoder, for SNR operation between 1) and 2), then there is a BER performance tradeoff to be made.

Comparing the convergence speed of the accelerating decoder to the MSA and the SCMSA algorithms through the difference in average number of iterations required at each SNR simulated to successfully decode a codeword is made on Table 6.2. As observed in Table 6.2, the accelerating decoder, with certain exceptions around −0.90 dB for the A_3 and A_5 cases, always require less iterations than the MSA execution. Thus, not only is

Table 6.2: accelerating convergence to the MSA and SCMSA expressed in iterations (%). The tabulated entries in gray correspond to SNR operation points where the accelerating decoder requires more iterations to converge than the MSA or the SCMSA decoder.

	MSA						SCMSA					
SNR (dB)	A_1	A_2	A_3	A_4	A_5	A_6	A_1	A_2	A_3	A_4	A_5	A_6
-1.30	~0.00						-3.04					
-1.20	3.99	3.99	0.00	0.00	0.00	0.00	3.20	3.20	-0.76	-0.76	-0.76	-0.76
-1.10	1.13	1.13	3.74	3.74	6.74	6.74	-1.44	-1.44	1.20	1.20	4.41	4.41
-1.00	1.03	1.03	1.05	1.05	3.15	3.15	-10.08	-10.10	-10.06	-10.08	-8.13	-8.15
-0.90	7.09	10.08	-0.51	2.49	-2.27	0.70	-10.57	-10.59	-16.88	-16.90	-18.22	-18.26
-0.80	18.66	18.32	12.92	12.58	8.94	8.60	-8.57	-8.57	-13.53	-13.54	-16.67	-16.67
-0.70	12.02	11.97	8.82	8.80	6.89	6.86	-6.46	-6.46	-9.36	-9.34	-11.03	-11.01
-0.60	8.19	8.13	6.34	6.25	4.96	4.90	-4.58	-4.58	-6.31	-6.34	-7.56	-7.56
-0.50	6.09	6.06	4.85	4.76	3.68	3.65	-3.20	-3.21	-4.39	-4.45	-5.48	-5.48

the BER obtained better than that obtained with the MSA (c.f Figure 6.5a)), but also, the convergence speed can be as 18.66% faster. Naturally, when the SNR progresses to higher levels, this convergence speed is lost, as the number of required iterations decreases. Comparison of the accelerating decoder with the SCMSA case yields a much different scenario. In this case, the decoder convergence speed can only aspire to be as slow as the MSA is to it. Herein, the outliers are SNR operation points for which a lower number of iterations is required under gear-shift decoding. The remaining of which see a gap of up to 18.22% for the A_5 decoder, with a similar profile to the convergence speed difference with regards to the MSA, i.e., in the SNR regions 1) and 2) the difference is low and grows in the waterfall region reaching its peak at 0.90 dB.

Decelerating Gearbox Performance The BER performance of the *decelerating gear-shift* decoder is shown in Figure 6.5b) for fixed-point representations of $Q5.2$ and $Q6.2$. Similar to the previous case, there is convergence in the SNR region 1) to the MSA BER performance and in 2) to the BER of the SCMSA. It is clear that the performance in 1) is polarized once again by the algorithm run in the trailing decoding iterations and otherwise in 2). Due to such behavior, in this case, given the difference of the SCMSA and MSA BER gradients in the waterfall region, the BER performance of the decelerating decoder converges faster to the SCMSA performance. In this case, superior BER is achieved by running the SCMSA more iterations, thus the better performing configuration is for $t_a = 20$.

The convergence speed of the decelerating decoder to the MSA and the SCMSA algorithms is compared in Table 6.3. It stands out from Table 6.3, that the decelerating decoder is much faster than the MSA decoder. The aforementioned intuition that the most powerful LDPC decoding algorithm is more valuable for when LLRs are still low in magnitude proves correct. In this case, speed of convergence can grow to almost 30% at

Table 6.3: decelerating convergence to the MSA and SCMSA expressed in iterations (%). The tabulated entries in gray correspond to SNR operation points where the decelerating decoder requires more iterations to converge than the MSA or the SCMSA decoder.

	MSA						SCMSA					
SNR (dB)	D_1	D_2	D_3	D_4	D_5	D_6	D_1	D_2	D_3	D_4	D_5	D_6
−1.30	0.00	0.00	0.00	0.00	6.66	6.66	-3.04	-3.04	-3.04	-3.04	3.41	3.41
−1.20	6.25	6.25	3.46	3.46	2.17	2.17	5.44	5.44	2.67	2.67	1.39	1.39
−1.10	1.54	1.54	0.64	0.64	0.81	0.83	-1.04	-1.04	-1.91	-1.91	-1.75	-1.73
−1.00	2.56	2.56	6.63	6.68	9.27	9.31	-8.72	-8.74	-5.10	-5.08	-2.75	-2.73
−0.90	12.59	15.79	16.22	19.50	18.17	21.51	-5.97	-5.95	-2.94	-2.94	-1.31	-1.31
−0.80	25.07	24.75	27.77	27.47	29.19	28.86	-3.63	-3.61	-1.55	-1.50	-0.45	-0.43
−0.70	17.22	17.24	18.97	18.95	19.79	19.80	-2.11	-2.06	-0.66	-0.63	0.03	0.09
−0.60	12.20	12.21	13.31	13.28	13.73	13.70	-1.03	-0.97	-0.06	-0.03	0.31	0.34
−0.50	9.31	9.29	10.00	9.97	10.14	10.08	-0.26	-0.26	0.36	0.36	0.49	0.46

0.80 dB for decoders D_3, D_4, and D_5. However, this promising behavior does not stand up to the test when compared to the SCMSA convergence speed. In almost every SNR simulated, there is a need to execute more decoding iterations, a maximum of 8% more, but this difference quickly grows to negligible as the decoder progresses to the error-floor region.

Accelerating vs. Decelerating Gearboxes Finnaly, the accelerating and *decelerating gear-shift* strategies show that, indeed, there are non-negligible BER differences and convergence speeds when the maximum number of iterations is moderate[3]. As shown in Table 6.7, the difference in convergence speed is as much as 15% faster for the decelerating decoder than the accelerating one. As a consequence, and taking into account the BER curve profile, we unequivocally conclude the better suitability of the decelerating approach.

Table 6.4: accelerating and decelerating convergence difference expressed in iterations (%). The tabulated entries in gray correspond to SNR operation points where the accelerating decoder outperforms the decelerating decoder in executed decoding iterations.

SNR (dB)	$A_1 - D_1$	$A_2 - D_2$	$A_3 - D_3$	$A_4 - D_4$	$A_5 - D_5$	$A_6 - D_6$
-1.30	0.00	0.00	0.00	0.00	6.24	6.24
-1.20	2.12	2.12	3.47	3.47	2.20	2.20
-1.10	0.40	0.40	-3.09	-3.09	-5.93	-5.91
-1.00	1.50	1.50	5.23	5.27	5.42	5.46
-0.90	4.89	4.93	15.45	15.47	18.73	18.78
-0.80	5.13	5.15	12.87	12.92	17.88	17.90
-0.70	4.44	4.49	9.04	9.04	11.62	11.65
-0.60	3.58	3.64	6.37	6.43	8.10	8.13
-0.50	2.95	2.95	4.79	4.85	6.09	6.06

Variable Quantization Shifting Varying the B_m of the represented LLRs has an impact on the BER performance[236]. In the waterfall region, the more decimal bits, the better the performance, but to little gain, since LLRs will soon benefit from more integer bits as the BER progresses to the error-floor and LLR scale quickly. A good compromise is to define y=2 decimal bits and adjust the integer part x accordingly to the chosen B_m[3]. The BER obtained for the SCMSA and MSA decoders for $Q\{2.2, 3.2, 4.2, 5.2, 6.2\}$ is shown in Figure 6.6. It is clear that the highest impact of varying the number of bits in the LLR representation is on the obtained error-floor, with distinct levels of error-floor achieved, regardless of the decoding algorithm, for $Q\{2.2, 3.2, 4.2\}$, and we can safely assume that for $Q\{5.2, 6.2\}$ the behavior is the same.

The introduction of a two-step variable LLR quantization, commuting at iteration t_b from $B_m = b_l$ to $B_m = b_h$, should work in a way that does not interfere with the obtained error-floor, even though the decoder is clipping the LLR messages before the t_b iteration threshold to B_m=b_l. Knowing that in the error floor region the SCMSA and the MSA decoders still execute a number of iterations in excess of 15 iterations, we can set t_b= 15 to evaluate the BER performance of the decoding algorithm under accelerating and decelerating decoding for t_a=20.

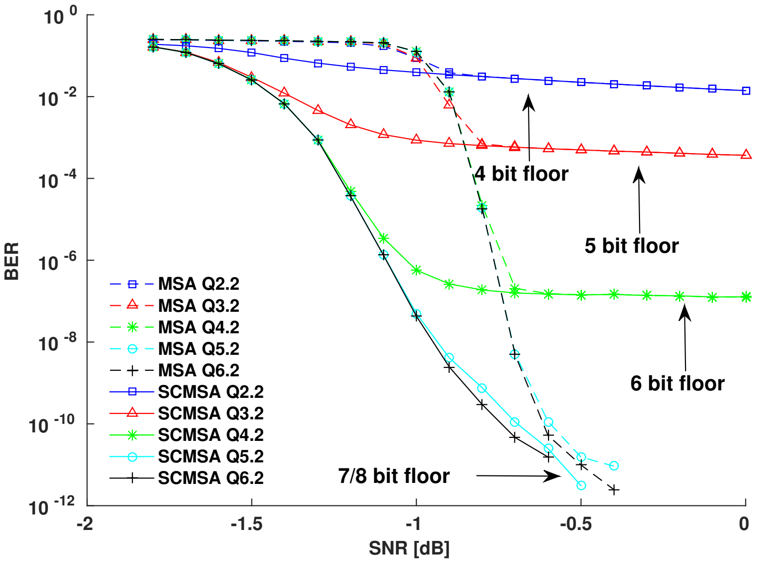
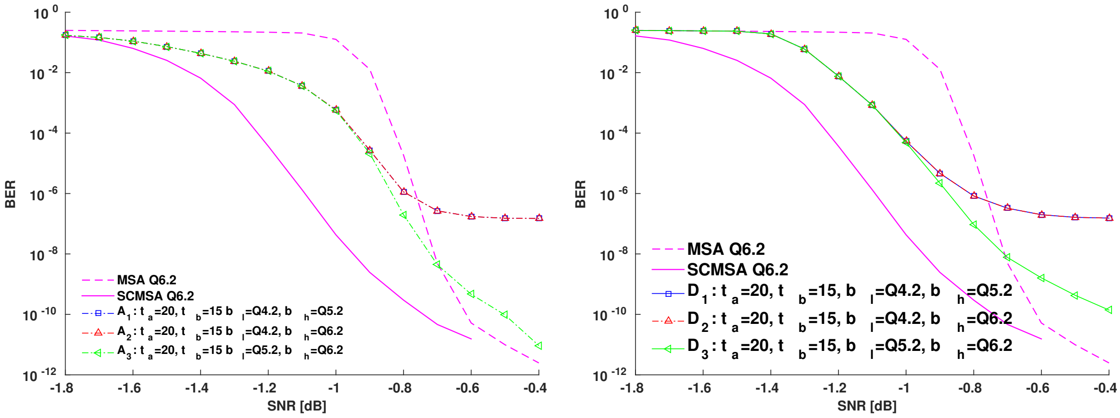


Figure 6.6: BER performance of the MSA and SCMSA decoders under different quantization levels.

The obtained results, shown in Figure 6.7, clearly show that even for a t_b=15, the obtained error-floor is the one obtained for the b_l bitwidth and not for the b_h bitwidth. Thus, while there was a strong motivation to reduce the energy consumption due to the switching of certain memory areas when operating in reduced bitwidth b_l, the BER performance proves otherwise. In addition to the degrading BER performance beyond



(a) accelerating gear-shift decoder

(b) decelerating gear-shift decoder

Figure 6.7: BER of the gear-shift decoders under variable quantization $b_l \in Q\{4.2, Q5.2\}$ and $b_h \in Q\{5.2, 6.2\} \setminus b_l$.

the MSA BER, there is also an increase of at least ∼11% in decoding iterations, when compared to the best fixed bitwidth configurations previously discussed.

6.2 LDPC Decoder under Unreliable Memory Storage

Fully reliable computing has been *de facto* paradigm for computer architectures ever since, as this ensures reliability of operation with reproducible results regardless of the underlying gate-level or instruction set characteristics. However, ensuring 100% reliable operation entails the inclusion of design guard-bands, desived for the worst case scenario, which can be bring tremendous penalty yields, as the exacerbation of process variations have been the norm when scaling down with Moore's law[246,298,299,299]. Precautions enabling 100% reliable operation under these high variations in the manufactured silicon have costly overheads in terms of power and area, adding to the challenge of developing efficient accelerators. In addition, this makes the window of opportunity for low-power techniques, such as aggressive voltage-scaling, to narrow down as systems become extremely sensitive to process variations, reducing its potential gains[300,301]. As a consequence, researchers have given their attention to alternative computation methods, such as *approximate computing*, whereupon the constraint of 100% reliable operation is relaxed in order to achieve lower power designs[286,290,291,300,302–304,304–306]. These architectures perform particularly well for as long as errors introduced at the system level do not go beyond a certain threshold at the quality or the performance of the application at hand[304]. It is particularly relevant to study the impact of memory unreliability on the communications systems performance, and in particular of LDPC decoders that see up to 90% of their area devoted to memory, so that designers can find operation points which

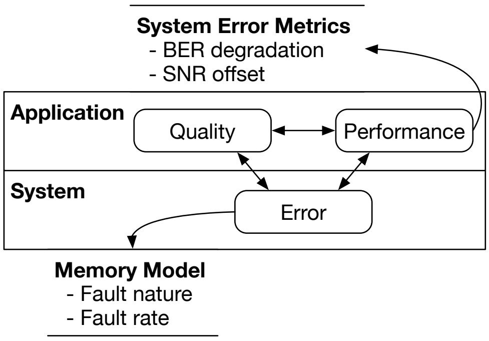


Figure 6.8: Metrics of correctness for LDPC *approximate computing* performance and system-level "correctness" parameters.

combine the BER performance negligibly close to that of reliable systems with the power savings of unreliable schemes[303,306,307].

While *approximate computing*, in its broader sense, deals with unreliability introduced at the system level, usually it refers to errors ensued by unreliable arithmetic logic. However, errors introduced by unreliable memory storage can be interpreted as a special case of *approximate computing*, as faulty bit-cells fail to retain data that is computed by functionally correct arithmetic logic. To assess how this impacts on the performance of the LDPC decoding system, metrics of "correctness" must be derived. Namely, at the system level, we can study and model how many faulty bit-cells are introduced into the decoder design, while at the application level what we interested in measuring is the resulting decoding degradation measured, seen as an equivalent degradation of the channel conditions (SNR) which expresses itself in degraded BER behavior (c.f. Figure 6.8).

6.2.1 Unreliable arithmetic and control silicon

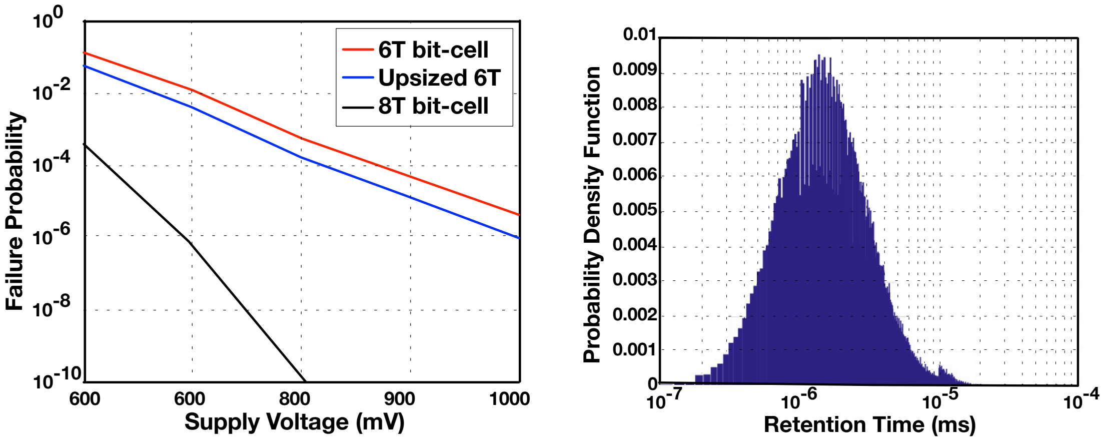
Due to the extensive variation introduced during the fabrication of silicon devices with the scaling down of technology nodes, accelerators should be protected against ensuing hard- and soft-errors. Unreliability at the FU and control unit operation has been addressed by May *et al.*[308]. Therein, votation schemes and duplication of silicon resources is performed to ensure 100% reliable operation of all components and to ensure that no data corruption will sacrifice the performance, i.e. the decoder design was made so that it would tolerate errors induced by the manufacturing variations in the silicon, and the developed strategies prevent those errors from propagating mostly due to redundancy of hardware resources. On the other hand, other researchers address these fabrication variations by allowing errors to propagate in the hardware. As desirable as it would be to ensure 100% reliable operation at the system-level, since this would entail correctness of the design at the application-level, the expression of errors allowed at the system-level can be contained at the application-level and their effect made to be negligible based on the *forward error correction* (FEC) properties of decoding system[290].

Error Mitigation at the Algorithm-level Ngassa *et al.* study the introduction of these errors at the FU-level alone, in particular for MSA-based decoders, whereupon they analyze what are the most critical arithmetic units for the BER performance. Studying the DE model, they show that the error propagation can be contained, though BER performance will still degrade[309]. Having determined the critical influence of the min unit in the MSA LDPC decoder they propose the utilization of the SCMSA as a way to contain the propagation of errors introduced by the faulty min operator[310]. The self-correction proves its capability of limiting the propagation of errors to the magnitude of LLRs. In its inception, the rationale behind the self-correction had been the containment of sign changes in the $L(q_{nm})$ message introduced by relaxing the log term in (2.27)[3]. However, the errors introduced by the min operator can be seen as an additional source of overestimation, as instead of the minimum the operator will yield the maximum, and BER degradation is therefore contained[309,310].

6.2.2 Unreliable Memory Storage

There are two sources that explain the presence of faulty-bit cells in an accelerator memory system, and therefore the accelerator operates under unreliable memory, that are related to the memory technology. 1) Aggressive voltage-scaling in traditional SRAM or 2) replacement of SRAM modules with eDRAM which must be operated with refresh rates that cannot guarantee 100% reliable operation[288]. The most common use of memory technology for special-purpose accelerators is SRAM. In the former, bit cells fail to retain the correct stored value due to sub-threshold operation of the silicon. Essentially, the quadratic voltage dependency of power makes it highly appealing to lower the operation voltage V_{DD} applied. However, the lower the V_{DD}, the higher the raw BER, i.e., low V_{DD} result in lower probability that the bit-cells will retain the written value correctly, as seen in Figure 6.9a).

In the latter, incorporation of the eDRAM modules explore the fact that no refresh procedure required on the memory will see a considerable power reduction. The rationale is to operate the memory in such a way that the system writing access pattern will see each bit-cell updated before the retained value is corrupted. For sub-45 nm processes, however, there is a high variation of the bit-cells DRT, as shown in Figure 6.9b). Hence, in most cases, the majority of the bit-cells can be operated reliably, but some bit-cells will be operated unreliably[246,307]

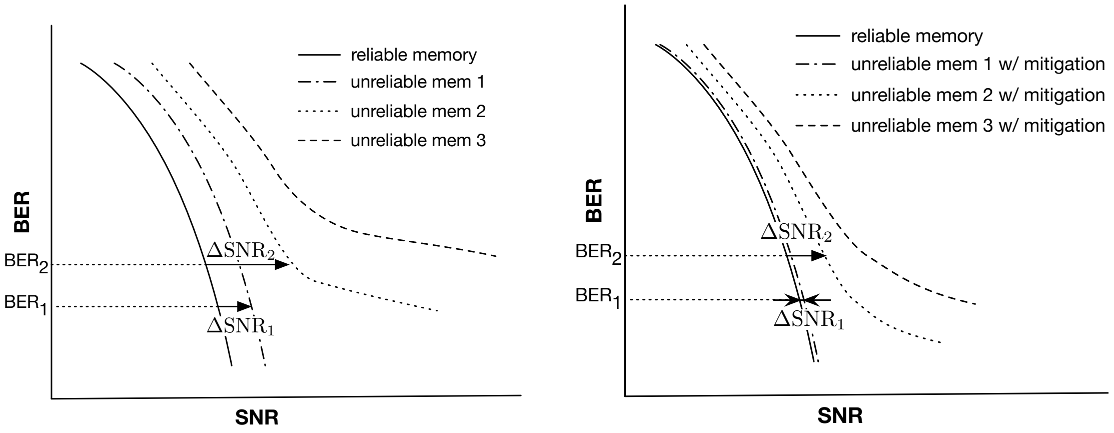


(a) SRAM bit-cell BER vs. operating voltage VDD

(b) eDRAM bit-cell DRT PDF

Figure 6.9: Unreliable operation of bit-cells in SRAM and eDRAM memories. The SRAM in a) failure probability corresponds to a 65 nm process[287,300]. The eDRAM DRT PDF in b) is for a 28 nm FD-SOI process[288].

The aforementioned methods that introduce unreliable storage to system designs, have wide and rich design spaces at the gate-level, a subject that lies out of the scope of this Thesis. However, at the system-level, the unreliable memory storage can be modeled in a way that is somewhat agnostic of the underlying gate-level details. In particular, we are interested in analyzing the BER degradation following the integration of unreliable memory modules into LDPC decoding designs as in Figure 6.10a), The remaining question becomes, how to make use of the inner error resilience of the LDPC code in a way that mitigates the observed BER degradation in combination with other mitigation techniques. There are additional strategies, designated as BER mitigation degradation



(a) Raw BER performance

(b) BER degradation mitigated performance

Figure 6.10: LDPC decoder operation under unreliable memory storage: a) BER degrades as a consequence of the unreliability introduced by the faulty memory; and b) BER improves by applying appropriate mitigation techniques.

techniques (c.f. Figure 6.10b), which further boost the error resilience of error-correcting applications to the system errors introduced by unreliable memories.

Unreliable memory modules are subject to probabilistic errors that can be modeled as a communication channel. This way, we can formalize "how much" unreliability is introduced, i.e., what is the probability of a bit-cell failure, the nature of the introduced fault, thus allowing a formal approach to propagate the errors introduced at the system-level through the LDPC decoder and assess their impact at the application-level. For simplicity of the analysis, we do not take into account the mapping of data elements d into their corresponding binary-valued entries s_k (typically performed using *two's complement* (2C) or *sign-magnitude* (*sign-mag*)). We assume that a binary-value entry, with k bits, s_k is written to the memory unit, which according to the writing address and bit-cell failure pattern can introduce errors to the stored s_k. Hence, instead of recovering s_k when reading at the same address, $\bar{s}_k$ is recovered, with bits stored in faulty bit-cells corrupted by the nature of the channel model's.

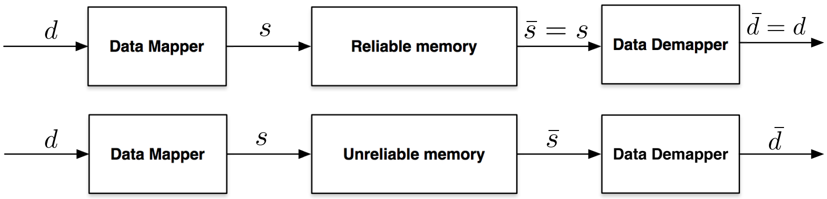


Figure 6.11: Reliable and unreliable memory taxonomy.

Binary Symmetric Channel A simple model for individual bit-cell failure is the *binary symmetric channel* (BSC)[311,312]. Herein, the channel $s_k \to \bar{s}_k$, is formalized by k parallel and independent BSCs, i.e., one for each bit-cell, with bit-flip probability ε. However, the BSC is not the most appropriate model to physically capture the silicon behavior on unreliable memory storage systems.

Stuck-at Channel To account for the deficiencies of the BSC into correctly modeling the physical phenomena which introduce faults to bit-cells in embedded *very large scale integration* (VLSI) processes with high variation, a better channel is required. Similarly to the BSC, the bit-cells fail independently with probability ε, however, they do so by becoming stuck-at 0 or stuck-at 1. Physically, this corresponds to the bit-cell circuit being shortened to power or to ground or to the transistor inability to commute[181].

6.2.3 Error Mitigation Strategies

Presence of faulty bit-cells in high density memories can be accommodated by a number of strategies that mitigate the impact of the faults on the application and performance correctness. Traditional ones include design safeguards or utilization of *error-correcting code*s (ECCs) in the memory that can elevate the memory area overhead to more than 30%[302,313]. However, other methods, such as data remapping aim at finding an appropriate data recoding that minimizes the level of disturbance introduced by the unreliable memory. Other methods are applicable at the system-level, with the intention not to be the preventing of the unreliability *per se*, but the mitigation of the performance degradation that it entails.

Application-level techniques Minimizing the performance loss of a system when memories are unreliable is an optimization problem for which objective functions that minimize the errors introduced by the faulty bit-cells in unreliable memory modules can be constructed, and tuned to each application so that the resulting performance of the system is improved. This has been studied for a number of problems concerning *multiple-input multiple-output* (MIMO) communication systems, such as, 1) achievable rate of the *binary phase-shift keying* (BPSK) modulation scheme, 2) repetition coding and 3) convolutional coding, when unreliable LLR memories are introduced[181]. Statistical correction has also been studied[294], whereupon a binary-valued entry s_k corresponding to an LLR is replaced by a statistical measure, such as the mean of not corrupted LLRs in the system.

Moreover, the degrading performance has also been studied for MSA-based LDPC decoders, whereupon the code threshold degradation is studied through a revised DE model that incorporates the *stuck-at channel* (SAC)[314]. This is based on the principle that the faults introduced by the corrupted bit-cells have a similar effect to an SNR degradation. In fact, under DE, the LDPC code threshold degrades to greater SNRs. Furthermore, DE permits obtaining another degradation measure relative to the overhead in decoding iterations between equivalent BER levels. However, DE models do not take system-level design parameters as input, and thus, leave out a number of design space options which determine the BER performance of the LDPC decoder.

System-level techniques While the aforementioned works concern the use of techniques applied at the application-level to mitigate the degradation of the system operation correctness, mitigation strategies can be applied at the system-level. In particular, by manufacturing hybrid memories, i.e., those incorporating different T cell designs, researchers have been able to construct a *high speed packet access* (HSPA) decoding system combining unreliable (6T) and reliable (8T) bit-cells. This way, the most sensitive part of the stored data can be protected by reliable cells while low cost unreliable ones still exist to lower the energy requirements[290].

The motivation behind the incorporation of unreliable memory storage systems onto communication systems has a broader rationale. Considering the aforementioned systems implement FEC, their underlying ECC already possess a certain error resilience. In fact, these systems deal with the correction of errors themselves, and thus, in a sense, the unreliability introduced by faulty bit-cells in unreliable memories may be expressed as a further degradation of the SNR[290].

All in all, incorporation of unreliable memory promises a prospect of tremendous power savings (up to 30% in the HSPA case) for manageable BER performance loss, thus, there is a strong motivation to assess how unreliability can be managed at the system- and application-level in order to exploit power reduction techniques based on unreliable memory storage incorporation onto LDPC decoders.

6.2.4 BER degradation mitigation strategies

The followed methodology to assess the performance of the LDPC decoder under unreliable memory storage is herein discussed. The aforementioned unreliable memory storage methods were discussed how to introduced cheap memory modules based on SRAM and eDRAM technology, the underlying setup does not take the memory technology into account. Instead, the level of errors introduced are modeled by the appropriate memory model, the SAC[181]. In this Chapter, the main figure of merit for our LDPC decoder becomes the BER performance and the decoding iterations overhead. BER Monte Carlo simulations must be performed beforehand the introduction of an unreliable memory module, based on a certain technology node and memory type, so that there is a guiding yield, above which the decoding success sees a too high degradation.

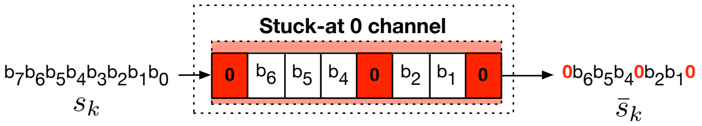


Figure 6.12: Memory storage following the stuck-at 0 channel. Bits b_7, b_3, b_0 will see their retained values stuck at the logical 0. However, a crossover in $\bar{s_k}$ only happens when bits set to logical 1 in s_k are stored in faulty bit-cells.

Considered Stuck-at 0 Channel Model The assumed channel model, the *stuck-at* 0 *channel* (SAZC) show in Figure 6.12, is a variation of the SAC where bits stored in faulty bit-cells become stuck at 0 only—faulty bit-cells are shorted always to the same voltage level, in this case, the one corresponding to a logical 0. Thus, the crossover probability becomes to half of the SAC case[181].

In addition, we will henceforth designate the failure probability of each individual memory bit-cell as $P_{S,SAZC}$. It is worth mentioning that in post-production testing, memories exceeding a given failure rate $P_{S,SAZC} \geq P$ of failing bit-cells are discarded. Fully reliable computation requires $P=0$, leading to yield penalties, and also, for a failing rate threshold P, the failure rate can be in the range of $[0, P]$, with failing rates around P having higher impacts on the error propagation to the application-level than an obtained $P_{S,SAZC} \approx 0$. To ensure that the LDPC decoder under unreliable memory will perform well under unreliable memory storage that passed the a given post-production discard threshold of $P_{S,SAZC} \leq P$, we consider the worst case scenario, i.e., the one where the maximum failure rate occurs.

As a consequence, we are able to make further considerations for a memory block m that stores W_m words, each with a bitwidth of B_m, and thus, possesses $S_m = W_m \times B_m$ bit-cells. Its number of stuck-at faults, under the considered scenario, is then fixed and given by $E_m = \lfloor S_m \times P_{S,SAZC} \rfloor$. In order to simulate this memory channel model, the bit-cell failure locations, i.e. the faulty bit-cell indexes, are modeled as a random variable that are *independent and identically distributed* (i.i.d.) and follow a discrete uniform distribution. Also, in order to assess in a computationally feasible time the BER performance of the LDPC decoder under unreliable storage, for each tested codeword there is a new bit-cell failure pattern in the memory. The motivation behind this is due to the following considerations. It would become a difficult problem to tackle analytically what would the worse error pattern in memory be like. Thus, we consider that the logical to physical bit-cell location is updated for each new iteration. This is a strategy not pursued for dedicated LDPC decoders, which is not necessary under 100% reliability, the same message is stored to the same bit-cell locations. When 100% reliable operation is no longer guaranteed, shuffling the failure locations ensures that whatever the worst error pattern is, it will only occur ever so often in time.

In order to assess the performance degradation ensuing from the unreliability introduced by faulty bit-cells, different setups must be considered. For the one, with loss of generality, LDPC decoders are divided onto two distinct memories spaces 1) the channel memory storing the likelihoods received from the channel demodulator, and 2) the messages memory, storing all intermediate data that is required in the computation of the LDPC decoding algorithm. Thus, the evaluation of the sensitivity of the decoding procedures to errors introduced onto each memory should be made. Usually, the channel memory is one or more orders of magnitude smaller than the messages memory, and will influence the decoding procedure since the initialization step, although at latter iterations its recurrent injection of errors will become negligible as the magnitude of the LLR messages increases, i.e. assuming that the LDPC decoder is operating in the

LLR-domain. The opportunity to greatly diminish the energy consumed by the channel memory is lower than on the messages memory due to the size difference. On the other hand, the tolerable fault injection rate, while still guaranteeing the correct decoding of codewords, is surely a more critical problem on the latter memory space, not only due to its much larger size, but also due to its recurrent injection of errors, not limited to only one type of messages, but both $L(q_{nm})$ and $L(r_{mn})$.

The SAZC is a special case of the SAC considering that physically bit-cells fail biased towards a logical value, in this case 0, corresponding to one of the voltage levels within the memory hardware. Naturally, the general SAC can be obtained from a memory following the SAZC, if consecutive writes interleave assertions to 0 and to 1. This is not considered, as it would add some overhead to the decoder design. Again, we are interested in modeling the worst case scenario for a particular set of memory characteristics.

Two's Complement vs. Sign-magnitude Data representation influences how the hardware errors propagate to the represented data. It is clear that there is no averaging of the introduced errors as only stuck-at 0 faults occur. Thus, a 2C representation sees errors differently than the *sign-mag* representation. Considering s_k and $\bar{s}_k$ in 2C representation, its value d_{2C} is given by

$$d_{2C} = -s_{k,B_m-1}2^{B_m-1} + \sum_{i=0}^{B_m-2} s_{k,i}2^i, \tag{6.3}$$

with with $s_{k,i}$ the i-th bit of the word s_k. It is readily observed that when a stuck-at 0 bit-cell exists at position j,d becomes

$$\bar{d}_{2C} = \begin{cases} -s_{k,B_m-1}2^{B_m-1} + \sum_{i=0}^{j-1} s_{k,i}2^i + \sum_{i=j+1}^{B_m-2} s_{k,i}2^i,\ j \in \{0, B_m-2\} \\ \sum_{i=0}^{B_m-2} s_{k,i}2^i,\ j = B_m - 1. \end{cases} \tag{6.4}$$

Thus, there is a difference $d_{2C} - \bar{d}_{2C} = 2^j$, but not at all times, only if a 1 is written to a stuck-at 0 bit-cell. Considering, for simplicity, that either logical values get written in the same frequency to all bit-cells, then the average error introduced is a negative offset of 2^{j-1}, in modulo arithmetic, i.e., even when the sign bit flips when s_k maps a negative value d, the corrupted word $\bar{s}_k$ sees the same offset difference. This representation suffers from a biased 2^j negative offset, in modulo arithmetic. In (6.4), two cases are distinguished, when the stuck-at bit position j affects all bits except the sign, and when the sign is affected.

Faulty bit-cells affect the *sign-mag* representation differently, as d_{sm} is obtained according to

$$d_{sm} = (-1)^{s_{k,B_m-1}} \times \sum_{i=0}^{B_m-2} s_{k,i} 2^i, \tag{6.5}$$

with d and $-d$ mapped to the same s_k word, except for the MSB, i.e., the *sign* bit. When a stuck-at 0 bit-cell exists at position j, the retrieved word is

$$\bar{d}_{sm} = \begin{cases} (-1)^{s_{k,B_m-1}} \times \left(\sum\limits_{i=0}^{j-1} s_{k,i} 2^i + \sum\limits_{i=j+1}^{B_m-2} s_{k,i} 2^i \right), \; j \in \{0, B_m - 2\} \\ \sum\limits_{i=0}^{B_m-2} s_{k,i} 2^i, \; j = B_m - 1. \end{cases} \tag{6.6}$$

When the faulty bit-cell affects the MSB bit (the *sign* bit), the demapped value $\bar{d}_{sm}$ is the same as the one demapped in the 2C data representation case. However, the error difference is in this case $d_{sm} - \bar{d}_{sm} = -s_{k,B_m-1} \times 2^j$, and thus the stuck-at 0 bit-cell, if not affecting the MSB, does not bias the offset to being always positive. It can also be negative, depending on the *sign* of the d_{sm}.

As a consequence of the introduced faulty bit-cells, the decoding algorithm will perform poorer than in the reliable memory case, thus seeing the yielded BER performance degrade[290]. In order to mitigate the impact that the faulty bit-cells possess on the decoder BER performance, mitigation strategies developed at the system-level can be followed in order to lower the impact of the errors introduced by the faulty bit-cells.

Unequal Bit Protection—*k*-MSB Protection This strategy is motivated by the fact that the errors introduced by faulty bit-cells that store MSBs have the highest impact under 2C and *sign-mag* representations. Especially, decoders operating in the LLR domain, see the greatest error when the sign bit is corrupted, and then the next highest magnitude error occurs for the next MSB. Considering that the LLR attributes to a positive value a higher probability of the represented bit state to be 0 (2.21), and to a negative value a higher probability of it being 1. However, for stuck-at 0 MSBs, the LLR is corrupted to represent the incorrect bit state. Hence, this strategy is based on the correct storage of the LLRs MSBs in reliable memories, or in reliable bit-cells, while the remaining of the word s_k continues to be stored in unreliable bit-cells (c.f. Figure 6.13). From the error magnitude introduced by a faulty bit-cell (6.6), (6.4), it is clear that while the first MSB can introduce a catastrophic error, due to representing the incorrect bit state, the remaining bit-cells also have a non-negligible effect. The higher their position in the MSB, the higher the error they may introduce. Hence, k-MSB protection works by not relaxing on the reliability assumption for the bit-cells retaining the k MSB in memory, as illustrated in in Figure 6.13. From (6.6), (6.4), we can also observe that the more bits are protected, the lower the magnitude of the introduced error by the faulty bit-cells. However, this is at odds with the power savings goal that relaxes on the 100% reliable operation requirement. The optimal tradeoff lies in the point where a significant fraction of the memory bit-cells is operated under lower power, and thus unreliably, permitting power savings in the memory system, though a sufficient number of MSBs is protected to ensure a decoding BER performance converging to the one obtained if the LDPC decoder would be under reliable memory operation. In SRAM technology, this mitigation technique has been proved feasible with a hybrid memory system combining 8T unreliable bit-cells for storing the *sign* bit (the MSB), while the remaining bits are stored in 6T bit-cells[290]. On the other hand, for eDRAM technology, this would involve the application of a refresh procedure with a rate ensuring the reliable storage of the MSBs, or the storing of these bits onto bit-cells with very high DRT[181,288].

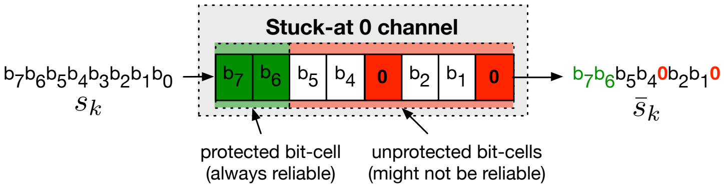


Figure 6.13: k-MSB protection. MSB bit-cells b_7 and b_6 will be always operated under reliable conditions in this case, while the remaining ones might have stuck-at failures such as b_3 and b_0, but not b_5, b_4, b_2, and b_1.

Follow-up Repair Iterations Another BER degradation mitigation strategy consists of introducing fault-free repair iterations at the end of the decoding procedure. This is motivated by the insight that errors in the decoder are harder to mitigate when the decoding process is in itself faulty—since the memory storing the LLRs is unreliable—and thus, a small number of fault-free decoding iterations may have the capability to remove such “residual errors”. This procedure is depicted in Figure 6.14. Essentially, the LDPC de-

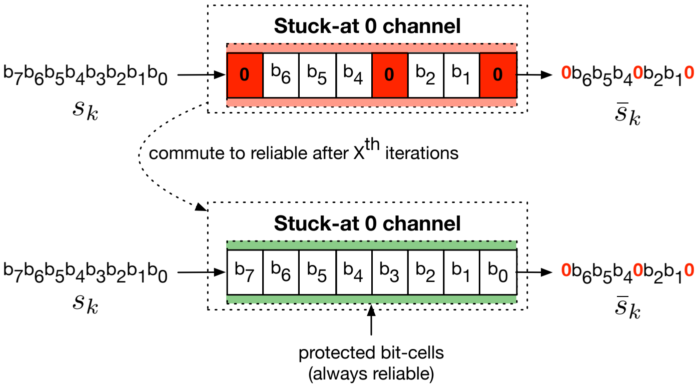


Figure 6.14: Follow-up repair iterations strategy. After a X iterations, the unreliable memories begin to be operated in a way that they become reliable again, thus not introducing errors henceforth.

coder will run a maximum of MAX_{iter} decoding iterations or finish early as soon as a valid codeword is decoded. Nevertheless, this does not influence the follow-up repair iterations scheme, it only affects how many iterations a received word has been decoded when the memory is operated unreliably and then reliably. The decoding procedure then becomes

a) the first X iterations ($0 \leq X \leq MAX_{iter}$) see errors present in all the memories in the LDPC decoder;

b) the following $MAX_{iter} - X - 1$ decoding iterations are run for a reliable message memory but the channel memory remains unreliable.

Physically, this would correspond to elevating the operation voltage V_{DD} in SRAM technology or increasing the refresh rate of eDRAM. Thus, the memories commute to a reliable operation point where data is not corrupted by the SAZC. Instead, as soon as a write operation is issued, there is no longer data corruption. However, the channel memory, being read-only, does not benefit from this, as it is written only once per decoding word. On the eDRAM case, one could think on the application of a faster refresh rate during the decoding phase *b)* guaranteeing that data will be retained correctly by the bit-cells. Characteristic of both SRAM and eDRAM repair strategies is the power overhead thus required.

6.2.5 Experimental Results

Table 6.5: Carried out simulations for the LDPC decoder with unreliable memory and applied BER degradation mitigation strategies.

	Details	Unreliable Memories	MSB Protection	Repair Iterations
Experiments	I	100% reliable	N/A	
	II	Channel	None$^{a)}$	None
	III		1-bit MSB	None
	IV	Channel and Messages	None	None
	Va)		1-bit MSB	None
	Vb)		2-bit MSB	
	VIa)		No	Yes X= 0, 100% iterations repaired$^{b)}$
	VIb)			Yes X= 9, 80% iterations repaired$^{c)}$
	VIc)			Yes X=14, 70% iterations repaired
	VId)			Yes X=19, 60% iterations repaired
	VIe)			Yes X=24, 50% iterations repaired
	VIf)			Yes X=29, 40% iterations repaired

a) equivalent to VI w/ X=50; b) equivalent to VI; c) it represents the maximum percentage repaired.

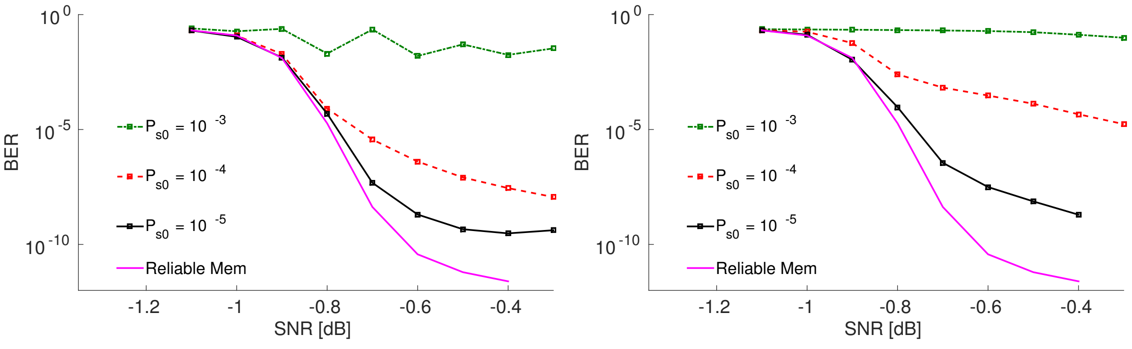
To evaluate the performance of the LDPC when unreliable memories are introduced in the decoders we perform Monte Carlo simulations, summarized in Table 6.5, for the

DVB-S2 LDPC normal frame ($N = 64800$ bits) $1/3$ codes, with block length N as defined in Table 6.6. The bit-cell failures follow a uniform distribution for a fixed number of failing cells, computed as $E_m{=}\lfloor S_m \times P_{s0} \rfloor$ for each memory block shown in Table 6.6, with P_{s0} the faulty inject-rate of bit-cells stuck-at 0, with $P_{s0} \in \{10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}\}$. Our simulations rely on a genie-aided early termination method, thus decoding only stops early whenever the decoded word matches the transmitted codeword or when reaching the maximum number of iterations. The BER simulation performance of the decoders is shown in Fig. 6.15 for no-protection and for k-MSB protection and in Fig. 6.16 for decoding with repair iterations.

Table 6.6: LDPC decoder and BER simulation parameters, memory block designation, type and dimensions.

	N Rate	Modulation	Decoding Algorithm	Memory	W_m	B_m	S_m
LDPC	64800	QPSK	MSA	Channel	64K	8	518K
Decoder	1/3			LLR	216K	8	1.7M

The maximum number of decoding iterations executed is 50, and the follow-up repair iterations experiment is run for $X \in \{0, 9, 14, 19, 24, 29\}$. Furthermore, in Table 6.7 we analyze the number of decoding iterations executed in all scenarios for a data point in the error-floor region. We analyze this in terms of the absolute value and additional number of iterations required for unreliable memories (experiments II, III, IV, V, and VI) compared to the fully reliable case (experiment I) using 2C data representation. In this regard, a note must be given to the actual number of repair iterations executed. Since we are using early-termination in the BER simulation, we know that the maximum number of repair iterations run is $MAX_{iter}{-}X$, the actual number of required repair iterations will differ for each simulated codeword.

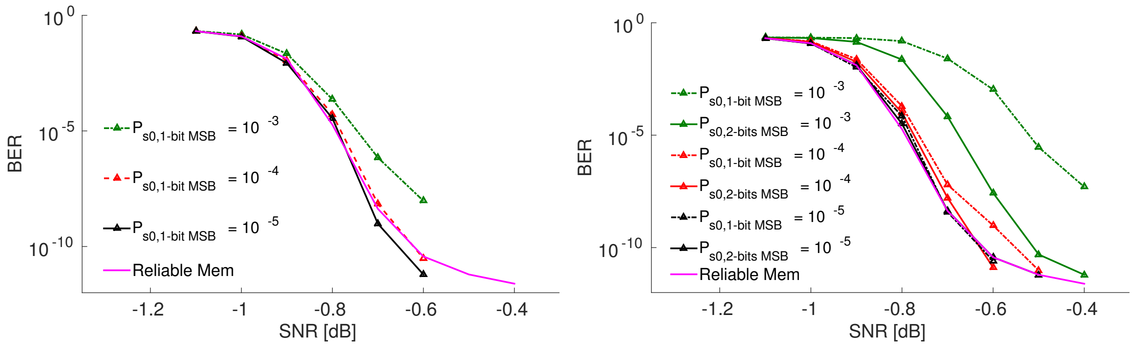


(a) LDPC exp. I and II

(b) LDPC exp. I and IV

Figure 6.15: LDPC BER performance under unreliable memory with stuck-at-0 probabilities $P_{s0} \in \{10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}\}$ for a) experiments I and II; and b) I and IV.

In Fig. 6.15a) we can observe the effects of introducing faults to the channel memory only. While the BER degradation is graceful on the waterfall region for $P_{s0}{=}10^{-5}$, only



(a) LDPC exp. I, IIIa) and IIIb)

(b) LDPC exp. I, Va) and Vb)

Figure 6.16: LDPC BER performance under unreliable memory w/ k-MSB protection with stuck-at-0 probabilities $P_{s0} \in \{10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}\}$, and $k \in \{1, 2\}$ for experiments I, IIIa) and IIIb).

to see the error floor be raised from 10^{-12}, under reliable memory storage, to 10^{-10}, this degradation is not so graceful for $P_{s0}{=}10^{-4}$ and $P_{s0} = 10^{-3}$. In the former, the error floor region starts 0.2 dB earlier than in the reliable memory case, with a degradation in terms of BER performance of six orders of magnitude, while in the latter, decoding performance is catastrophic since BER hovers around 10^{-1} for the tested SNR range. Considering that the channel memory takes up roughly a tenth of the bit-cells of the message memories, this degradation will only worsen considerably when faults are injected in both memories, as observed in Figure 6.15b). In fact, BER curves for the tested memory fault injection rates of $P_{s0} \in \{10^{-5}, 10^{-4}, 10^{-3}\}$ converge to higher BER levels and for higher SNRs. The $P_{s0}{=}10^{-3}$ is particularly daunting with no effect of the LDPC code decoding capacity at all. In a sense, what is observed for the higher SNR points is the noise introduced solely by the faulty memory, whose magnitude is too high for the underlying decoding performance. Graceful degradation of the BER is still observed for the $P_{s0}{=}10^{-5}$ case, but in a tightly contained SNR region as the trajectory of the BER curve converging to the reliable memory case is cut short on the waterfall region at BER levels of $\sim 10^{-7}$ and the error floor finally bottoms out at 10^{-8}. Clearly, the smooth degradation observed for other wireless coding schemes, such as *hybrid automatic repeat request* (HARQ) and Turbo of the HSPA+ standard[16,290].

Clearly, without a BER degradation mitigation strategy of some sort, the yielded BER performance of the decoder will not be able to comply to the standard requirements. Especially for the cases where the fault injection rate is high, where the opportunity to save on the memory power is higher. As previously discussed, the motivation for the mitigation strategies is to find one with low overhead capable to tolerating high fault injection rates[288,290]. In Figure 6.16a) the BER performance of the k-MSB protection is shown when only the channel memory is unreliable. As observed, 1-MSB brings the BER performance of the LDPC decoder to converge with the BER performance of the reliable memory case. Exception is made on the P_{s0}=10^{-3} which, nevertheless, sees its BER with a profile close to the latter. When faults are injected also in the messages memory, illustrated in Figure 6.16b), clearly, there is a performance gap between the P_{s0}=10^{-3} and the P_{s0}=10^{-4}, whereupon the BER performance in the former sees an irrecoverable shift of roughly 0.4 dB, though for the tested range of SNR values the error floor behavior is not observed, and thus, we cannot infer the behavior of the BER beyond the 10^{-7} level. For the latter case, and lower fault injection rates, the BER converges to the reliable memory case. If we step up the MSB protection to 2 bits (2-MSB protection) we observe that not only does the BER of fault injection rates of $P_{s0} \in \{10^{-4}, 10^{-5}\}$ have negligible degradation, but also the degradation ensuing for $P_{s0} = 10^{-3}$ is affordable when compared to the 1-MSB protection case. Also, as read in Table 6.7, the overhead in decoding iterations for at -0.5 dB is only relevant for $P_{s0} = 10^{-3}$ where it dropped from a staggering 50% to 13%, when increasing the one protected bit to two bits. There is a positive net gain here if we consider 8-bit words. The iteration overhead drops by almost fivefold, while the memory fraction that can be operated under lower power is reduced by a factor of $\frac{7/8}{6/8} \approx 1.17$.

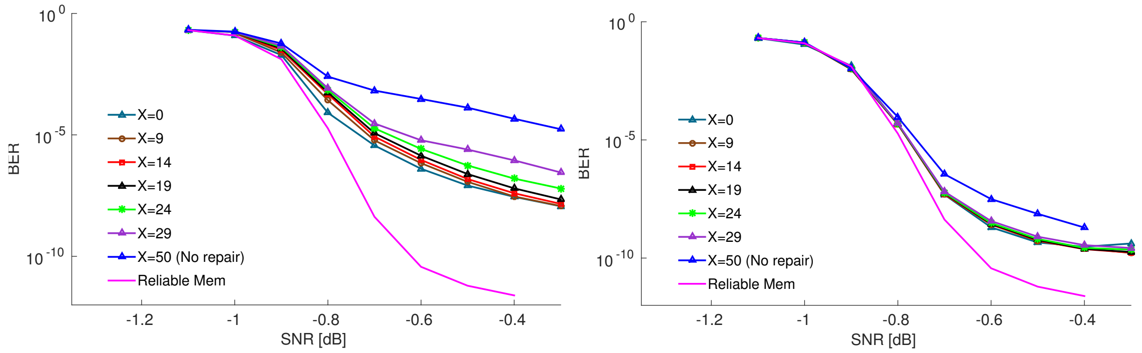
Table 6.7: Decoding iterations overhead under unreliable memory measured relative to the reliable memory scenario, for $SNR = -0.5$dB.

	Exp.	I		II		III		IV		Va)		Vb)	
	(−0.5dB)	Avg. Iter.	Δ iter. (%)	Avg. Iter.	Δ iter. (%)	Avg. Iter.	Δ iter. (%)	Avg. Iter.	Δ iter. (%)	Avg. Iter.	Δ iter. (%)	Avg. Iter.	Δ iter. (%)
P_{s0}	10^{-3}	33.04	N/A	50.00	51	38.09	15	50.00	51	49.50	50	37.41	13
	10^{-4}			50.00	51	33.62	2	50.00	51	34.72	5	33.44	1
	10^{-5}			38.16	16	33.18	0.4	38.16	16	33.10	0.2	33.09	∼0

	Exp.	VI			
	(−0.5dB)	$P_{s0} = 10^{-4}$		$P_{s0} = 10^{-5}$	
		Avg. Iter.	Δ iter. (%)	Avg. Iter.	Δ iter. (%)
χ	b) 9	43.04	30	35.18	6
	c) 14	43.93	33	35.28	7
	d) 19	45.05	36	35.42	7
	e) 24	46.86	42	35.94	8
	f) 29	49.05	48	37.17	13

6.2.6 Power savings for the eDRAM case

Due to the iterative nature of the LDPC decoding algorithms, sub-optimality, SNR degradation and errors introduced by unreliable memory storage can be accommodated to a certain extent by an overhead in the number of issued decoding iterations. Naturally, BER degradation is usually entailed as the average number of decoding iterations require to successfully decode are related to it. Notwithstanding, considering the SNR operation



(a) LDPC exp. I and VI with $P_{s0} = 10^{-4}$ **(b)** LDPC exp. I and VI with $P_{s0} = 10^{-5}$

Figure 6.17: LDPC decoder BER behavior for experiments I and VI, starting the repair at iteration $X \in \{0, 9, 14, 19, 24, 29\}$.

for which the LDPC is expected to operate, and setting a limit to the maximum BER at such operation point, we are able to determine what are the limits to power and energy savings ensued by the utilization of unreliable memory storage.

In Table 6.7, we analyze the overhead, $\Delta Iter$, in decoding iterations for each of the considered scenarios. Considering the decoder power

$$P_d = P_{FU} + P_{Mem}, \tag{6.7}$$

with P_{FU} the power drawn by the decoder FUs, and P_{Mem} the power drawn by the memory. It is clear that to an overhead in decoding iterations corresponds an equivalent power overhead. Thus, the overall power overhead ΔP_d breaks down into

$$P_d + \Delta P_d = (P_{FU} + \Delta P_{FU}) + (P_{Mem} + \Delta P_{Mem}) \,. \tag{6.8}$$

Observing (6.8), it is clear that both overheads depend on the number of iterations, but P_{Mem} and ΔP_{Mem} also depend on the BER degradation mitigation strategy implemented.

No BER Degradation Mitigation Strategy In this case, $P_{Mem} \propto Iterations$ and, thus, accepting the unlikely scenario that the BER is acceptable for the tested $P_{s0} \in \{10^{-3}, 10^{-4}, 10^{-5}\}$, only the latter fault injection rate sees a reasonable decoding overhead of 16% more decoding iterations issued. This means that the following condition must hold

$$\begin{aligned} P_{FU} + P_{Mem} &\geq (1 + \Delta Iter) \left(P_{FU} + P'_{Mem} \right) \\ P'_{Mem} &\leq \frac{P_{Mem} - \Delta Iter \times P_{FU}}{1 + \Delta Iter} \end{aligned} \tag{6.9}$$

with P'_{Mem} the power drawn by the unreliable memory.

Unequal Bit Protection—*k*-MSB Protection Under this approach, the limits to the power savings also follow (6.9), since the MSB protection scheme applied is constantly in use. A closer analysis to Table 6.7 reveals that under this mitigation strategy, $< X\%$ of more decoding iterations are issued under a $P_{s0} = 10^{-3}$ for 2-MSB protection and $< 5\%$ more decoding iterations are issued for $P_{s0} = 10^{-4}$ under 1-MSB protection. In the memory operation, the higher the number of bit-cells operated unreliably, the more the power savings. On the other hand, this entails a step up in the number of MSBs protected. Thus, fault-injection rates of 10^{-3} required 2 bits per word as opposed to a single bit for 10^{-4}. Notwithstanding, this tradeoff depends on particular memory technology and technology node considerations to be unraveled.

Follow-up Repair Iterations This strategy represents the most complex case. For the one, the BER will only converge to the performance of the scenario where errors are injected only in the channel memory. For the other, there are power savings at all times in the channel memory, but the message memory reverts to a reliable operation, thus drawing as much power as in the fully reliable decoder for as many follow-up repair iterations required. Thus, we can write the power saving condition as

$$\begin{aligned} P_{FU} + P_{Mem} &\geq (1 + \Delta Iter)\left(P_{FU} + P'_{Mem}\right) \text{ and } P_{Mem} = P_c + P_m \\ P_{FU} + P_{Mem} &\geq (1 + \Delta Iter)\left(P_{FU} + P'_c + \rho \times P'_m + (1-\rho) \times P_m\right) \\ P'_m &\leq \frac{\Delta Iter\,(P_{FU} + P_c) + \rho \times P_m}{\rho}, \end{aligned} \tag{6.10}$$

with P_c and P_m the power drawn by the channel and the messages memories, respectively and ρ is the fraction of follow-up repair iterations, i.e., $\rho = (MAX_{Iter} - X)/MAX_{Iter}$.

While it is fairly easy to find the operating conditions under which there are actual energy savings, their feasibility is a whole different subject. The hardest challenge in this regard is due to the assumed paradigm of operation, as the majority of hardware *computer-aided design* (CAD) tools assume 100% reliable operation. It is challenging to guide a memory compiler into synthesizing an unreliable memory block that is incorporated on a certain chip design. Similarly, *field-programmable gate array*s (FPGAs) or programmable hardware the memory interface required to either lower the operation voltages or to slow down the refresh rates beyond the point of reliability. All in all, the number of silicon-level design problems that arise from producing an unreliable memory system are such, that it would entail a tremendous effort well past the scope of this Thesis, though the work herein produced with regards to power savings through unreliable memory systems point in a promising direction not only for LDPC decoders in particular, but also to other ECC for wireless systems[16,290] and *approximate computing* in general.

Power Savings with eDRAM Technology To consolidate our argument behind the potential of power savings in the memory system of LDPC decoders by exploring the lowering of the refresh rate of eDRAM technology beyond the 100% reliability threshold, we attempt at producing an estimate on the power savings for a set of technology nodes applied to the LDPC decoder architecture simulated[232]. Moreover, we consider its possible configurations, some of which benchmarked in Chapter 5 under the dataflow decoder model, which range from $360/L$-factorizable setups, with $L = \{1, 2, 4, 8\}$, with clock frequencies set to $f \in \{100, 200, 400\}$ MHz.

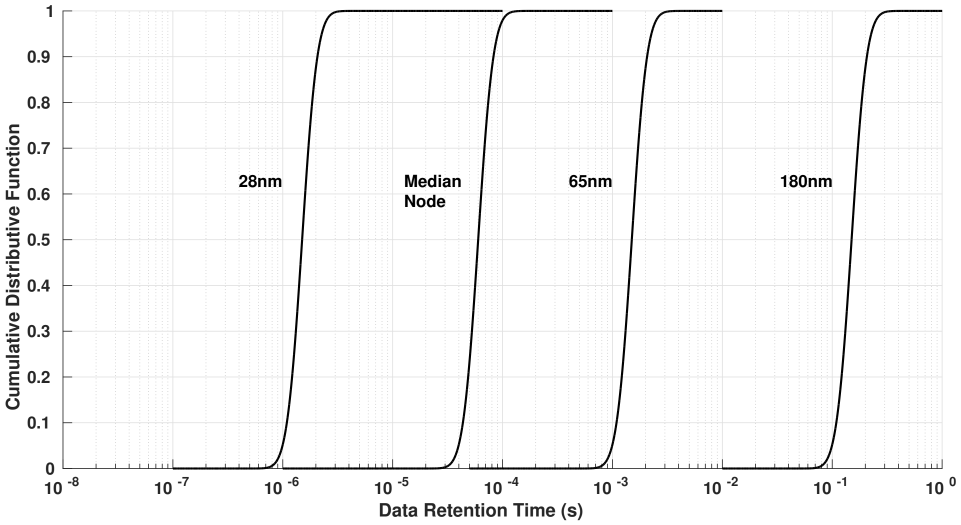


Figure 6.18: Typical DRT CDFs for eDRAM nodes on $\{28, 65, 180\}$ nm and in an estimated *median node*.

A caveat exists on the proposed methodology when applied to eDRAM technology. The underlying assumption behind the operation of this type of memory is that lowering the refresh rate, bit-cells commence to fail due to the refresh period extending these cells DRT. However, two factors must be weighed in on the discussion. The first is that the refresh procedure can render a memory completely inoperable for some technology nodes to be under 100% reliable operation. Essentially, the refresh procedure extends through the whole of the refresh period, hence, memory is never available for reading nor writing as it is constantly being refreshed. Thus, in this case the frequencies of refreshing applied must be relaxed and unreliable memory operation is a requirement for increasing the memory availability[288]. Secondly, if the average time between consecutive writes in the LDPC decoder system is faster than the minimum DRT for which a certain P_{s0} is ensured, then we can dismiss the refresh procedure altogether, although to do so, we need to evaluate the DRT probabilistic distribution for all nodes and configurations of the decoding system. The typical *cumulative distribution functions* (CDFs) for the deeply-scaled 28 nm, scaled 65 nm, 65 nm *complementary metal–oxide–semiconductor* (CMOS) (*median node*), and mature 180 nm eDRAM technology nodes are plotted in Figure 6.18 and in Figure 6.19 with respect to their required refresh rate.

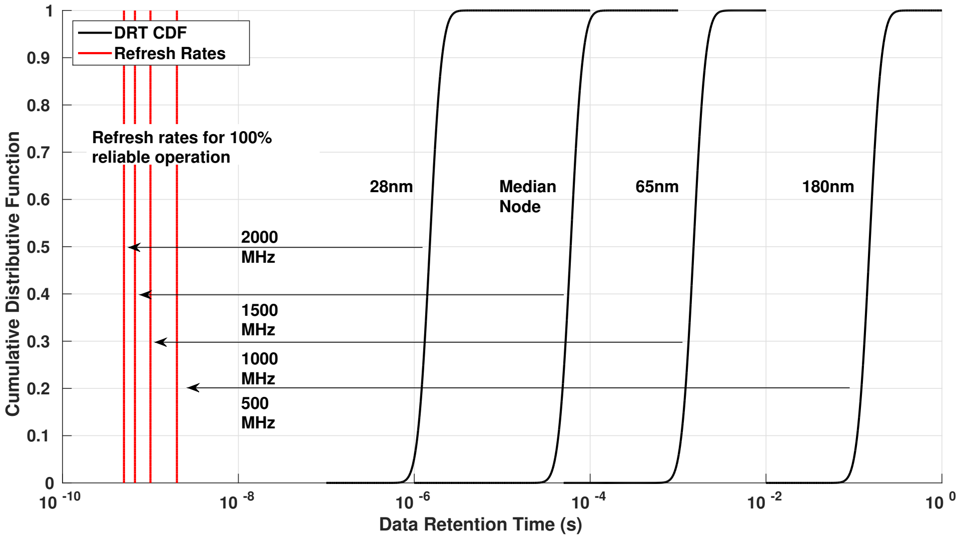


Figure 6.19: Typical DRT CDFs for eDRAM nodes and refresh rate periods on $\{28, 65, 180\}$ nm and in an estimated *median node*.

Under the refresh rates portrayed, the 28 nm node has no memory availability, 65 nm shows ~80%, and the 180 nm has insignificant unavailability, leaving the *median node* at around ~40% availability. In other words, this means that the 28 nm must forgo the refresh rate completely. Combining this information with the average time between consecutive writes of the same bit-cell, which can be approximated by the half the latency of a single decoding iteration we can project the operation points of the decoders onto each CDF as shown in Figure 6.20. As observed, the LDPC decoder configurations lie in the bottom tail of the distribution of the *median node* and on the upper tail of the 28 nm CDF. This entails that for the former, no refresh rate suffices to adhere to a probability under the required 10^{-3} fault injection-rate, which using a 2-MSB protection scheme leads to supportable BER degradation. According to Ganapathy *et al.* this unfolds at least a 45% power savings on the *median node* memory operation when the refresh rate is relaxed to that level[288]. Naturally better savings occur when the refresh rate is forgo. On the 28 nm case, it is not safe to assume that the relaxing of the refresh rate to allow for 50% availability at $P_{s0} = 10^{-3}$ will allow the decoder to read and write with the pace required for the decoding throughputs attained[232]. Regardless, the *median node* sees close to half the power saved, which accounting for the decoding iterations overhead and 1-MSB to 2-MSB protection leaves such saving at 38%, which is still a considerable feat for the proposed *k*-MSB protection scheme. Moreover, if we consider the 180 nm technology node discussed by Teman *et al.*[315], a reduction in memory power of ~55% is experienced,

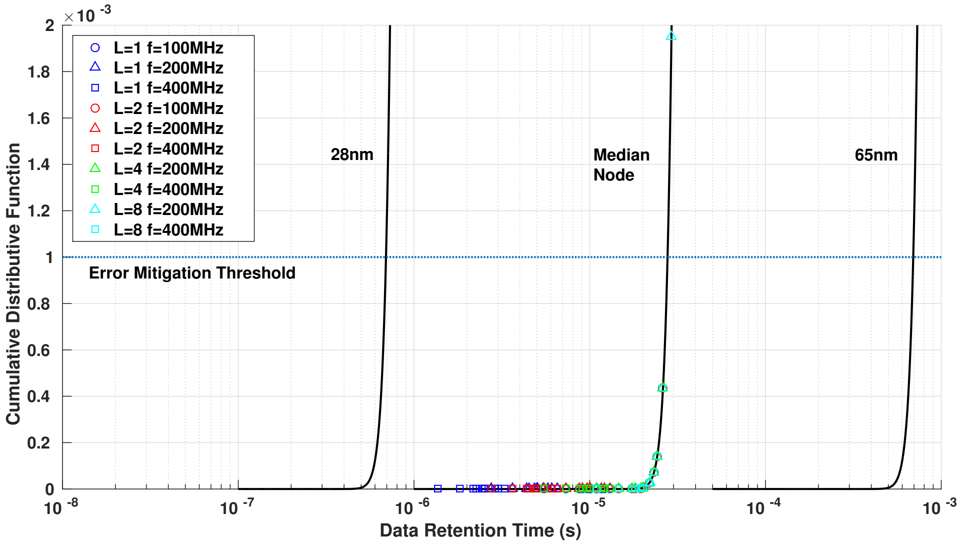


Figure 6.20: Zoomed in DRT CDF decoder configurations under the P_{s0}=10^{-3} threshold for $L \in \{1, 2, 4, 8\}$ and $f \in \{100, 200, 400\}$ MHz.

which compounded by the 2-MSB protection yields a gain of around 47% which is close to half the power required for the same eDRAM technology under 100% reliable operation.

6.3 Summary

In this chapter, we analyzed how to introduce algorithm and system-level modifications to LDPC decoders in order to bring the energy consumed down. The observed tradeoff involves the co-optimization of BER performance—both with regards to the BER level achieved and to the overhead number of decoding iterations required to reach that level—and energy saved at the system-level.

Gear-shift Decoding Whereas an energy saving optimization targeted at the decoding algorithm implemented by the FU will necessarily impact on the BER performance, it will save power at the FU-level per decoding iteration, but may not save energy over time due to the decoding iterations overhead[15]. In particular, we introduced the concept of accelerating and decelerating gear-shift decoding in order to study the behavior of the LDPC decoder when the most powerful decoding algorithm is applied first and then followed by the least powerful algorithm, and otherwise, respectively. Despite having been unable to reproduce the findings of Ardakani *et al.*[292], where the performance attained convergences to the best decoding algorithm, our MSA-based methodology for *gear-shifting* attains BER levels in the ROI with less decoding iterations required when compared to the least complex algorithm[15]. Considering under this methodology a dedicated decoder design implementation has not been performed yet, we withhold from advancing with an estimate to the power saved by the gear-shift FU, although a positive net gain is expected for the decelerating decoder when compared to the pure SCMSA approach[15].

Unreliable Memory Storage Due to the amount of logic devoted to memory, this fraction of the system represents where the highest power gains can be achieved. However, aggressive voltage- or clock-gating will eventually hit the 100% reliability wall. In order to improve the gains of these techniques, the 100% reliability requirement is dropped in favor of contained levels of unreliability introduced by faulty bit-cells. Under the proposed methodology, we characterize the fault-injection rates at which bit-cells fail, using the stuck-at model[16,181,290], in order to estimate the BER degradation entailed. To contain this degradation, we propose BER degradation mitigation strategies based on system-level techniques, 1) *k*-msb protection and 2) trailing repair iterations. The smart utilization of 1) allows that fault-injection rates that lead to catastrophic BER degradation to gracefully degrade for at least 2-MSB protected, as well as 2) that enables a graceful degradation towards the best possible scenario possible.

Considering that the methodology proposed modifies only the memory blocks at the system-level, it can be applied to existing decoder designs. To this end, we quantify the gains attainable with the methodology proposed for the incorporation of unreliable memory on LDPC decoders. For the partial-parallel decoder benchmarked[232], we show that 38% to 47% power savings are possible within negligible BER degradation, for the eDRAM technology at 65 nm CMOS (*median node*) and 180 nm. [a)]

[a)] The work herein described has been presented in[15–17].

7
Conclusions and Future Work

Contents

One of the main objectives of this Thesis was the proposal of methodologies for the development of efficient parallel LDPC decoders on both programmable and reconfigurable architectures. To this end, we have covered a selected design space area of the binary and non-binary LDPC decoding algorithms and codes. The proposed parallel kernels target single-GPU systems, distributed GPU cluster systems and modern heterogeneous CPU/GPU processors, using the data-parallel programming models CUDA and OpenCL. Parallelism is explored at different levels according to the underlying processor architecture and decoding algorithm employed. Namely, we have formalized a taxonomy for the thread-granularity of the different LDPC decoders found in the literature and discuss the suitability of the *thread-per-edge* (TpE), *thread-per-node* (TpN) and *block-per-codeword* (BpC) for the decoding role.

The objectives set forth have been reached by overcoming numerous challenges. The obtained results highlight that very high decoding throughputs are possible, and are especially important for the task of fast Monte Carlo BER simulation where analysis of the error-floor region down to 10^{-12} was made possible in hours of computation. This has enabled of the study of a *two-phased message-passing* (TPMP) decoding architecture for the *self-corrected min-sum algorithm* (SCMSA), whose BER performance was studied, evidencing a non-degrading behavior as opposed to other correction techniques such as scaling. Furthermore, the challenges overcome in the development of the Monte Carlo simulator for the GPU cluster entailed the proposal and optimization of auxiliary kernels for fast parallel execution. To this end, optimization of task- and data-parallelism levels were leverage such that the decoding throughput was maximized. These works were communication in the following publications:

[1] J. Andrade, G. Falcao, V. Silva, S. Yamagiwa, and L. Sousa, *Encyclopedia of Computer Science and Technology*. Taylor & Francis, 2015, ch. Accelerating Conventional Processing Using GPU Clusters: LDPC Decoders.

[2] G. Falcao, J. Andrade, V. Silva, S. Yamagiwa, and Sousa, “Stressing the BER simulation of LDPC codes in the error floor region using GPU clusters,” in *International Symposium on Wireless Communication Systems (ISWCS 2013)*, Aug 2013, pp. 1–5.

[3] J. Andrade, G. Falcao, V. Silva, J. Barreto, N. Goncalves, and V. Savin, “Near-LSPA performance at MSA complexity,” in *Communications (ICC), 2013 IEEE International Conference on*, June 2013, pp. 3281–3285.

In addition, the non-binary decoding case was studied on the GPU architecture. The proposed decoder solutions combine the insights gathered from the binary LDPC decoding case with the parallel *fast Walsh-Hadamard transform* (FWHT). These works have led to the publications:

[4] J. Andrade, G. Falcao, V. Silva, and K. Kasai, "FFT-SPA Non-binary LDPC Decoding on GPU," in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, May 2013, pp. 5099–5103.

[13] J. Andrade, G. Falcao, and V. Silva, "Optimized Fast Walsh–Hadamard Transform on GPUs for non-binary LDPC decoding," *Parallel Computing*, vol. 40, no. 9, pp. 449 – 453, 2014.

With the task-parallelism expression of the decoding algorithms performed on programmable architectures serving as a body of knowledge, we targeted reconfigurable devices in order to further improve the power efficiency of the LDPC decoders, while allowing fast execution. By taking advantage of *high-level synthesis* (HLS) tools we presented strategies for attaining high performances using distinct HLS approaches. Each approach was based on a different underlying generated architecture: MaxCompiler is based on a pure dataflow description written in Java; OpenCL (Altera) is based on an accelerator system composed of wide-pipeline kernels; and Vivado HLS is loosely defined by its C-based source code annotated with with directives for guiding the hardware generation, so-called loop-annotated. Under each model, we carried out optimizations with regards to data-parallelism and granularity of execution such that the characteristics of the LDPC decoder that attain the highest performance are identified. Namely, we have established the better overall nature of the dataflow approach followed by the MaxCompiler tool with regards to the quality of the decoder solutions obtained. Whereas the loop-annotated offers a compromise between a high flexibility of code writing but requires moderate to extensive annotation in order to reach good performances, the wide-pipeline approach was the most constrained design flow. Not only in respect to the nature of how logical to physical memory space mapping is handled, but also due to the composition of multi-kernel decoders which incur in high overheads. Hence, while moderate decoding throughputs can be obtained for dataflow and loop-annotated decoder designs, wide-pipeline decoders still have room to benefit if, for instance, the OpenCL compliance adherence was dropped in favor of better usage of the physical memory spaces within the FPGA logic. The following publications have been made concerning this work:

[5] J. Andrade, G. Falcao, V. Silva, M. Owaida, N. Bellas, C.D. Antonopoulos, and P. Ienne, "Towards High-Throughput with Low-Effort Programming: From General-Purpose Manycores to Dedicated Circuits," in *DATE'13: Workshop on Designing for Embedded Parallel Computing Platforms: Architectures, Design Tools, and Applications (DEPCP)*, March 2013.

[6] J. Andrade, F. Pratas, G. Falcao, V. Silva, and L. Sousa, "Combining flexibility with low power: Dataflow and wide-pipeline LDPC decoding engines in the Gbit/s era,"

in *Application-specific Systems, Architectures and Processors (ASAP), 2014 IEEE 25th International Conference on*, June 2014, pp. 264–269.

[7] J. Andrade, G. Falcao, and V. Silva, "Flexible design of wide-pipeline-based WiMAX QC-LDPC decoder architectures on FPGAs using high-level synthesis," *Electronics Letters*, vol. 50, no. 11, pp. 839–840, 2014.

[8] J. Andrade, N. George, K. Karras, D. Novo, V. Silva, P. Ienne, and G. Falcao, "Fast Design Space Exploration Using Vivado HLS: Non-binary LDPC Decoders," in *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, May 2015, pp. 97–97.

[9] J. Andrade, N. George, K. Karras, D. Novo, V. Silva, P. Ienne, and G. Falcao, "From low-architectural expertise up to high-throughput non-binary ldpc decoders: Optimization guidelines using high-level synthesis," in *Field Programmable Logic and Applications (FPL), 2015 25th International Conference on*, Sept 2015, pp. 1–8.

[10] F. Pratas, J. Andrade, G. Falcao, V. Silva, and L. Sousa, "Open the Gates: Using High-level Synthesis towards programmable LDPC decoders on FPGAs," in *Global Conference on Signal and Information Processing (GlobalSIP), 2013 IEEE*, Dec 2013, pp. 1274–1277.

[11] J. Andrade, G. Falcao, V. Silva, and K. Kasai, "Flexible non-binary LDPC decoding on FPGAs," in *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*. IEEE, May 2014, pp. 1936–1940.

Furthermore, we also identified certain key features of the *Silicon-to-OpenCL* (SOpenCL) compiler whose optimization leads to overall better quality of the generated hardware:

[12] M. Owaida, G. Falcao, J. Andrade, C. Antonopoulos, N. Bellas, M. Purnaprajna, D. Novo, G. Karakonstantis, A. Burg, and P. Ienne, "Enhancing Design Space Exploration by Extending CPU/GPU Specifications Onto FPGAs," *ACM Trans. Embed. Comput. Syst.*, vol. 14, no. 2, pp. 33:1–33:23, Feb. 2015.

Finally, certain techniques for improving the decoders power efficiency are proposed. They target algorithm modifications, in particular, through gear-shift decoding techniques based on the *min-sum algorithm* (MSA), and, also, the introduction of unreliable memory storage to the decoder architectures. The proposed methodologies for the former allow a seamless commuting from the MSA to the SCMSA based on a functional description of the architecture. In the latter, we have introduced methods which allow the relaxation of the 100% reliable memory operation to permit memory energy efficiency gains close to 38∼47%. These works were published in:

[15] J. Andrade, G. Falcao, and V. Silva, "Accelerating and Decelerating Min-Sum-based Gear-shift LDPC Decoders," in *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*, April 2015, pp. 5099–5103.

[16] J. Andrade, A. Vosoughi, G. Wang, G. Karakonstantis, A. Burg, G. Falcao, V. Silva, and J. Cavallaro, "On the performance of LDPC and turbo decoder architectures with unreliable memories," in *Signals, Systems and Computers, 2014 48th Asilomar Conference on*, Nov 2014, pp. 542–547.

[17] J. Mu, A. Vosoughi, J. Andrade, A. Balatsoukas-Stimming, G. Karakonstantis, A. Burg, G. Falcao, V. Silva, and J. Cavallaro, "The Impact of Faulty Memory Bit Cells on the Decoding of Spatially-Coupled LDPC Codes," in *Signals, Systems and Computers, 2015 49th Asilomar Conference on*, 2015.

7.1 Future Work

The work presented in this Thesis permits the continuing of research in fast and efficient *forward error correction* (FEC) using LDPC codes on programmable and reconfigurable architectures, and also on dedicated hardware.

Programmable LDPC Decoders on CPUs and GPUs A direction for the research of LDPC decoders on GPUs is to find design methodologies that do not set the maximization of the decoding throughput at odds with the latency minimization. The results obtained by Wang *et al.* are difficult to replicate on the GPU engine[156]. This evidences how limited the support for HyperQ and CUDA streams is nowadays, still. Notwithstanding, the theoretical bounds to this approach are extremely promising. So long as the overhead of managing a number of CUDA streams is kept low, the decoding throughputs projected are in the range of our proposed ones, with the latency reducing by a factor that is the inverse of the number of streams. A two-fold effect is conveyed, memory transfers are hidden behind computation completely and, thus, do not contribute to the decoding throughput, and a large data transfer is broken down into several smaller ones.

Moreover, whereas data-parallel programming models such as OpenCL can take advantage of both the CPU and the GPU cores on the same CPU/GPU chip, it does not convey a programming model where the performance in the CPU can exploit the high bandwidth purported by the memory cache and the computational power of wide *single-instruction multiple-data* (SIMD)-registers (AVX and SSE). However, dealing with intrinsics for vector processing incurs in higher development costs than compared to OpenCL kernel development. Since the OpenCL compiler tries to vectorize the instructions onto the SSE and AVX registers at its disposal on the CPU, there is still room to find what is the best OpenCL description, or way to write code, that can lead to a better packing of instructions—the comparison between the presented methodology and the work of Le Gal[165] shows that a high performance gap still exists.

Embedded and mobile processors for low power or battery-operated devices have not been considered in this Thesis. However, this breed of programmable architectures has pushed the envelope of power-saving techniques to the actual programming itself. Not only do they allow higher performance-to-watt ratios than conventional x86 and GPU devices natively, but also provide intrinsics to control how an algorithm runs across the different computational resources. For instance, big.LITTLE architectures[316], provide cores specifically optimized for low power and for high capability bounds, the latter within the thermal limits. This way, improved ways of *dynamic voltage and frequency scaling* (DVFS) across different cores are available to the designer to meet required computational performances within a limited power budge. With the increasing concerns with energy consumption, green and efficient LDPC decoder solutions will sooner or later exploit the use of mobile processors.

Reconfigurable LDPC Decoders using High-Level Synthesis The greatest challenge of the current generation of HLS is to overcome the quality of the provided solutions and balancing what knowledge is required of the designer. For the one hand, models built on C-based languages permit the targeting of FPGAs systems without recoding the code base. In some cases, a solution that is able to run on the FPGA can be generated, such as the case of OpenCL, while others require the integration of an IP core onto a host platform, such as Vivado HLS. On the other hand, approaches such as those provided by the MaxCompiler, which use the JAVA programming language for defining dataflow computation explicitly on hardware, make the development of hardware accelerators to incur higher *non-recurring engineering* (NRE) costs. The quality of the generated LDPC decoders was found to improve considerably when decoder architecture knowledge is introduced. The key tradeoff can be summarized by the lower the need to introduce hardware constraints to the compiler, the worse the flexibility of design and the poorer the quality of the obtained FPGA solutions. With this respect, we find that two key issues are loosely addressed in OpenCL. The platform that hosts the OpenCL kernel must be optimized for different kernel requirements, preferably at the OpenCL kernel level. In addition, the memory hierarchy must not be imposed by the compiler but, instead, the programmer should be allowed to define how key FPGA resources are consumed by the defined kernels.

Platform-agnostic Programming Addressing the on-going issues with data-parallel programming models on programmable architectures or with reconfigurable architectures using HLS is a challenge that can be overcome with different approaches. More often than not, the intended general approach to problem may be captured by the underlying programming model or architecture minutiae. In order to refrain from doing so, the optimization at the LDPC decoder architecture level should be made agnostic of the programming language and model utilized. By performing the decoder optimization at the LDPC domain and having code for that particular configuration be generated from an *intermediate representation* (IR) to any intended platform using, virtually, any programming model supported as a back-end, is a strategy that has bore fruits in other fields. For instance Delite is one such compiler infrastructure[317] and others can be constructed through LMS[318]. On the front-end, a *domain-specific language* (DSL) defined for the sole purpose of LDPC decoding can be defined using the Scala language[319]. This methodology has proved to be extremely effective in dealing with machine learning, data query and transformation, graph analysis (Opti{ML,QL,Graph})[320,321]. It is still unanswered whether the targeting of LDPC decoders to different computing architectures can benefit from this approach.

Unreliable Memory on FEC Systems The rationale behind *approximate computing* stems from the fact that a large percentage of power is wasted on the 100% reliability requirement. While this allows reproducible results across computing systems, certain applications can relax this requirement with little to graceful performance degradation, not necessarily perceived in the *quality of service* (QoS) metrics. The majority of the effort has, thus far, been placed upon the contained propagation of errors throughout unreliable logic, usually dealing only with the arithmetic component[291,322]. However, for the particular case of FEC systems we observe two amenable characteristics for the introduction of unreliability at the memory-system level. For the one, memory takes up the largest fraction of the decoding system. For the other, errors introduced by the incorrect retainment of data by the memory bit-cells is analogous to introducing a non-linear degradation to the communication channel. Due to their FEC functions, these systems come with inner resilience that can be exploited to more easily allow graceful degradation of the performance, in this case, usually measured as BER performance.

The initial effort has characterized the effects of memory unreliability for multiple FEC systems, *high speed packet access* (HSPA)+Turbo[16,290], block-LDPC[16] and spatially-coupled LDPC[17]. With it system-level mitigation strategies and statistical correction techniques have been proposed[294]. Furthermore, several memory systems have been characterized for the exploitation of unreliable memory operation[288,315]. However, it is still largely missing from the literature the actual system-whole design of FEC system incorporating unreliable memories. In particular, the potential body of knowledge gathered from designing dedicated systems that explore unreliable memory systems, and accompanying mitigation strategies, are not necessarily limited to the role of LDPC decoding.

A

Survey of the LDPC Decoders on Programmable Hardware

Table A.1: Summary of the Programmable LDPC Decoders.

Work	Parallelism		Data	LDPC Code					Decoding		Perf.			Platform
	Thread-	Data-	Width	Code	q	N	Rate	Tanner	Sched.	Alg.	Latency (ms)	Throughput (Mbit/s)	Iter.	Name
[106]	BpC	512	6–8	802.16e	2	768	1/2	Struct.	TDMP	MSA	2.62	150	10	GeForce 9800 GTX+
						1152					3.88	152		
						1536					4.91	160		
		384				1920					5.27	140		
		256				2304					3.69	160		
[107]	CpC	128	8	802.16e	2	2304	1/2	Struct.	TDMP	MSA	1.09	270	10	Cell B.E.
[108]	TpN	N/A	N/A	Regular [323]	2	2304	1/2	Sparse	TPMP	signed-log FFT-SPA	N/A	2.00	N/A$^{1)}$	GeForce GTX 650Ti
					4							2.20		
					8							2.50		
					16							2.50		
					32							2.40		
					64							2.00		
					128							1.50		
					256							1.00		
[109]	N/A	N/A	N/A	DVB-RCS	2	4800	2/3	N/A	N/A	MSA	N/A	2.0	N/A$^{2)}$	Tesla C2070
							1/2					1.9		
[110] [111]	N/A	N/A	N/A	Convolutional	2	2532	5/6	N/A	N/A	MSA	60.50	0.04	10$^{3)}$	GeForce GTX 460
										SPA	60.14	0.04		
						4608				MSA	90.96	0.05		
										SPA	90.26	0.05		
						6144				MSA	111.00	0.06		
										SPA	110.29	0.06		
[112]	TpE	N/A	32	N/A	2	816	1/2	Sparse	TPMP	LSPA	0.63	1.29	10	Tesla C2050
[113]						4000					1.64	2.44		
[114]						8000					1.81	4.42		
[115]	N/A use case of GPU-accelerated LDPC decoders for video coding on cloud computing													

Continued on next page

Table A.1 *Continued from previous page*

Work	Parallelism		Data	LDPC Code					Decoding		Perf.			Platform
	Thread-	Data-	Width	Code	q	N	Rate	Tanner	Sched.	Alg.	Latency (ms)	Thr. (Mbit/s)	Iter.	Name
[116]	BpN	1	32	Regular	256	6000	1/3	Sparse	TPMP	FFT-SPA	33.80	1.42	15	GeForce GTX 280
		4									33.70	5.69		2×GeForce GTX 295
[117]	CpC	48	N/A	Regular	2	8000	1/2	Sparse	TPMP	SPA	N/A$^{4)}$	N/A$^{4)}$	N/A$^{4)}$	Intel SCC
						20000								
[118] [119] [120]	PpE	1	32	Regular (Mackay)	2	999	9/10	Sparse 2-D Text.	TPMP	SPA	15.50	0.06	25	GeForce 8800 GTX
						816	1/2				31.30	0.03		
						1908	8/9				47.00	0.04		
						4896	1/2				39.00	0.13		
						4000					47.00	0.09		
[121]	CpC	96	8	Regular (Mackay)	2	576	1/6	Sparse	TPMP	MSA	0.69	79.3	10	Cell B.E.
						672	1/6				0.82	78.5		
						960	1/2				1.16	79.6		
						960	1/6				1.18	78.4		
						1152	1/2				1.39	79.6		
						1152	1/6				1.41	78.4		
						1248	1/2				1.51	79.6		
						1248	1/6				1.53	78.4		
[122]	TpN	16	8	DVB-S2	2	64800	1/4	Struct.	TPMP	MSA	N/A	79/87	10	Tesla C2050
							1/3					69~74		
							2/5					61~65		
							1/2					55~65		
							3/5					41~43		
							2/3					55~57		
							3/4					47~49		
							4/5					40~41		
							5/6					35~36		
							8/9					36~36		

Continued on next page

Table A.1 *Continued from previous page*

Work	Parallelism		Data	LDPC Code					Decoding		Perf.			Platform
	Thread-	Data-	Width	Code	q	N	Rate	Tanner	Sched.	Alg.	Latency (ms)	Thr. (Mbit/s)	Iter.	Name
							9/10					36∼36		
[123]	TpN	1	32	Regular (Mackay)	2	1024	1/2	Sparse 2-D Text.	TPMP	SPA	2.00	0.51	10	GeForce 8800 GTX
						4896					3.00	1.63		
						4000					3.00	1.33		
[124]	CpC	24	8	Regular (Mackay)	2	256	1/2	Sparse	TPMP	SPA	N/A	68.5	10	Cell B.E.
						504						69.1		
						1024						69.5		
	N/A	1	8			1024						2.08		Xeon Nehalem 2x-Quad
						8000						2.55		
	TpN	1	8			1024						14.60		GeForce 8800 GTX
			32									10.00		
			8			4896						31.90		
			32									17.90		
			8			8000						40.40		
			32									18.30		
			8			20000						40.10		
			32									11.30		
[125]	TpN	1	8	Regular (Mackay)	2	1024	1/2	Sparse	TPMP	MSA	2.00	4.00	10	Radeon HD 5870
						8000					2.00	22.00		
						1024					6.00	2.00		Phenom II X4-940
						8000					15.00	7.00		
						1024					1.00	17.00		Virtex6 LX760
						8000					3.00	17.00		
[126]	CpC	96	8	802.16e	2	576	5/6	Struct.	TPMP	MSA	N/A	77.7	10	Cell B.E.
							3/4					72.6		
							1/2					80.0		
						1248	5/6					78.2		

Continued on next page

Table A.1 *Continued from previous page*

Work	Parallelism		Data	LDPC Code					Decoding		Perf.			Platform
	Thread-	Data-	Width	Code	q	N	Rate	Tanner	Sched.	Alg.	Latency (ms)	Thr. (Mbit/s)	Iter.	Name
							3/4					72.9		
							1/2					80.0		
[127]	CpC	128	8	DVB-T2	2	16200	1/2	Struct.	TPMP	MSA	2592.00	3.5	20	Samsung Exynos 4412 SoC
							3/4				2764.80	3.5		
							5/6				2962.29	3.3		
						64800	1/2				592.46	3.2		
							3/4				592.46	3.0		
							5/6				628.36	2.8		
[128]	TpN	128	8	DVB-T2	2	16200	1/2	Struct.	TPMP	MSA	11.14	186.10	20	GeForce GTX 570
							3/4				10.78	192.40		
							5/6				10.94	189.60		
						64800	1/2				50.76	163.40		
							3/4				50.54	164.10		
							5/6				52.76	157.20		
						16200	1/2				43.75	47.4		i7-950
							3/4				43.65	47.5		
							5/6				45.28	45.8		
						64800	1/2				186.39	44.5		
							3/4				197.02	42.1		
							5/6				206.84	40.1		
[129]	TpN	1	32	DVB-T2	2	16200	N/A	Struct.	N/A	MSA	N/A	1.80	N/A	GeForce GTX 570
						64800						0.90		
[130]	TpN	1	32	Cyclic (PG)	2	1057	244/1057	Struct.	TPMP	LSPA	N/A	0.63	3.0 dB	GeForce GTX 285
										MSA		0.51		
						4161	10/57			LSPA		0.63	4.0 dB	
										MSA		0.27		
[131]	TpN	1	32	Cyclic (PG)	2	237	46/237	Struct.	TPMP	LSPA	N/A	0.46	3.0 dB	GeForce GTX 285

Continued on next page

Table A.1 *Continued from previous page*

Work	Parallelism		Data	LDPC Code					Decoding		Perf.			Platform
	Thread-	Data-	Width	Code	q	N	Rate	Tanner	Sched.	Alg.	Latency (ms)	Thr. (Mbit/s)	Iter.	Name
						1057	244/1057					0.39		
						4161	10/57					0.14	3.5 dB	
				802.11n		1944	1/2					0.5	1.5 dB	
				802.16e		2304						0.85	1.5 dB (∼ 15)	
[132]	TpN	N/A	N/A	QC-LDPC [324]	2	5120	1/5	Struct.	TPMP	LSPA	∼1.60	N/A	N/A	GeForce 9600 GT
						6144	1/6				∼1.60			
						10704					∼3.80			
[133]	BpC	N/A	32	QC code candidates	2	N/A	N/A	Sparse	TDMP	OMSA	N/A	$24.50^{5)}$	N/A	GeForce GTX 260
[134]	N/A	N/A	32	802.3an	2	2048	21/25	Sparse	TPMP	SPA	N/A	24.5	$10^{6)}$	GeForce GTX 480
												146.6		
[135]	N/A	N/A	8	LDPC-IRA [195]	2	2432	1/2	N/A	TDMP	OMSA	N/A	28.00	$10^{7)}$	Storm-1
												29.90		
[136]	N/A	64	8	802.16e	2	1536	1/2	N/A	TDMP	OMSA	N/A	32.9	$10^{a)}$	
												32.6	$10^{b)}$	
												31.1	$10^{c)}$	
												21.7	$10^{d)}$	
[137]	TpC	32	N/A	Regular	2	1008	1/2	N/A	TPMP	SPA	N/A	S[160]	N/A	GeForce GTX 450
		64												
		128												
		1024												
		8000												
		128		802.16e		2304								GeForce GTX 295
		1024												
		3000												
		6400												
						2532					6000	1.20		
[138]	TpN	32	N/A	Convolutional	2		5/6	Struct.	TPMP	N/A			30	GeForce GTX 460

Continued on next page

Table A.1 *Continued from previous page*

Work	Parallelism		Data	LDPC Code					Decoding		Perf.			Platform
	Thread-	Data-	Width	Code	q	N	Rate	Tanner	Sched.	Alg.	Latency (ms)	Thr. (Mbit/s)	Iter.	Name
						3792					37000	1.31		
						4608					74000	1.35		
						6144					371000	1.37		
[139]	TpN	4	32	Regular	2	4000	1/2	Sparse	TDMP	MSA	N/A	965.60	10	GeForce GTX 660
						4896						961.90		
						8000						951.90		
						20000						943.90		
[140]	TpN	3072	8	802.16e	2	768	1/2	Struct.	TDMP	MSA	N/A	704.00	5	GeForce GTX 580
		2048				1152						689.00		
		1536				1536						712.00		
		1024				1920						695.00		
		1024				2304						235.60		GeForce 9800 GTX+
												618.30		Tesla C2050
												710.00		GeForce GTX 580
		3584		802.11n		648						633.00		
		1536				1296						691.00		
		1024				1944						685.00		
[141]	TpC	143360	8	Regular	2	204	1/2	Sparse	TPMP	PLRA	53.10	550.40	10	GeForce GTX 660Ti
		71680				816					108.50	539.00		
		25088				4000					194.60	515.60		
		12544				8000					198.00	506.70		
		4480				20000					183.40	488.60		
		8960				9972					421.50	212.00		
[142]	TpN	16	8	N/A	2	1000	1/2	N/A	TPMP	SPA	N/A	Speedup Sequential CPU	N/A	Tesla C1060
						3000								
						6000								
						10000								

Continued on next page

Table A.1 *Continued from previous page*

Work	Parallelism		Data	LDPC Code					Decoding		Perf.			Platform
	Thread-	Data-	Width	Code	q	N	Rate	Tanner	Sched.	Alg.	Latency (ms)	Thr. (Mbit/s)	Iter.	Name
						20000								
[143] [144]	N/A	N/A	N/A	LDPCA	2	1584	53/66	N/A	N/A	SPA	173.00s (Fm) 174.00s (Sc) 157.00s (Cg) 128.00s (HL)		N/A	Tesla C1060
[145]	TpN	N/A	N/A	CMMB	2	9216	1/2	Struct.	N/A	N/A	18.00	5.12	10.3	GeForce 8800 GT
[146]	TpN	N/A	N/A		2		1/2				18.00	5.12	10.3	
							3/4				15.20	6.06	10.1	
[147]	BpN	N/A	N/A	N/A	16	6048	1/4	N/A	Seq.	FFT-SPA	N/A	5.00	N/A	GeForce GTX 580
							1/2					12.50		
							1/4		TPMP			3.00		
							1/2					8.50		
[148]	N/A	1	7	802.11n	2	648	1/2	Struct.	TPMP	MSA	0.05	13.4	3	Spartan6 LX150T
[149]	TpN	1	N/A	LDPCA	2	N/A	N/A	Sparse	TPMP	LSPA		0.15 (HL) 0.44 (Fm) 0.48 (Sc) 0.46 (Cg)	N/A	GeForce 9800 GTX+
[150]	N/A	N/A	N/A	N/A	32	651	1/8	N/A	N/A	Min-Max	43.07	0.48	15	GeForce GTX 650Ti
[151]	N/A	N/A	N/A	QC	2	160	N/A	Sparse	TPMP	IRRWBF	0.13	1.24	5	PXA320
						1280					1.13	1.13	5	
				π-rot		2000					1.92	1.04	5	
[152]	TpN	1	N/A	802.15.3c	2	672	1/2	Sparse	TPMP	MSA	18000.05	N/A	5	Tesla C2050
		16									2000.39			
[153]	TpN	252	N/A	802.11n	2	1944	1/2	Sparse	TPMP	LSPA	12.56	39.01	10	GeForce GTX 470
		224		802.16e		2304					10.59	48.74		
		168		Regular		3072					10.54	48.96		
		84				6144					10.92	47.27		

Continued on next page

Table A.1 *Continued from previous page*

Work	Parallelism		Data	LDPC Code					Decoding		Perf.			Platform
	Thread-	Data-	Width	Code	q	N	Rate	Tanner	Sched.	Alg.	Latency (ms)	Thr. (Mbit/s)	Iter.	Name
		42				12288					10.46	49.35		
		14				24756					7.14	48.21		
[154]	BpN	1	N/A	Regular [325]	32	620	1/2	Sparse	TPMP	Min-Max	N/A	0.02	10	i7-640LM
												0.04		Phenom II X4-945
												1.26		GeForce GTX 470
[155]	TpN	300	N/A	802.11n	2	1944	1/2	Struct.	TPMP	NMSA	N/A	39.82	10	GeForce GTX 470
				802.16e		2304						52.31		
[156]	TpN	1280	32	802.11n	2	1944	1/2	Struct.	TPMP	NMSA	N/A	316.07	10	4×GeForce GTX Titan
				802.16e		2304						236.70		
		6									0.21	62.50		
		12									0.24	110.25		
		80									0.27	155.43		
		224									0.34	201.39		
		320									0.43	253.36		
		1600									1.27	304.16		
[157]	TpN	N/A	32	Regular	2	2048	1/2	N/A	N/A	SPA	< 1.00	N/A	N/A	GeForce 8800 GT
						4096					< 1.00			
[158]	TpN	N/A	N/A	Regular	2	10240	101/320	Sparse	TPMP	SPA	< 5.00	N/A	30	GeForce 8800 GT
[159]	BpC	1	16	802.11n	2	1944	1/2	Struct.	TDMP	MSA	11.92	83.50	10	GeForce GTX 480
				802.16e		2304					10.29	114.60		
[160]	TpN	1	N/A	Regular	2	10008	1/2	Sparse	TPMP	SPA	0.06	N/A	1	N/A
						100000					0.17			
				QC [325]		15000	5/6	Struct.			0.21	2.38		
						18000					0.22	2.81		
[161]	TpC	N/A	N/A	QC-CC Proposed	2	768–1536	1/2–2/3	Struct.	TPMP	NMSA	N/A	15	20	GeForce GTX 260
[162]	TpN	N/A	N/A	RA	2	20000	1/2	Sparse	TPMP	LSPA	N/A	29.3	1.45 dB	GeForce GTX 280

Continued on next page

Table A.1 *Continued from previous page*

Work	Parallelism		Data	LDPC Code					Decoding		Perf.			Platform
	Thread-	Data-	Width	Code	q	N	Rate	Tanner	Sched.	Alg.	Latency (ms)	Thr. (Mbit/s)	Iter.	Name
[163]	TpN	N/A	32	QC	2	10128	5/6	Struct.	TPMP	LSPA	3.80	N/A	3.2 dB	GeForce GTX 460
						15168					5.00			
						18630					5.30			
						24576					7.90			
						18000	1/2				3.20		2.3 dB	
[164]	TpN	1	8–10	DVB-S2	2	64800	5/6	Sparse	TPMP	SPA	N/A	77.06	5.9	GeForce GTX Titan
						16200						117.81	7.4	
				802.11n		1944						92.78	3.5	
				802.16e		2304						102.07	3.6	
				DVB-S2		64800	3/4					135.05	3.6	
						16200						191.06	3.0	
				802.11n		1944						138.14	2.2	
				802.16e		2304	3/4A					164.75	2.2	
							3/4B					161.00	2.2	
				DVB-S2		64800	2/3					162.66	3.0	
						16200						178.09	2.5	
				802.11n		1944						163.09	1.8	
				802.16e		2304	2/3A					96.36	1.8	
							2/3B					196.67	1.8	
				DVB-S2		64800	3/5					121.96	2.3	
						16200						116.61	2.1	
		64		802.16e		576	1/2			OMSA	0.12	310		
		128								NMSA	0.13	560		
		64				1248	1/2			OMSA	0.26	310		
		128								NMSA	0.28	564		
		64		802.11n		1944	5/6			OMSA	0.42	294		
		128								NMSA	0.53	473		

Continued on next page

[165] CpC 8 2 Struct. TDMP 10 i7 4960 HQ

Table A.1 *Continued from previous page*

Work	Parallelism		Data	LDPC Code					Decoding		Perf.			Platform
	Thread-	Data-	Width	Code	q	N	Rate	Tanner	Sched.	Alg.	Latency (ms)	Thr. (Mbit/s)	Iter.	Name
		64				1944	1/2			OMSA	0.46	270		
		128								NMSA	0.55	456		
		64		802.11an		2048	13/16			OMSA	0.78	168		
		128								NMSA	1.14	230		
		64		802.16e		2304	1/2			OMSA	0.47	314		
		128								NMSA	0.55	533		
		64		Mackay		4000				OMSA	0.77	334		
		128								NMSA	0.99	519		
		64				8000				OMSA	1.58	323		
		128								NMSA	2.30	445		
		64		CMMB		9216				OMSA	1.78	332		
		128								NMSA	2.35	503		
		64				9216	3/4			OMSA	.175	337		
		128								NMSA	2.40	493		
		64		DVB-S2		16200				OMSA	4.89	212		
		128								NMSA	8.36	248		
		64		Mackay		20000	1/2			OMSA	5.09	251		
		128								NMSA	7.57	338		
		64		DVB-S2		64800				OMSA	19.02	218		
		128								NMSA	34.74	242		

[1)] obtained at 5dB for an undetermined number of decoding iterations; [2)] obtained at 3.39 dB and -0.58 dB for an undetermined number of decoding iterations;

[3)] - AWGN and BER simulation modules included; [4)] - absolute throughput not known, authors study the scalability of the Intel SCC system for LDPC decoding

[5)] - when used for the QC-LDPC 2304 1/2 for reference; [6)] - maximum of 10 iterations but early termination is employed;

[7)] - best is for reordered scheduling for minimization of the pipeline stalling; LDPCA: Coastguard (Cg); Foreman (Fm); Hall Monitor (HM); and Soccer (Sc)

[{a,b,c,d})] - a) naive *iterative parity-check* (IPC); b) IPC with stability check; c) IPC with confirmation; d) standard schedule;

The resulting order of references is the one that minimizes the table size and prevents too many page breaks.

B

Galois Field Arithmetic

Contents

Herein, the concepts regarding finite field algebra are covered, briefly for groups and rings and special focus is given to Galois fields, since they provide the underlying concepts upon which the construction of non-binary codes are based[47,58,326,327].

Groups A set contains any number of elements or objects. It has no conditions imposed on it and its dimension can be finite or infinite. The number of elements in a set is defined as its cardinality. We can apply two binary operations to elements of a set: multiplication "$\cdot$"; and addition "$+$". Conditions can then be applied to the set under such binary operations

- *Commutatity:* For two elements a and b, $a \cdot b = b \cdot a$ under multiplication or $a + b = b + a$ under addition.
- *Identity:* For any element a in the set there is an element b such that $a \cdot b = a$ under multiplication and $a + b = a$ under addition.
- *Inverse:* For any element a in the set its inverse a^{-1} must also be in the set, whereupon $a \cdot a^{-1} = a^{-1} \cdot a = b$, the identity element.
- *Associativity:* For the elements a, b and c in the set, $(a \cdot b) \cdot c = a \cdot (b \cdot c)$.
- *Distributivity:* For the elements a, b and c in the set, $a \cdot (b + c) = a \cdot b + a \cdot c$.

A group is defined as a set with the multiplication operation. In groups with closure, the result of the multiplication of any two elements in the set must result in a third element belonging also to the set. A group is defined by the following properties

- associativity under multiplication,
- identity under multiplication
- inverse.

Groups that also possess commutativity under multiplication are designated commutative or abelian groups.

Rings If the binary operations addition and multiplication are allowed, we may define a ring that follows the conditions

- associativity,
- distributivity,
- commutativity under addition.

Commutative rings also possess commutativity under multiplication. If it possesses a multiplicative identity 1 then it is designated a ring with identity. An example of a ring is the ring of integers $\mathbb{Z}_q$, under modulo-q addition and multiplication, where q is the ring cardinality. The following tables present the addition and multiplication tables for the ring $\mathbb{Z}_8$.

Table B.1: Addition and multiplication table for the $\mathbb{Z}_8$ ring.

+	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	·	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	0	1	2	3	4	5	6	7	-	0	0	0	0	0	0	0	0
1	1	2	3	4	5	6	7	0	-	0	1	2	3	4	5	6	7
2	2	3	4	5	6	7	0	1	-	0	2	4	6	0	2	4	6
3	3	4	5	6	7	0	1	2	-	0	3	6	1	4	7	2	5
4	4	5	6	7	0	1	2	3	-	0	4	0	4	0	4	0	4
5	5	6	7	0	1	2	3	4	-	0	5	2	7	4	1	6	3
6	6	7	0	1	2	3	4	5	-	0	6	4	2	0	6	4	2
7	7	0	1	2	3	4	5	6	-	0	7	6	5	4	3	2	1

It can be easily seen that the elements in $\mathbb{Z}_8$ obey the three properties of a ring. In addition, all the elements commute under multiplication and the multiplicative identity 1 is present, thus $\mathbb{Z}_8$ is a commutative ring with identity.

B.1 Fields

A field is similar to a ring, as it uses both binary operations. However, its definition has the following extended conditions

- commutativity under addition,
- distributivity,
- commutativity under multiplication when the additive identity element 0 is removed,
- identity,
- inverse.

An example of a field is the set of the real number $\mathbb{R}$. The set of all integers $\mathbb{Z}$ is not a field because not all integers have a multiplicative inverse.

In the case that the field has a finite number of elements it is designated by Galois Field and denoted by GF(q) where q is the cardinality of the field and is either a prime number or a power of a prime number greater than 1. A field requires that the set of elements in the field $\{1, 2, 3, \cdots q - 1\}$ form a group under multiplication modulo-q. For instance, GF(5)=$\{0, 1, 2, 3, 4\}$ is a finite field because the elements $\{1, 2, 3, 4\}$ are a group under modulo-q multiplication. The group also has closure since any two elements multiplied together produce an element inside the group, as seen in Table B.2.

Table B.2: Multiplication table for non-zero elements in GF(5).

$\cdot$	1	2	3	4
1	1	2	3	4
2	2	4	1	3
3	3	1	4	2
4	4	3	2	1

If we consider GF(6)=$\{0, 1, 2, 3, 4, 5\}$ as an hypothetical field, the set $\{1, 2, 3, 4, 5\}$ must form a group, which is not the case as seen in Table B.3.

Table B.3: Multiplication table of the set $\{1, 2, 3, 4, 5\}$.

$\cdot$	1	2	3	4	5
1	1	2	3	4	5
2	2	4	**0**	2	4
3	3	**0**	3	**0**	3
4	4	2	**0**	4	2
5	5	4	3	2	1

By not complying to the group condition, we discard GF(6) as a finite field.

Another property noteworthy is the order of an element, which defines the number of times an element has to be multiplied by itself before it produces the identity 1, formally the order m is such that the m-th power of the element β is 1, $\beta^m = 1$. Moreover, in finite fields the order of an element divides $q - 1$. Table B.4 portrays the order of each of the non-zero elements in GF(5).

Table B.4: Order of each non-zero element in GF(5).

Element in GF(5)	Order
1	1
2	4
3	4
4	2

Every element in GF(5) has an order which is dividable by $q - 1 = 4$, and elements whose order matches $q - 1$ are called primitive elements, because every element in the field may be produced by powers of such elements. In the GF(5) case, the field may be constructed resorting to the primitive element 2 or to the primitive element 3. The resulting field is depicted in Table B.5. Finite fields can also be of the form GF(p^m). In this case, they are designated extension fields, and contain the elements $\{0, 1, \alpha, \alpha^2, \alpha^3, \cdots, \alpha^{p^m-2}\}$, where α is the primitive element with order $p^m - 1$. This means all elements in the finite field may be represented as powers of α. Each element in GF(p^m) can be expressed as a m-tuple vector with elements from GF(p) and addition between elements is done modulo-p.

Table B.5: Defining GF(5) with primitive elements 2 and 3.

Powers of 2	Element in GF(5)	Powers of 3	Element in GF(5)
2^1	2	3^1	3
2^2	4	3^2	4
2^3	3	3^3	2
2^4	1	3^4	1

B.1.1 Primitive Polynomials

Extension fields can be defined through a class of polynomials designated primitive polynomials. An irreducible polynomial $f(x)$ with degree m defined over GF(p) is primitive if the smallest positive integer n for which $f(x)$ divides $x^n - 1$ is $n = p^m - 1$ [47,58,326,327].

For instance, the irreducible polynomial

$$f(x) = x^4 + x + 1, \tag{B.1}$$

with $p = 2$, is primitive if it divides $x^{2^4-1}+1 = x^{16-1}+1 = x^{15}+1$, since $-1 = 1$ in modulo-2. It can be shown[47], that the polynomial in (B.1) indeed divides $x^{15}+1$, and thus $f(x) = x^4+x+1$ is a primitive polynomial to GF(2^4) since it also does not divide any x^n+1 with $0 \leq n < 15$. Thus, we can define a primitive element α as the root of $f(x)$, and express higher powers of α as the sum of lower powers of α, derived from $f(x)$. Namely, if α is the root of $f(x)$, then $\alpha^4 = \alpha+1$, $\alpha^5 = \alpha^2+\alpha$, $\alpha^6 = \alpha^3+\alpha^2$, $\alpha^7 = \alpha^4+\alpha^3 = \alpha^3+\alpha+1$ and so on. Table B.6 depicts the construction of GF(2^3) built over the primitive polynomial $f(x) = x^3+x+1$, whose primitive element is α, the root of $f(x)$.

Table B.6: Constructing GF(2^3) using $f(x) = x^3+x+1$ as the primitive polynomial.

Element in GF(2^3)	Combination of lower powers of α	Element as 3-tuple vector over GF(2)
0	0	000
1	1	001
α	α	010
α^2	α^2	100
α^3	$\alpha+1$	011
α^4	$\alpha^2+\alpha$	110
α^5	$\alpha^3+\alpha^2 = \alpha+1+\alpha^2$	111
α^6	$\alpha^4+\alpha^3 = \alpha^2+\alpha+\alpha+1 = \alpha^2+1$	101

The m-tuple representation is defined by sum of the powers of α, up to the m-th power, generating the field's element. For instance, $\alpha^6 = \alpha^2+1$ so its tuple representation is 101. The closure property can be proved by writing

$$\begin{aligned}\alpha^7 &= \alpha^6\cdot\alpha = (\alpha^2+1)\cdot\alpha\\ &= \alpha^3+\alpha = \alpha+1+\alpha\\ &= 1,\end{aligned}\tag{B.2}$$

from which it can be generalized that $\alpha^{c(q-1)} = 1$ whenever c is a non-negative integer. Addition in the extension field can be accomplished through the modulo-p sum of the m-tuple representation of the elements. Multiplication is accomplished by modulo-p addition of the powers of the multiplying elements. For this field in particular, GF(2^3) the addition and multiplication tables are shown in Table B.7.

Table B.7: Addition and multiplication table for the extension field GF(2^3).

$+$	0	1	α	α^2	α^3	α^4	α^5	α^6	$\cdot$	0	1	α	α^2	α^3	α^4	α^5	α^6
0	0	1	α	α^2	α^3	α^4	α^5	α^6		0	0	0	0	0	0	0	0
1	1	0	α^3	α^6	α	α^5	α^4	α^2		0	1	α	α^2	α^3	α^4	α^5	α^6
α	α	α^3	0	α^4	1	α^2	α^6	α^4		0	α	α^2	α^3	α^4	α^5	α^6	1
α^2	α^2	α^6	α^4	0	α^5	α	α^3	1		0	α^2	α^3	α^4	α^5	α^6	1	α
α^3	α^3	α	1	α^5	0	α^6	α^2	α^4		0	α^3	α^4	α^5	α^6	1	α	α^2
α^4	α^4	α^5	α^2	α	α^6	0	1	α^3		0	α^4	α^5	α^6	1	α	α^2	α^3
α^5	α^5	α^4	α^6	α^3	α^2	1	0	α		0	α^5	α^6	1	α	α^2	α^3	α^4
α^6	α^6	α^2	α^5	1	α^4	α^3	α	0		0	α^6	1	α	α^2	α^3	α^4	α^5

B.1.2 Matrix-Representation of Fields

For the polynomial $a(x) = a_0 + a_1x + a_2x^2 + \cdots + a_{r-1}x^{r-1} + x^r$ corresponds a $r \times r$ matrix M, designated the companion matrix whose characteristic polynomial, i.e. $\det(\mathbf{M} - \lambda\mathbf{I}) = a(\lambda)$.

$$\mathbf{M} = \begin{bmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -a_0 & -a_1 & -a_2 & \cdots & -a_{r-1} \end{bmatrix} \tag{B.3}$$

The properties drawn regarding the generation of finite field elements as the powers of a primitive element can be proven likewise for the companion matrix representation.

Thus, $\mathbf{M}^i = \mathbf{I}$ if $i = p^m - 1$, and $\mathbf{M}^i \neq \mathbf{I}$ for $1 \leq i < q^m - 1$. Under the latter set of powers, M^i defines the elements of the finite field GF(p^m). For instance, the elements in GF(2^3), generated by the primitive polynomial $f(x) = x^3 + x + 1$ can be represented as

$$\overbrace{\begin{bmatrix} 000 \\ 000 \\ 000 \end{bmatrix}}^{0} \overbrace{\begin{bmatrix} 010 \\ 001 \\ 110 \end{bmatrix}}^{\mathbf{M}} \overbrace{\begin{bmatrix} 001 \\ 110 \\ 011 \end{bmatrix}}^{\mathbf{M}^2} \overbrace{\begin{bmatrix} 110 \\ 011 \\ 111 \end{bmatrix}}^{\mathbf{M}^3} \overbrace{\begin{bmatrix} 011 \\ 111 \\ 101 \end{bmatrix}}^{\mathbf{M}^4} \overbrace{\begin{bmatrix} 111 \\ 101 \\ 100 \end{bmatrix}}^{\mathbf{M}^5} \overbrace{\begin{bmatrix} 101 \\ 100 \\ 010 \end{bmatrix}}^{\mathbf{M}^6} \overbrace{\begin{bmatrix} 100 \\ 010 \\ 001 \end{bmatrix}}^{\mathbf{M}^7} \tag{B.4}$$

B.1.3 Fourier Transform

In the particular case of *binary extension field* (GF(2^m)), the *discrete Fourier transform* (DFT) is not 2^m point but a m-dimension two-point DFT[28,47] in GF(2^m). In fact, the DFT of x, $\mathsf{F}(x) = \mathsf{W} \cdot x$, where W is the Hadamard matrix. Hadamard matrices[328,329] may be defined as

$$\mathsf{W}_n\mathsf{W}_n^{\mathsf{T}} = n\mathbf{I}_n, \tag{B.5}$$

or alternatively through the recursion

$$\begin{aligned} \mathsf{W}_1 &= \begin{bmatrix} 1 \end{bmatrix} \quad \mathsf{W}_2 = \begin{bmatrix} \mathsf{W}_1 & \mathsf{W}_1 \\ \mathsf{W}_1 & -\mathsf{W}_1 \end{bmatrix} \\ \mathsf{W}_m &= \begin{bmatrix} \mathsf{W}_{m-1} & \mathsf{W}_{m-1} \\ \mathsf{W}_{m-1} & -\mathsf{W}_{m-1} \end{bmatrix}, \end{aligned} \tag{B.6}$$

where m is the order of the binary extension field GF(2^m). Yet another alternative to efficiently compute the Hadamard matrix is to generate its elements on the fly, based solely on their positions within the matrix

$$w_{ij} = (-1)^{\widehat{i \cdot j}}, \tag{B.7}$$

where $\widehat{\cdot}$ denotes the Hamming weight and $\cdot$ the binary multiplication operand.

An interesting property, later on of value to the decoding algorithms of non-binary *low-density parity-check* (LDPC) codes, is that we may write $A^i\alpha^j = \alpha^i\alpha^j = \alpha^{i+j} = A^{i+j}$. Thus, for $x \in$ GF(2^m), $A^i \times x = a^i \times x$, where A^i and a^i are elements in GF(2^m).

Definition 1. *Let $\tilde{p}(x) = p(a^{-1}x)$ and $\tilde{P}(x) = P(A^{-1}x)$ and $P(\cdot)$ the companion matrix of $p(\cdot)$. The Fourier transform of $\tilde{p}(x)$ yields the Fourier transform of $\tilde{P}(x)$: $\mathsf{F}\left(\tilde{p}(x)\right) = \mathsf{F}\left(\tilde{P}(x)\right)$.*

Proof. By using the Fourier Transform definition

$$\begin{aligned} \mathsf{F}(\tilde{p}(x)) = \tilde{p}(z) &= \sum_{x \in \mathrm{GF}(2^m)} \tilde{p}(x)(-1)^{z \cdot x} = \sum_{x \in \mathrm{GF}(2^m)} p(a^{-1}x)(-1)^{z \cdot x} \\ &= \sum_{x \in \mathrm{GF}(2^m)} p(x)(-1)^{z \cdot (Ax)} = \sum_{x \in \mathrm{GF}(2^m)} p(x)(-1)^{(A^{\mathsf{T}}z) \cdot x} \\ &= P(A^{\mathsf{T}}z) = \mathsf{F}(\tilde{P}(x)). \end{aligned}$$

C

List of Hardware Employed

C. List of Hardware Employed

Table C.1: List of utilized hardware—CPU, GPU, FPGA and power sensor apparatus—throughout the execution of the Thesis. Keywords with which each architecture is referred, henceforth in the text, are annotated in the leftmost column.

Utilized Hardware						
GPU	Name	Stream Processors	SM/SMX	Compute Capability	Memory (MB)	Clock (MHz)
G1	Tesla C1060	240	30	1.3	4096	1296
G2	Tesla C2050	448	14	2.0	3071	1147
G3	Tesla M2050					
G4	Tesla K20c	2496	13	3.5	5120	706
G5	Tesla K40c	2880	15	3.5	11520	745
G6	GeForce GTX 680	1536	8	3.0	2048	1006
G7	GeForce GTX Titan	2688	14	3.5	6143	837
G8	Radeon HD 5870	1600	N/A	N/A	2048	850
G9	GeForce GTX 560 Ti	384	8	2.1	1019	1660
CPU	Name	Logical Cores	L1 (KB)	L2 (KB)	L3 (KB)	Clock (MHz)
C1	Phenom II X4 945	4	128	512	6144	3000
C2	i7 3770K	8	128	1024	8192	3500
C3	Xeon E5645	12	192	1536	12288	2400
FPGA	Name	Gen. Logic Element	Logic Elements	BRAM Blocks	DSP Blocks	DRAM (MB)
F1	Virtex 6 LX760	*a*)	118560	720[1)]	864[3)]	N/A
F2	Virtex 5 LX330T MAX2336B	*b*)	51840	324[1)]	192[3)]	6×2048
F3	Virtex 6 SX475T MAX3XXXB	*a*)	74400	1064[1)]	2016[3)]	6×2048
F4	Stratix V D5 Nallatech 385 N	*c*)	172600	2014[2)]	3180[4)]	2×4096
F5	Virtex 7 VX690T VC 709	*d*)	693120	1470[1)]	3600[3)]	2×4096
GPU Cluster	Type	CPU	GPU	Memory	MPI	Interconnection
Master Node	N/A	Intel i7 920	Tesla M2050	12 GB	OpenMPI 1.4.2	1 GBit/s Ethernet 40 Gbit/s IB QDR
Compute Nodes	8× NEC LX 113Rc-1G 8× NEC LX 116Rc-1G	Xeon E5645 Xeon E5645	Tesla M2050			
Power Sensor	Allegro ACS712 Hall effect current sensor; PIC18F4550					

a) - Slice with four LUTs and eight flip-flops.; *b*) - Slice with four LUTs and four flip-flops.

c) - *adaptive logic modules* (ALMs) with six LUTs and two flip-flops; *d*) - CLB with eight LUTs and sixteen flip-flops.

[1)] - 36Kb BRAM block; [2)] - 20Kb M20K BRAM block; [3)] - DSP48E, 25x18 multiplier; [4)] - 18x18 multiplier.

Bibliography

[1] J. Andrade, G. Falcao, V. Silva, S. Yamagiwa, and L. Sousa, *Encyclopedia of Computer Science and Technology*. Taylor & Francis, 2015, ch. Accelerating Conventional Processing Using GPU Clusters: LDPC Decoders.

[2] G. Falcao, J. Andrade, V. Silva, S. Yamagiwa, and Sousa, "Stressing the BER simulation of LDPC codes in the error floor region using GPU clusters," in *International Symposium on Wireless Communication Systems (ISWCS 2013)*, Aug 2013, pp. 1–5.

[3] J. Andrade, G. Falcao, V. Silva, J. Barreto, N. Goncalves, and V. Savin, "Near-LSPA performance at MSA complexity," in *Communications (ICC), 2013 IEEE International Conference on*, June 2013, pp. 3281–3285.

[4] J. Andrade, G. Falcao, V. Silva, and K. Kasai, "FFT-SPA Non-binary LDPC Decoding on GPU," in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, May 2013, pp. 5099–5103.

[5] J. Andrade, G. Falcao, V. Silva, M. Owaida, N. Bellas, C.D. Antonopoulos, and P. Ienne, "Towards High-Throughput with Low-Effort Programming: From General-Purpose Manycores to Dedicated Circuits," in *DATE'13: Workshop on Designing for Embedded Parallel Computing Platforms: Architectures, Design Tools, and Applications (DEPCP)*, March 2013.

[6] J. Andrade, F. Pratas, G. Falcao, V. Silva, and L. Sousa, "Combining flexibility with low power: Dataflow and wide-pipeline LDPC decoding engines in the Gbit/s era," in *Application-specific Systems, Architectures and Processors (ASAP), 2014 IEEE 25th International Conference on*, June 2014, pp. 264–269.

[7] J. Andrade, G. Falcao, and V. Silva, "Flexible design of wide-pipeline-based WiMAX QC-LDPC decoder architectures on FPGAs using high-level synthesis," *Electronics Letters*, vol. 50, no. 11, pp. 839–840, 2014.

[8] J. Andrade, N. George, K. Karras, D. Novo, V. Silva, P. Ienne, and G. Falcao, "Fast Design Space Exploration Using Vivado HLS: Non-binary LDPC Decoders," in *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, May 2015, pp. 97–97.

[9] J. Andrade, N. George, K. Karras, D. Novo, V. Silva, P. Ienne, and G. Falcao, "From low-architectural expertise up to high-throughput non-binary ldpc decoders: Optimization guidelines using high-level synthesis," in *Field Programmable Logic and Applications (FPL), 2015 25th International Conference on*, Sept 2015, pp. 1–8.

[10] F. Pratas, J. Andrade, G. Falcao, V. Silva, and L. Sousa, "Open the Gates: Using High-level Synthesis towards programmable LDPC decoders on FPGAs," in *Global Conference on Signal and Information Processing (GlobalSIP), 2013 IEEE*, Dec 2013, pp. 1274–1277.

[11] J. Andrade, G. Falcao, V. Silva, and K. Kasai, "Flexible non-binary LDPC decoding on FPGAs," in *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*. IEEE, May 2014, pp. 1936–1940.

[12] M. Owaida, G. Falcao, J. Andrade, C. Antonopoulos, N. Bellas, M. Purnaprajna, D. Novo, G. Karakonstantis, A. Burg, and P. Ienne, "Enhancing Design Space Exploration by Extending CPU/GPU Specifications Onto FPGAs," *ACM Trans. Embed. Comput. Syst.*, vol. 14, no. 2, pp. 33:1–33:23, Feb. 2015.

[13] J. Andrade, G. Falcao, and V. Silva, “Optimized Fast Walsh–Hadamard Transform on GPUs for non-binary LDPC decoding,” *Parallel Computing*, vol. 40, no. 9, pp. 449 – 453, 2014.

[14] J. Andrade, V. Silva, and G. Falcao, “From OpenCL to gates: The FFT,” in *Global Conference on Signal and Information Processing (GlobalSIP), 2013 IEEE*, Dec 2013, pp. 1238–1241.

[15] J. Andrade, G. Falcao, and V. Silva, “Accelerating and Decelerating Min-Sum-based Gear-shift LDPC Decoders,” in *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*, April 2015, pp. 5099–5103.

[16] J. Andrade, A. Vosoughi, G. Wang, G. Karakonstantis, A. Burg, G. Falcao, V. Silva, and J. Cavallaro, “On the performance of LDPC and turbo decoder architectures with unreliable memories,” in *Signals, Systems and Computers, 2014 48th Asilomar Conference on*, Nov 2014, pp. 542–547.

[17] J. Mu, A. Vosoughi, J. Andrade, A. Balatsoukas-Stimming, G. Karakonstantis, A. Burg, G. Falcao, V. Silva, and J. Cavallaro, “The Impact of Faulty Memory Bit Cells on the Decoding of Spatially-Coupled LDPC Codes,” in *Signals, Systems and Computers, 2015 49th Asilomar Conference on*, 2015.

[18] A. A. Santos and J. Andrade, “Stochastic Volatility Estimation with GPU Computing,” in *Proc Conf. on Indirect Estimation Methods in Finance and Economics*, May 2014.

[19] R. Ralha, G. Falcao, J. Andrade, M. Antunes, J. Barreto, and U. Nunes, “Distributed Dense Stereo Matching for 3D Reconstruction using Parallel-based Processing Advantages,” in *ICASSP’15: Proceedings of the 40th IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, April 2015.

[20] R. G. Gallager, “Low-Density Parity-Check Codes,” *IRE Transactions on Information Theory*, vol. 8, no. 1, pp. 21–28, jan 1962.

[21] R. G. Gallager, *Low-Density Parity-Check Codes*. Cambridge, MA: MIT Press, 1962.

[22] V. V. Zyablov, “An Estimation of the Complexity of Constructing Binary Linear Cascading Codes,” *Problems Inf. Theory*, vol. 7, pp. 3–10, 1971.

[23] M. Tanner, “A Recursive Approach to Low Complexity Codes,” *IEEE Transactions on Information Theory*, 1981.

[24] N. Wiberg, “Codes and decoding on general graphs,” Ph.D. dissertation, University of Liköping, 1996.

[25] B. Sklar and F. Harris, “The ABCs of Linear Block Codes,” *IEEE Signal Processing Magazine*, vol. 21, no. 4, pp. 14–35, 2004.

[26] J. Chen and M. Fossorier, “Near optimum universal belief propagation based decoding of low-density parity check codes,” *Communications, IEEE Transactions on*, vol. 50, no. 3, pp. 406 – 414, 2003.

[27] M. A. S. T.J. Richardson and R. L. Urbanke, “Design of Capacity-Approaching Low-Density Parity Check Codes,” *IEEE Trans. Inform. Theory*, vol. 47, pp. 619–637, Feb. 2001.

[28] T. J. Richardson and R. L. Urbanke, “The capacity of low-density parity-check codes under message-passing decoding,” *Information Theory, IEEE Transactions on*, vol. 47, no. 2, pp. 599–618, Feb. 2001.

[29] C. E. Shannon, “A mathematical theory of communication,” *Bell System Technical Journal*, vol. 27, pp. 379–423, 1948.

[30] C. Berrou, A. Glavieux, and P. Thitimajshima, “Near Shannon Limit Error-Correcting Coding and Decoding: Turbo-Codes,” in *IEEE International Conference on Communication*, May 1993.

[31] The Consultative Committee for Space Data Systems, “Low Density Parity Check Codes for Use in Near-Earth and Deep Space Applications,” *Orange Book, Issue 2, Consulative Committee for Space Data Systems (CCSDS) Experimental Specification 131.1-O-2*, 2007.

[32] “IEEE Standard for Information Technology - Telecommunications and Information Exchange Between Systems - Local and Metropolitan Area Networks - Specific Requirements Part 3: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications- Corrigendum 2: IEEE Std 802.3an-2006 10GBASE-T Correction,” *IEEE Std 802.3-2005/Cor 2-2007 (Corrigendum to IEEE Std 802.3-2005)*, pp. 1–2, Aug 2007.

[33] "IEEE Standard for Information technology– Local and metropolitan area networks– Specific requirements– Part 11: Wireless LAN Medium Access Control (MAC)and Physical Layer (PHY) Specifications Amendment 5: Enhancements for Higher Throughput," *IEEE Std 802.11n-2009 (Amendment to IEEE Std 802.11-2007 as amended by IEEE Std 802.11k-2008, IEEE Std 802.11r-2008, IEEE Std 802.11y-2008, and IEEE Std 802.11w-2009)*, pp. 1–565, Oct 2009.

[34] "IEEE Standard for Information technology - Telecommunications and information exchange between systems - Local and metropolitan area networks - Specific requirements. Part 15.3: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for High Rate Wireless Personal Area Networks (WPANs) Amendment 2: Millimeter-wave-based Alternative Physical Layer Extension," *IEEE Std 802.15.3c-2009 (Amendment to IEEE Std 802.15.3-2003)*, Oct 2009.

[35] "IEEE Standard for Air Interface for Broadband Wireless Access Systems," *IEEE Std 802.16-2012*, Aug 2012.

[36] EN 302 307 V1.1.1, European Telecommunications Standards Institute (ETSI), "Second generation framing structure, channel coding and modulation systems for broadcasting, interactive services, news gathering and other broad-band satellite applications," Digital Video Broadcasting (DVB); , 2005.

[37] EN 302 755 V1.1.1, European Telecommunications Standards Institute (ETSI), " Frame structure channel coding and modulation for a second generation digital terrestrial television broadcasting system (DVB-T2)," Digital Video Broadcasting (DVB);, 2009.

[38] EN 302 769 V1.1.1, European Telecommunications Standards Institute (ETSI), "Frame structure channel coding and modulation for a second generation digital transmission system for cable systems (DVB-C2)," Digital Video Broadcasting (DVB); , 2010.

[39] EN 301 545-2 V1.1.1, European Telecommunications Standards Institute (ETSI), " Part 2: Lower Layers for Satellite standard," Digital Video Broadcasting (DVB); Second Generation DVB Interactive Satellite System (DVB-RCS2);, 2011.

[40] ITU-T G.9960, Telecommunication Standardization Sector of ITU, "Unified high-speed wireline-based home networking transceivers – System architecture and physical layer specification," Series G: Transmission Systems and Media, Digital Systems and Networks, 2012.

[41] ITU-T G.709, Telecommunication Standardization Sector of ITU, "Interfaces for the optical transport network," Series G: Transmission Systems and Media, Digital Systems and Networks, 2012.

[42] The 3rd Generation Partnership Project (3GPP), "Evolved universal terrestrial radio access (E-UTRA); multiplexing and channel coding, Tech. Spec. 36.212 Release-11," Dec 2012.

[43] J. Andrews, S. Buzzi, W. Choi, S. Hanly, A. Lozano, A. Soong, and J. Zhang, "What Will 5G Be?" *Selected Areas in Communications, IEEE Journal on*, vol. 32, no. 6, pp. 1065–1082, June 2014.

[44] J. Sayir, "Advances in Coding Algorithms for 5G," 2015. [Online]. Available: http://montecristo.co.it.pt/ja_thesis/Newcom.pdf

[45] J. Kliewer, "Advances in Error Control Strategies for 5G." [Online]. Available: http://montecristo.co.it.pt/ja_thesis/Joerg_Kliewer.pdf

[46] M. C. Davey and D. MacKay, "Low-density parity check codes over GF(q)," *Communications Letters, IEEE*, vol. 2, no. 6, pp. 165–167, Jun. 1998.

[47] R. A. Carrasco and M. Johnston, *Non-Binary Error Control Coding for Wireless Communication and Data Storage*. Wiley, Chichester, 2008.

[48] M. Lentmaier, M. Prenda, and G. Fettweis, "Efficient message passing scheduling for terminated LDPC convolutional codes," in *Information Theory Proceedings (ISIT), 2011 IEEE International Symposium on*, July 2011, pp. 1826–1830.

[49] S. Abu-Surra, E. Pisek, and R. Taori, "Spatially-coupled low-density parity check codes: Zigzag-window decoding and code-family design considerations," in *2015 Information Theory and Applications Workshop*, 2015.

[50] D. J. Costello, L. Dolecek, T. Fuja, J. Kliewer, D. Mitchell, and R. Smarandache, "Spatially coupled sparse codes on graphs: theory and practice," *Communications Magazine, IEEE*, vol. 52, no. 7, pp. 168–176, July 2014.

[51] G. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 8, 1965. [Online]. Available: http://www.cs.utexas.edu/users/fussell/courses/cs352h/papers/moore.pdf

[52] G. Moore, "Progress In Digital Integrated Electronics," 2015, IEEE Text Speech. [Online]. Available: http://montecristo.co.it.pt/ja_thesis/Gordon_Moore_1975_Speech.pdf

[53] R. Dennard, V. Rideout, E. Bassous, and A. LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions," *Solid-State Circuits, IEEE Journal of*, vol. 9, no. 5, pp. 256–268, Oct 1974.

[54] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2006.

[55] J. Nickolls and W. Dally, "The gpu computing era," *Micro, IEEE*, vol. 30, no. 2, pp. 56–69, March 2010.

[56] R. Tessier, K. Pocek, and A. DeHon, "Reconfigurable Computing Architectures," *Proceedings of the IEEE*, vol. 103, no. 3, pp. 332–354, March 2015.

[57] M. Maxfield, "Google's project ARA smartphones to use lattice ECP5 FPGAs," *EE Times*, April 2014.

[58] S. Lin and D. Costello, *Error control coding*. Prentice-Hall Englewood Cliffs, NJ, 1983.

[59] T. Richardson and F. Technologies, "The renaissance of Gallager's low-density parity-check codes," *Communications Magazine, IEEE*, no. August, pp. 126–131, 2003.

[60] D. MacKay, "Good error-correcting codes based on very sparse matrices," in *Information Theory. 1997. Proceedings., 1997 IEEE International Symposium on*, jun-4 jul 1997, p. 113.

[61] A. Morello and V. Mignone, "DVB-S2: The Second Generation Standard for Satellite Broad-Band Services," *Proceedings of the IEEE*, vol. 94, no. 1, 2006.

[62] J. Thorpe, "Design of LDPC graphs for hardware implementation," in *Information Theory, 2002. Proceedings. 2002 IEEE International Symposium on*, 2002, p. 483.

[63] *IEEE 802.11 Wireless LANsWWiSE Proposal: High Throughput extension to the 802.11 Standard*, IEEE 11-04-0886-00-000n.

[64] *IEEE 802.11n Wireless LAN Medium Access Control MAC and Physical Layer PHY specifications*, IEEE 802.11n-D1.0, 2006.

[65] T. Richardson, M. Shokrollahi, and R. Urbanke, "Design of capacity-approaching irregular low-density parity-check codes," *Information Theory, IEEE Transactions on*, vol. 47, no. 2, pp. 619 –637, feb 2001.

[66] S.-Y. Chung, J. Forney G.D., T. J. Richardson, and R. Urbanke, "On the design of low-density parity-check codes within 0.0045 dB of the Shannon limit," *Communications Letters, IEEE*, vol. 5, no. 2, pp. 58–60, Feb. 2001.

[67] S.-Y. Chung, "On the construction of some capacity-approaching coding schemes," MIT, Tech. Rep., 2000.

[68] R. G. Gallager, "Low-Density Parity-Check Codes," *IRE Transactions on Information Theory*, vol. 8, no. 1, pp. 21–28, jan 1962.

[69] R. G. Gallager, *Low-Density Parity-Check Codes*. MIT Press, Cambridge, 1963.

[70] V. V. Zyablov and M. S. Pinkser, "Estimation of the error-correction complexity for Gallager low-density codes," *Problems Inf. Theory*, vol. 11, no. 18-28, 1976.

[71] G. A. Margulis, "Explicit Construction of Graphs Without Short Cycles," *Combinatorica*, vol. 2, pp. 71–78, 1982.

[72] M. Tanner, "A Recursive Approach to Low Complexity Codes," *IEEE Transactions on Information Theory*, 1981.

[73] N. Wiberg, "Codes and iterative decoding on general graphs," *European Transactions on Telecommunications and Related Technologies*, pp. 512–525, 1995.

[74] N. Wiberg, H.-a. Loeliger, and R. Kotter, "Codes and iterative decoding on general graphs," *Proceedings of 1995 IEEE International Symposium on Information Theory*, p. 468, 1995.

[75] D. J. C. MacKay and R. M. Neal, "Near Shannon limit performance of low density parity check codes," *Electronics Letters*, vol. 32, no. 18, p. 1645, Aug. 1996.

[76] D. MacKay and R. Neal, "Near shannon limit performance of low density parity check codes," *Electronics Letters (Reprint)*, vol. 33, pp. 457–458, mar 1997.

[77] D. MacKay, "Good error-correcting codes based on very sparse matrices," *IEEE Transactions on Information Theory*, vol. 45, no. 2, pp. 399–431, Mar. 1999.

[78] S. Papaharalabos, P. Sweeney, B. G. Evans, P. T. Mathiopoulos, G. Albertazzi, A. Vanelli-Coralli, and G. E. Corazza, "Modified sum-product algorithms for decoding low-density parity-check codes," *Communications, IET*, vol. 1, no. 3, pp. 294–300, Jun. 2007.

[79] S. Papaharalabos, M. Papaleo, P. T. Mathiopoulos, M. Neri, A. Vanelli-Coralli, and G. E. Corazza, "DVB-S2 LDPC Decoding Using Robust Check Node Update Approximations," *Broadcasting, IEEE Transactions on*, vol. 54, no. 1, pp. 120–126, Mar. 2008.

[80] X.-Y. Hu, E. Eleftheriou, D.-M. Arnold, and A. Dholakia, "Efficient Implementations of the Sum-Product Algorithm for Decoding LDPC Codes," in *Proceedings of the IEEE Global Telecommunications Conf. (GLOBECOM'01)*, November 2001, pp. 1036–1036E.

[81] J. Erfanian, S. Pasupathy, and G. Gulak, "Reduced complexity symbol detectors with parallel structure for ISI channels," *Communications, IEEE Transactions on*, vol. 42, no. 234, pp. 1661–1671, 1994.

[82] X.-Y. Hu, E. Eleftheriou, D.-M. Arnold, and A. Dholakia, "Efficient implementations of the sum-product algorithm for decoding LDPC codes," in *Global Telecommunications Conference, 2001. GLOBECOM '01. IEEE*, vol. 2, 2001, pp. 1036 –1036E vol.2.

[83] V. Savin, "Self-corrected Min-Sum decoding of LDPC codes," in *Information Theory. ISIT 2008, IEEE International Symposium on*, july 2008, pp. 146 –150.

[84] M. P. C. Fossorier, M. Mihaljevic, and H. Imai, "Reduced complexity iterative decoding of low-density parity check codes based on belief propagation," *Communications, IEEE Transactions on*, vol. 47, no. 5, pp. 673–680, May 1999.

[85] J. Chen, A. Dholakia, E. Eleftheriou, M. P. C. Fossorier, and X.-Y. Hu, "Reduced-Complexity Decoding of LDPC Codes," *Communications, IEEE Transactions on*, vol. 53, no. 8, pp. 1288–1299, 2005.

[86] M. R. Yazdani, S. Hemati, and A. H. Banihashemi, "Improving belief propagation on graphs with cycles," *Communications Letters, IEEE*, vol. 8, no. 1, pp. 57–59, 2004.

[87] A. Anastasopoulos, "A comparison between the sum-product and the min-sum iterative detection algorithms based on density evolution," in *Global Telecommunications Conference, 2001. GLOBECOM '01. IEEE*, vol. 2, 2001, pp. 1021 –1025 vol.2.

[88] H. Wymeersch, H. Steendam, and M. Moeneclaey, "Log-Domain decoding of LDPC codes over GF(q)," in *IEEE International Conference on Communications*, 2004, pp. 772–776.

[89] J. Hagenauer, E. Offer, and L. Papke, "Iterative decoding of binary block and convolutional codes," *IEEE Transactions on Information Theory*, vol. 42, no. 2, pp. 429–445, Mar. 1996.

[90] G. Byers and F. Takawira, "Fourier transform decoding of non-binary LDPC codes," in *Southern African Telecommunication Networks and Applications Conference (SATNAC) 2004*, no. 2, 2004.

[91] L. Barnault and D. Declercq, "Fast decoding algorithm for LDPC over GF(2q)," in *Information Theory Workshop, 2003. Proceedings. 2003 IEEE*, 2003, pp. 70–73.

[92] H. S. Cruz and J. R., "Reduced-Complexity Decoding of Q-ary LDPC Codes for Magnetic Recording," *IEEE Transactions on Magnetics*, vol. 39, no. 2, pp. 1081–1087, 2003.

[93] K. Kasai and K. Skaniwa, "Fourier Domain Decoding Algorithm of Non-Binary LDPC Codes For Parallel Implementation," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E93-A, no. 1, pp. 1–2, 2010.

[94] K. Kasai and K. Sakaniwa, "Fourier Domain Decoding Algorithm of Non-Binary LDPC Codes for Parallel Implementation," in *IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2011*, Prague, Czech Republic, 2011, pp. 3128–3131.

[95] C. Poulliat, M. Fossorier, and D. Declercq, "Design of Regular (2 , d c) -LDPC Codes over GF (q) Using Their Binary Images," *IEEE Transactions on Communications*, vol. 56, no. 10, pp. 1626–1635, 2008.

[96] D. Declercq and M. Fossorier, "Decoding Algorithms for Nonbinary LDPC Codes Over GF," *Communications, IEEE Transactions on*, vol. 55, no. 4, pp. 633–643, Apr. 2007.

[97] Christian Spagnol, E. M. Popovici, and W. P. Marnane, "Hardware Implementation of GF(2m) LDPC Decoders," *Hardware Implementation of GF(2m) LDPC Decoders*, vol. 56-I, no. 12, pp. 2609–2620, 2009.

[98] J. Chen, S. Member, M. P. C. Fossorier, and S. Member, "Density Evolution for Two Improved BP-Based Decoding Algorithms of LDPC Codes," *IEEE Transactions on Communications*, vol. 6, no. 5, pp. 208–210, 2002.

[99] D. J. Costello and G. D. Forney, "Channel Coding: The Road to Channel Capacity," *Proceedings of the IEEE*, vol. 95, no. 6, pp. 1150–1177, Jun. 2007.

[100] K. Gunnam, G. Choi, W. Wang, E. Kim, and M. Yeary, "Decoding of Quasi-cyclic LDPC Codes Using an On-the-Fly Computation," in *Signals, Systems and Computers, 2006. ACSSC '06. Fortieth Asilomar Conference on*, 2006, pp. 1192–1199.

[101] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near shannon limit error-correcting coding and decoding: Turbo-codes. 1," in *Communications, 1993. ICC '93 Geneva. Technical Program, Conference Record, IEEE International Conference on*, vol. 2, 1993, pp. 1064–1070 vol.2.

[102] E. Sharon, S. Litsyn, and J. Goldberger, "An efficient message-passing schedule for LDPC decoding," in *23rd IEEE Convention of Electrical and Electronics Engineers in Israel*, September 2004, pp. 223 – 226.

[103] G. Falcao, M. Gomes, V. Silva, L. Sousa, and J. Cacheira, "Configurable M-factor VLSI DVB-S2 LDPC Decoder Architecture with Optimized Memory Tiling Design," *EURASIP Journal on Wireless Communications and Networking*, vol. 2012, no. 1, pp. 1–16, 2012.

[104] M.P.C. Fossorier, "Quasicyclic low-density parity-check codes from circulant permutation matrices," *Information Theory, IEEE Transactions on*, vol. 50, no. 8, pp. 1788 – 1793, 2004.

[105] A. Ilic, F. Pratas, and L. Sousa, "Cache-aware roofline model: Upgrading the loft," *IEEE Computer Architecture Letters*, vol. 99, no. RapidPosts, p. 1, 2013.

[106] K. K. Abburi, "A Scalable LDPC Decoder on GPU," in *VLSI Design (VLSI Design), 2011 24th International Conference on*, 2011, pp. 183–188.

[107] K. Abburi, "Cell processor based ldpc encoder/decoder for wimax applications," in *Proceedings of the International Conference on Soft Computing for Problem Solving (SocProS 2011) December 20-22, 2011*, ser. Advances in Intelligent and Soft Computing, K. Deep, A. Nagar, M. Pant, and J. C. Bansal, Eds. Springer India, 2012, vol. 131, pp. 781–790.

[108] M. Beermann, E. Monzó, L. Schmalen, and P. Varyx, "High Speed Decoding of Non-Binary Irregular LDPC Codes Using GPUs," in *Proc. IEEE SiPS*, 2013.

[109] F. L. Blasco, "Implementation of a Multi-User Detector for satellite return links on a GPU platform," in *2014 7th Advanced Satellite Multimedia Systems Conference and the 13th Signal Processing for Space Communications Workshop (ASMS/SPSC)*. IEEE, Sep. 2014, pp. 66–72.

[110] C. H. Chan and F. C. M. Lau, "Parallel decoding of LDPC convolutional codes using OpenMP and GPU," *2012 IEEE Symposium on Computers and Communications (ISCC)*, no. c, pp. 225–227, Jul. 2012.

[111] C. H. Chan and F. C. M. Lau, "Simulation of LDPC convolutional decoders with CPU and GPU," in *2012 2nd International Conference on Consumer Electronics, Communications and Networks (CECNet)*. IEEE, Apr. 2012, pp. 2854–2857.

[112] C.-C. Chang, Y.-L. Chang, M.-Y. Huang, and B. Huang, "Accelerating Regular LDPC Code Decoders on GPUs," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 4, no. 3, pp. 653–659, Sep. 2011.

[113] C.-C. Chang, M.-Y. Huang, and Y.-L. Chang, "Design of GPU-based platform for LDPC decoder," in *2011 IEEE International Geoscience and Remote Sensing Symposium*. IEEE, Jul. 2011, pp. 3429–3432.

[114] Y.-L. Chang, C.-C. Chang, M.-Y. Huang, and B. Huang, "High-throughput GPU-based LDPC decoding," in *Proc. SPIE 7810, Satellite Data Compression, Communications, and Processing VI*, B. Huang, A. J. Plaza, J. Serra-Sagristà, C. Lee, Y. Li, and S.-E. Qian, Eds. International Society for Optics and Photonics, Aug. 2010, pp. 781 008–781 008–8.

[115] H.-P. Cheng, Y.-C. Shen, J.-L. Wu, and K. Aizawa, "High efficient distributed video coding with parallelized design for cloud computing," in *Proceedings of the 19th ACM International Conference on Multimedia - MM '11*. New York, New York, USA: ACM Press, Nov. 2011, p. 1257.

[116] A. D. Copeland, N. B. Chang, and S. Leung, "GPU Accelerated Decoding of High Performance Error Correcting Codes," in *13th Workshop in High Performance Embedded Computing*, 2009.

[117] A. Diavastos, P. Petrides, G. Falcao, and P. Trancoso, "LDPC Decoding on the Intel SCC," *2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pp. 57–65, Feb. 2012.

[118] G. Falcao, S. Yamagiwa, V. Silva, and L. Sousa, "Stream-Based LDPC Decoding on GPUs," in *First Workshop on General Purpose Processing on Graphics Processing Units (GPGPU)*, 2007, pp. 1–7.

[119] G Falcao, V. Silva, M. Gomes, and L. Sousa, "Edge Stream Oriented LDPC Decoding," in *16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*. IEEE, Feb. 2008, pp. 237–244.

[120] G. Falcao, S. Yamagiwa, V. Silva, and L. Sousa, "Parallel LDPC Decoding on GPUs Using a Stream-Based Computing Approach," *Journal of Computer Science and Technology*, vol. 24, no. 5, pp. 913–924, Sep. 2009.

[121] G Falcao, L. Sousa, and V. Silva, "Embedded multicore architectures for LDPC decoding," in *2010 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*. IEEE, Jul. 2010, pp. 349–356.

[122] G. Falcao, J. Andrade, V. Silva, and L. Sousa, "Real-time DVB-S2 LDPC decoding on many-core GPU accelerators," in *Proceedings of the International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, May 2011, pp. 1685–1688.

[123] G. Falcão, L. Sousa, and V. Silva, "Massive parallel LDPC decoding on GPU," in *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, February 2008, pp. 83–90.

[124] G. Falcao, L. Sousa, and V. Silva, "Massively LDPC Decoding on Multicore Architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 2, pp. 309–322, 2011.

[125] G. Falcao, M. Owaida, D. Novo, M. Purnaprajna, N. Bellas, C.D. Antonopoulos, G. Karakonstantis, A. Burg, and P. Ienne, "Shortening Design Time through Multiplatform Simulations with a Portable OpenCL Golden-model: The LDPC Decoder Case," in *IEEE 20th International Symposium on Field-Programmable Custom Computing Machines (FCCM'12)*, May 2012, pp. 224–231.

[126] G. Falcao, V. Silva, L. Sousa, and J. Marinho, "High coded data rate and multicodeword WiMAX LDPC decoding on the Cell/BE," *Electronics Letters*, vol. 44, no. 24, pp. 1415–1417, November 2008.

[127] S. Grönroos and J. Bjorkqvist, "Performance evaluation of LDPC decoding on a general purpose mobile CPU," in *2013 IEEE Global Conference on Signal and Information Processing*. IEEE, Dec. 2013, pp. 1278–1281.

[128] S. Grönroos, K. Nybom, and J. Björkqvist, "Efficient GPU and CPU-based LDPC decoders for long codewords," *Analog Integrated Circuits and Signal Processing*, Jun. 2012.

[129] S. Grönroos, K. Nybom, and J. Björkqvist, "Complexity analysis of software defined dvb-t2 physical layer," *Analog Integrated Circuits and Signal Processing*, vol. 69, no. 2-3, pp. 131–142, 2011.

[130] H. Ji, J. Cho, and W. Sung, "Massively parallel implementation of cyclic ldpc codes on a general purpose graphics processing unit," in *Signal Processing Systems, 2009. SiPS 2009. IEEE Workshop on*, 2009, pp. 285–290.

[131] H. Ji, J. Cho, and W. Sung, "Memory Access Optimized Implementation of Cyclic and Quasi-Cyclic LDPC Codes on a GPGPU," *Journal of Signal Processing Systems*, vol. 64, no. 1, pp. 149–159, 2011.

[132] B. Jiang, J. Bao, and X. Xu, "Efficient simulation of QC LDPC decoding on GPU platform by CUDA," in *2012 International Conference on Wireless Communications and Signal Processing (WCSP)*. IEEE, Oct. 2012, pp. 1–5.

[133] Jing Cui, Y. Wang, and H. Yu, "Systematic Construction and Verification Methodology for LDPC Codes," in *Wireless Algorithms, Systems, and Applications Lecture Notes*, ser. Lecture Notes in Computer Science, Y. Cheng, D. Y. Eun, Z. Qin, M. Song, and K. Xing, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, vol. 6843.

[134] S. Kang and J. Moon, "Parallel ldpc decoder implementation on gpu based on unbalanced memory coalescing," in *Communications (ICC), 2012 IEEE International Conference on*, 2012, pp. 3692–3697.

[135] J. A. Kennedy and D. L. Noneaker, "Decoding of a quasi-cyclic LDPC code on a stream processor," in *2010 - MILCOM 2010 MILITARY COMMUNICATIONS CONFERENCE*. IEEE, Oct. 2010, pp. 2062–2067.

[136] J. Kennedy and D. Noneaker, "Scheduling parity checks for increased throughput in early-termination, layered decoding of qc-ldpc codes on a stream processor," *EURASIP Journal on Wireless Communications and Networking*, vol. 2012, no. 1, pp. 1–10, 2012.

[137] F. Lau and L. Shi, "Programming graphics processing units for the decoding of low-density parity-check codes," in *Advanced Communication Technology (ICACT), 2012 14th International Conference on*, 2012, pp. 1002–1005.

[138] F. C. M. Lau, "Implementation of Decoders for LDPC Block Codes and LDPC Convolutional Codes Based on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 3, pp. 663–672, Mar. 2014.

[139] B. Le Gal, C. Jego, and J. Crenne, "A High Throughput Efficient Approach for Decoding LDPC Codes onto GPU Devices," *IEEE Embedded Systems Letters*, vol. 6, no. 2, pp. 29–32, Jun. 2014.

[140] R. Li, J. Zhou, Y. Dou, S. Guo, D. Zou, and S. Wang, "A multi-standard efficient column-layered LDPC decoder for Software Defined Radio on GPUs," in *2013 IEEE 14th Workshop on Signal Processing Advances in Wireless Communications (SPAWC)*. IEEE, Jun. 2013, pp. 724–728.

[141] Y. Lin and W. Niu, "High Throughput LDPC Decoder on GPU," *IEEE Communications Letters*, vol. 18, no. 2, pp. 344–347, Feb. 2014.

[142] F. J. Martínez-Zaldívar, a. M. Vidal-Maciá, A. Gonzalez, and V. Almenar, "Tridimensional block multiword LDPC decoding on GPUs," *The Journal of Supercomputing*, vol. 58, no. 3, pp. 314–322, Mar. 2011.

[143] Y.-S. Pai, H.-P. Cheng, Y.-C. Shen, and J.-L. Wu, "Fast decoding for ldpc based distributed video coding," in *Proceedings of the international conference on Multimedia*, ser. MM '10. New York, NY, USA: ACM, 2010, pp. 1211–1214.

[144] Y.-S. Pai, Y.-C. Shen, and J.-L. Wu, "High efficient distributed video coding with parallelized design for LDPCA decoding on CUDA based GPGPU," *Journal of Visual Communication and Image Representation*, vol. 23, no. 1, pp. 63 – 74, 2012.

[145] J.-y. Park, "LDPC decoding for CMMB utilizing OpenMP and CUDA parallelization," in *2011 17th Asia-Pacific Conference on Communications (APCC)*, no. October, 2011, pp. 910–914.

[146] J.-Y. Park and K.-S. Chung, "Parallel ldpc decoding using cuda and openmp," *EURASIP Journal on Wireless Communications and Networking*, vol. 2011, no. 1, pp. 1–8, 2011.

[147] D. Romero and N. Chang, "Sequential Decoding of Non-binary LDPC Codes on Graphics Processing Units," in *IEEE ASILOMAR 2012*, Nov 2012, pp. 1267–1271.

[148] E. Scheiber, G. H. Bruck, and P. Jung, "Implementation of an LDPC decoder for IEEE 802 . 11n using Vivado TM High-Level Synthesis," in *International Conference on Electronics, Signal Processing and Communication Systems*, no. 4, 2013.

[149] T.-C. Su, Y.-C. Shen, and J.-L. Wu, "Real-time decoding for LDPC based distributed video coding," in *Proceedings of the 19th ACM international conference on Multimedia - MM '11*. New York, New York, USA: ACM Press, Nov. 2011, p. 1261.

[150] H. P. Thi, S. Ajaz, and H. Lee, "Efficient Min-Max Nonbinary LDPC Decoding on GPU," in *SoC Design Conference, 2014. ISOCC '14. International*, 2014, pp. 266—-267.

[151] H. Tiwari, H. N. Bao, and Y. B. Cho, "A parallel irrwbf ldpc decoder based on stream-based processor," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 23, no. 12, pp. 2198–2204, 2012.

[152] Tsou-Han Chiu, Hsien-Kai Kuo, and B. Lai, "A highly parallel design for irregular LDPC decoding on GPGPUs," in *Signal Information Processing Association Annual Summit and Conference (APSIPA ASC), 2012 Asia-Pacific*, 2012, pp. 1–5.

[153] G. Wang, M. Wu, Y. Sun, and J. R. Cavallaro, "GPU Accelerated Scalable Parallel Decoding of LDPC Codes," in *ASILOMAR '11: Proceedings of the Asilomar Conference on Signals, Systems, and Computers*, 2011.

[154] G. Wang, H. Shen, B. Yin, M. Wu, Y. Sun, and J. Cavallaro, "Parallel Nonbinary LDPC Decoding on GPU," in *IEEE ASILOMAR 2012*, Nov 2012, pp. 1277–1281.

[155] G. Wang, M. Wu, Y. Sun, and J. R. Cavallaro, "A massively parallel implementation of QC-LDPC decoder on GPU," in *Proc. IEEE Symp. on Application Specific Processors*, ser. SASP '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 82–85.

[156] G. Wang, M. Wu, B. Yin, and J. R. Cavallaro, "High throughput low latency ldpc decoding on gpu for sdr systems," in *Global Conference on Signal and Information Processing (GlobalSIP), 2013 IEEE*, Dec 2013, pp. 1258–1261.

[157] S. Wang, S. Cheng, and Q. Wu, "A parallel decoding algorithm of ldpc codes using cuda," in *Signals, Systems and Computers, 2008 42nd Asilomar Conference on*, 2008, pp. 171–175.

[158] S. Wang, L. Cui, S. Cheng, and R. C. Huck, "GPU Acceleration for Particle Filter based LDPC Decoding," in *nVidia Research Summit GPU Technology Conference (GTC)*, San Jose, CA, 2009.

[159] X. Wen, J. Xianjun, P. Jaaskelainen, H. Kultala, C. Canfeng, H. Berg, and B. Zhisong, "A high throughput LDPC decoder using a mid-range GPU," in *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, May 2014, pp. 7515–7519.

[160] S. F. Yau, T. L. Wong, and F. C. M. Lau, "Extremely fast simulator for decoding LDPC codes," in *Advanced Communication Technology (ICACT), 2011 13th International Conference on*, 2011, pp. 635–639.

[161] Yixiang Wang, Hui Yu, and Youyun Xu, "Quasi-cyclic low-density parity-check convolutional code," in *2011 IEEE 7th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*. IEEE, Oct. 2011, pp. 351–356.

[162] L. Yuan, Z. Xing, Y. Zhang, and X. Chen, "An Optimizing Strategy Research of LDPC Decoding Based on GPGPU," in *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, Jul. 2013, pp. 1901–1906.

[163] Y. Zhao, X. Chen, C.-W. Sham, W. M. Tam, and F. C. M. Lau, "Efficient Decoding of QC-LDPC Codes Using GPUs," in *Algorithms and Architectures for Parallel Processing*, ser. Lecture Notes in Computer Science, Y. Xiang, A. Cuzzocrea, M. Hobbs, and W. Zhou, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, vol. 7016, pp. 294—-305.

[164] A. Mink and A. Nakassis, "LDPC error correction for Gbit/s QKD," in *SPIE Sensing Technology + Applications*, E. Donkor, A. R. Pirich, H. E. Brandt, M. R. Frey, S. J. Lomonaco, and J. M. Myers, Eds. International Society for Optics and Photonics, May 2014, p. 912304.

[165] B. Le Gal and C. Jego, "High-throughput multi-core LDPC decoders based on x86 processor," *Parallel and Distributed Systems, IEEE Transactions on*, vol. PP, no. 99, pp. 1–1, 2015.

[166] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*, 4th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[167] G. Blake, R. G. Dreslinski, and T. Mudge, "A Survey of Multicore Processors," *Signal Processing Magazine*, vol. 26, pp. 26–37, 2009.

[168] S. Furber, *ARM System-on-Chip Architecture*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[169] G. Falcão, V. Silva, L. Sousa, and J. Marinho, "High coded data rate and multicodeword WiMAX LDPC decoding on Cell/BE," *Electronics Letters*, vol. 44, no. 24, pp. 1415–1416, 2008.

[170] D. B. Kirk and W. H. Wen-mei, *Programming massively parallel processors: a hands-on approach*. NVIDIA, Morgan Kaufman, 2012.

[171] Khronos Group, *OpenCL 2.0 Specification*. Khronos Group, 2014.

[172] "TOP500 The List." [Online]. Available: http://www.top500.org

[173] W. S. Carter, K. Duong, R. H. Freeman, H.-C. Hsieh, J. Y. Ja, J. E. Mahoney, L. T. Ngo, and S. L. Sze, "A User Programmable Reconfigurable Logic Array," in *Proceedings of the IEEE Custom Integrated Circuits Conference*. IEEE, May 1986, pp. 233–235, first peer-review, public description of a commercial FPGA.

[174] A. DeHon, "The Density Advantage of Configurable Computing," *Computer*, vol. 33, no. 4, pp. 41–49, Apr 2000.

[175] J. Rabaey, "Reconfigurable processing: the solution to low-power programmable DSP," in *Acoustics, Speech, and Signal Processing, 1997. ICASSP-97., 1997 IEEE International Conference on*, vol. 1, Apr 1997, pp. 275–278 vol.1.

[176] J. Varghese, M. Butts, and J. Batcheller, "An efficient logic emulation system," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 1, no. 2, pp. 171–174, 1993.

[177] Altera Corp., "Altera SDK for OpenCL Optimization Guide," 2013. [Online]. Available: http://www.altera.com/literature/hb/opencl-sdk/aocl_optimization_guide.pdf

[178] O. Pell and V. Averbukh, "Maximum performance computing with dataflow engines," *Computing in Science and Engineering*, vol. 14, no. 4, pp. 98–103, 2012.

[179] Xilinx, "Vivado Design Suite User Guide; High-Level Synthesis," 2015. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_3/ug902-vivado-high-level-synthesis.pdf

[180] H. Kaeslin, *Digital integrated circuit design: from VLSI architectures to CMOS fabrication*. Cambridge University Press, 2008.

[181] C. Roth, C. Benkeser, C. Studer, G. Karakonstantis, and A. Burg, "Data mapping for unreliable memories," in *Communication, Control, and Computing (Allerton), 2012 50th Annual Allerton Conference on*, Oct 2012, pp. 679–685.

[182] G. Falcao, V. Silva, L. Sousa, and J. Andrade, "Portable LDPC Decoding on Multicores Using OpenCL," *Signal Processing Magazine*, vol. 29, no. 4, pp. 81–109, 2012.

[183] H. Jin, A. Khandekar, and R. McEliece, "Irregular repeat-accumulate codes," in *Proc. 2nd Int. Symp. Turbo codes and related topics*, 2000, pp. 1–8.

[184] "Encyclopedia of Sparse Graph Codes." [Online]. Available: http://www.inference.phy.cam.ac.uk/mackay/codes/data.html

[185] M. Gomes, G. Falcao, V. Silva, V. Ferreira, A. Sengo, and M. Falcao, "Flexible parallel architecture for DVB-S2 LDPC decoders," in *IEEE Global Telecommunications Conference, 2007. GLOBECOM'07*, 2007, pp. 3265–3269.

[186] G. Falcao, M. Owaida, D. Novo, M. Purnaprajna, N. Bellas, C. Antonopoulos, G. Karakonstantis, A. Burg, and P. Ienne, "Shortening Design Time through Multiplatform Simulations with a Portable OpenCL Golden-model: The LDPC Decoder Case," in *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, 2012, pp. 224–231.

[187] S. Yamagiwa and L. Sousa, "Caravela: A Novel Stream-Based Distributed Computing Environment," *Computer*, vol. 40, no. 5, pp. 70–77, May 2007.

[188] NVIDIA, *CUDA C Programming Guide 6.5*. NVIDIA, 2011.

[189] J. Fang, A. Varbanescu, and H. Sips, "A Comprehensive Performance Comparison of CUDA and OpenCL," in *Parallel Processing (ICPP), 2011 International Conference on*, Sept 2011, pp. 216–225.

[190] T. Richardson and F. Technologies, "The renaissance of Gallager's low-density parity-check codes," *Communications Magazine, IEEE*, no. August, pp. 126–131, 2003.

[191] C. Roth, a. Cevrero, C. Studer, Y. Leblebici, and a. Burg, "Area, throughput, and energy-efficiency trade-offs in the VLSI implementation of LDPC decoders," *2011 IEEE International Symposium of Circuits and Systems (ISCAS)*, pp. 1772–1775, May 2011.

[192] K. Kasai, R. Matsumoto, and K. Sakaniwa, "Information reconciliation for QKD with rate-compatible non-binary LDPC codes," in *IEEE Symp. on Inf. Theory*, 2010, pp. 922–927.

[193] Intel, "The Explosion of Petascale in the Race to Exascale," in *International Supercomputing Conference*, Hamburg, Berlin, 2012.

[194] G. Falcao, V. Silva, L. Sousa, and J. Marinho, "High coded data rate and multicodeword WiMAX LDPC decoding on Cell/BE," *Electronics Letters*, vol. 44, no. 24, pp. 1415–1416, 2008.

[195] M. Mansour, "A Turbo-Decoding Message-Passing Algorithm for Sparse Parity-Check Matrix Codes," *Signal Processing, IEEE Transactions on*, vol. 54, no. 11, pp. 4376–4392, Nov 2006.

[196] Y. Cai, S. Jeon, K. Mai, and B. Kumar, "Highly parallel fpga emulation for ldpc error floor characterization in perpendicular magnetic recording channel," *Magnetics, IEEE Transactions on*, vol. 45, no. 10, pp. 3761–3764, 2009.

[197] J. Ding and M. Yang, "eIRA LDPC Codes on FPGA," *Communications Letters, IEEE*, vol. 15, no. 6, pp. 665–667, June 2011.

[198] F. Verdier and D. Declercq, "A low-cost parallel scalable fpga architecture for regular and irregular ldpc decoding," *Communications, IEEE Transactions on*, vol. 54, no. 7, pp. 1215–1223, 2006.

[199] Y. Dai, Z. Yan, and N. Chen, "Optimal Overlapped Message Passing Decoding of Quasi-Cyclic LDPC Codes," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 16, no. 5, pp. 565–578, May 2008.

[200] Xilinx Inc., "The Xilinx SDAccel Development Environment," available from http://www.xilinx.com/publications/prod_mktg/sdnet/sdaccel-backgrounder.pdf.

[201] M. Owaida, N. Bellas, K. Daloukas, and C. D. Antonopoulos, "Synthesis of platform architectures from opencl programs," in *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*. IEEE, 2011, pp. 186–193.

[202] P. Coussy and A. Morawiec, Eds., *High-Level Synthesis: from Algorithm to Digital Circuit*, 1st ed. Springer Netherlands, 2008.

[203] D. J. C. MacKay and R. M. Neal, "Near Shannon limit performance of low density parity check codes," *Electronics Letters*, vol. 33, no. 6, pp. 457–458, 1997.

[204] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The Landscape of Parallel Computing Research: A View from Berkeley," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec 2006.

[205] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin, "Scaling the Bandwidth Wall: Challenges in and Avenues for CMP Scaling," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09. New York, NY, USA: ACM, 2009, pp. 371–382.

[206] C. Lin and L. Snyder, *Principles of Parallel Programming*, 1st ed. USA: Addison-Wesley Publishing Company, 2008.

[207] F. Pratas, "Stream-based computing and fine-grained parallelism: From algorithms to reconfigurable hardware," Ph.D. dissertation, IST - Technical University of Lisbon, December 2012.

[208] J. Owens and D. Luebke, "A Survey of General-Purpose Computation on Graphics Hardware," *Computer graphics . . .*, vol. 26, no. 1, pp. 80–113, 2007.

[209] R. Vuduc and K. Czechowski, "What gpu computing means for high-end systems," *Micro, IEEE*, vol. 31, no. 4, pp. 74–78, July 2011.

[210] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream Computing on Graphics Hardware," *ACM Trans. Graph.*, vol. 23, no. 3, pp. 777–786, Aug. 2004.

[211] A. Munshi, "The OpenCL Specification," *Khronos OpenCL Working Group*, 2009.

[212] The Linux Documentation Project, "Linux Programmer's Manual, PThreads."

[213] "Threading Building Blocks (Intel© TBB) 4.4." [Online]. Available: https://www.threadingbuildingblocks.org

[214] B. Chapman, G. Jost, and R. Van der Pas, *Using OpenMP: portable shared memory parallel programming*. The MIT Press, 2007.

[215] Pacheco, P. S., *Parallel Programming with MPI*. Morgan Kaufman Publishers, Inc., San Francisco, CA., 1997.

[216] "SPIR$^{\text{TM}}$ The first open standard intermediate language for parallel compute and graphics." [Online]. Available: https://www.khronos.org/spir

[217] "SYCL$^{\text{TM}}$ for OpenCL$^{\text{TM}}$." [Online]. Available: https://www.khronos.org/assets/uploads/developers/library/2014-gdc/SYCL-for-OpenCL-GDC-Mar14.pdf

[218] "WebCL$^{\text{TM}}$ Heterogeneous parallel computing in HTML5 web browsers." [Online]. Available: https://www.khronos.org/webcl

[219] "Threading Building Blocks (Intel© TBB) 4.4." [Online]. Available: http://ark.intel.com/products/codename/29902/Ivy-Bridge

[220] "Intel Tick-Tock Model." [Online]. Available: http://www.intel.com/content/www/us/en/silicon-innovations/intel-tick-tock-model-general.html

[221] AMD, "GPU Computing: Past, Present and Future with ATI Stream Technology," 2010. [Online]. Available: http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/GPUComputing-PastPresentandFuturewithATIStreamTechnology.pdf

[222] "The GREEN 500." [Online]. Available: http://www.green500.org/

[223] J. Dongarra and P. Luszczek, "LINPACK Benchmark," in *Encyclopedia of Parallel Computing*, D. Padua, Ed. Springer US, 2011, pp. 1033–1036. [Online]. Available: http://dx.doi.org/10.1007/978-0-387-09766-4_155

[224] InfiniBand Trade Association, "InfiniBand© Roadmap." [Online]. Available: http://www.infinibandta.org/content/pages.php?pg=technology_overview

[225] Cray Inc., “The Gemini Network.” [Online]. Available: http://wiki.ci.uchicago.edu/pub/Beagle/SystemSpecs/Gemini_whitepaper.pdf

[226] Wes Kendall, “A Comprehensive MPI Tutorial Resource.” [Online]. Available: http://mpitutorial.com/

[227] G. Falcao, “Parallel algorithms and architectures for ldpc decoding,” Ph.D. dissertation, Universidade de Coimbra, 2010.

[228] S. Muller, M. Schreger, and M. Kabutz, “A novel LDPC decoder for DVB-S2 IP,” *Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE '09*, pp. 1308–1313, 2009.

[229] Y. Sun and J. R. Cavallaro, “Efficient hardware implementation of a highly-parallel 3GPP LTE/LTE-advance turbo decoder,” *INTEGRATION, the VLSI journal*, vol. 44, no. 4, pp. 305–315, Sept. 2011.

[230] K. Shimizu, T. Ishikawa, T. Ikenaga, S. Goto, and N. Togawa, “Partially-parallel LDPC decoder based on high-efficiency message-passing algorithm,” in *Computer Design, 2005. Proceedings. 2005 International Conference on*, Oct. 2005.

[231] A.J. Blanksby and C.J. Howland, “A 690-mW 1-Gb/s 1024-b, rate-1/2 low-density parity-check code decoder,” *IEEE Journal of Solid-State Circuits*, vol. 37, no. 3, pp. 404 – 412, 2002.

[232] G. Falcao, M. Gomes, V. Silva, L. Sousa, and J. Cacheira, “Configurable M-factor VLSI DVB-S2 LDPC decoder architecture with optimized memory tiling design,” *EURASIP Journal on Wireless Communications and Networking*, no. 98, March 2012.

[233] F. Kienle, T. Brack, and N. Wehn, “A Synthesizable IP Core for DVB-S2 LDPC Code Decoding,” in *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 100–105.

[234] J. Andrade, “Dvb-s2 ldpc codes decoding on gpus,” MSc, University of Coimbra, July 2010.

[235] “CUDA Occupancy Calculator.” [Online]. Available: http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

[236] C.-H. Chung, Y.-L. Ueng, M.-C. Lu, and M.-C. Lin, “Adaptive quantization for low-density-parity-check decoders,” in *IEEE Symp. on Inf. Theory and its Applications*, 2010, pp. 13–18.

[237] David Declercq, “Status of Knowledge on Non-Binary LDPC Decoders, Part II: Reduced Complexity Non-Binary Decoders.” [Online]. Available: http://sites.ieee.org/scv-sscs/files/2010/08/Tutorial_GFqDecoding_Part2.pdf

[238] Charles Van Loan, *Computational Frameworks for the Fast Fourier Transform*. Society for Industrial and Applied Mathematics, Philadelphia, 1992.

[239] V. Volkov and B. Kazian, “Fitting FFT onto the G80 Architectures,” University of California, Berkeley, 2008.

[240] M. Onsjo, K. Kasai, and O. Watanabe, “CUDA Implementation of Iterative Updating: the Radix-2 Algorithm and Discrete Fourier Transforms,” *Research Reports on Mathematical and Computing Sciences*, Feb. 2010.

[241] G. Falcao, J. Andrade, V. Silva, and L. Sousa, “GPU-based DVB-S2 LDPC decoder with high throughput and fast error floor detection,” *Electronics Letters*, vol. 47, no. 9, pp. 542–543, April 2011.

[242] A. B. Carlson and P. Crilly, *Communication Systems*. McGraw Hill Education, 2009.

[243] NVIDIA, *CUDA Toolkit 7.0 CURAND Guide*. NVIDIA, 2015.

[244] M. Matsumoto and T. Nishimura, “Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudo-random Number Generator,” *ACM Trans. Model. Comput. Simul.*, vol. 8, no. 1, pp. 3–30, Jan. 1998.

[245] G. Martin and G. Smith, “High-Level Synthesis: Past, Present, and Future,” *IEEE Des. Test*, vol. 26, no. 4, pp. 18–25, Jul. 2009.

[246] V. Gutnik and A. Chandrakasan, “Embedded power supply for low-power DSP,” *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 5, no. 4, pp. 425–435, Dec 1997.

[247] A. Lukefahr, S. Padmanabha, R. Das, R. Dreslinski Jr, T. F. Wenisch, and S. Mahlke, “Heterogeneous microarchitectures trump voltage scaling for low-power cores,” in *Proceedings of the 23rd international conference on Parallel architectures and compilation*. ACM, 2014, pp. 237–250.

[248] G. Stitt, “Are Field-Programmable Gate Arrays Ready for the Mainstream?” *Micro, IEEE*, vol. 31, no. 6, pp. 58–63, Nov 2011.

[249] R. Nikhil, “Bluespec System Verilog: efficient, correct RTL from high level specifications,” in *Formal Methods and Models for Co-Design, 2004. MEMOCODE '04. Proceedings. Second ACM and IEEE International Conference on*, June 2004, pp. 69–70.

[250] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, “LegUp: high-level synthesis for FPGA-based processor/accelerator systems,” in *Proc. ACM/IEEE FPGA*, 2011, pp. 33–36.

[251] J. Villarreal, A. Park, W. Najjar, and R. Halstead, “Designing Modular Hardware Accelerators in C with ROCCC 2.0,” in *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, May 2010, pp. 127–134.

[252] Cadence, “C-to-Silicon Compiler High-Level Synthesis Automated high-level synthesis for design and verification,” *White Paper*, 2011. [Online]. Available: http://www.cadence.com/rl/Resources/datasheets/C2Silicon_ds.pdf

[253] M. Lin, I. Lebedev, and J. Wawrzynek, “OpenRCL: Low-Power High-Performance Computing with Reconfigurable Devices,” *2010 International Conference on Field Programmable Logic and Applications*, pp. 458–463, Aug. 2010.

[254] M. Owaida, N. Bellas, K. Daloukas, and C. D. Antonopoulos, “Massively Parallel Programming Models Used as Hardware Description Language: The OpenCL Case,” in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, November 2011.

[255] A. Papakonstantinou, K. Gururaj, J. A. Stratton, D. Chen, J. Cong, and W.-m. Hwu, “Fcuda: Enabling efficient compilation of cuda kernels onto fpgas,” in *Proceedings of the 7th IEEE Symposium on Application Specific Processors*, 2009, pp. 35–42.

[256] I. Mavroidis, I. Mavroidis, I. Papaefstathiou, L. Lavagno, M. Lazarescu, E. de la Torre, and F. Schafer, “FASTCUDA: Open Source FPGA Accelerator & Hardware-Software Codesign Toolset for CUDA Kernels,” in *Digital System Design (DSD), 2012 15th Euromicro Conference on*, Sept 2012, pp. 343–348.

[257] A. J. Blanksby and C. J. Howland, “Parity-Check Code Decoder,” *IEEE Journal of Solid-State Circuits*, vol. 37, no. 3, pp. 404–412, 2002.

[258] M. M. Mansour and N. R. Shanbhag, “High-throughput LDPC decoders,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 11, pp. 976–996, Dec. 2003.

[259] Z. Cui and Z. Wang, “A 170 Mbps (8176, 7156) quasi-cyclic LDPC decoder implementation with FPGA,” in *Circuits and Systems, 2006. ISCAS 2006. Proceedings. 2006 IEEE International Symposium on*, 2006, pp. 5095–5098.

[260] X. Zhang and P. H. Siegel, “Will the real error floor please stand up?” *2012 International Conference on Signal Processing and Communications (SPCOM)*, pp. 1–5, Jul. 2012.

[261] M. Technologies, “MaxCompiler,” 2011. [Online]. Available: http://www.maxeler.com/media/documents/MaxelerWhitePaperMaxCompiler.pdf

[262] J. Chen, A. Dholakia, E. Eleftheriou, M. Fossorier, and X.-Y. Hu, “Reduced-complexity decoding of ldpc codes,” *IEEE Transactions on Communications*, vol. 53, no. 8, pp. 1288 – 1299, aug. 2005.

[263] Xilinx, “Vivado Design Suite: AXI Reference Guide,” 2015. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf

[264] Xilinx, "Zynq-7000 All Programmable SoC and 7 Series Devices Memory Interface Solutions v2.3 User Guide," 2015. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/mig_7series/v2_3/ug586_7Series_MIS.pdf

[265] Y. Sun, G. Wang, and J. Cavallaro, "Multi-layer parallel decoding algorithm and vlsi architecture for quasi-cyclic ldpc codes," in *IEEE Int. Symp. on Circuits and Systems (ISCAS)*, 2011, pp. 1776–1779.

[266] E. Scheiber, G. Bruck, and P. Jung, "Implementation of an LDPC decoder for IEEE 802.11n using Vivado High-Level Synthesis," in *Proc. IEEE ICECS*, 2013.

[267] W. Sulek, M. Kucharczyk, and G. Dziwoki, "GF(q) LDPC decoder design for FPGA Implementation," in *Proc. IEEE CCNC*, 2013.

[268] C. Spagnol, W. Marnane, and E. Popovici, "FPGA Implementations of LDPC over GF(2m) Decoders," in *Proc. IEEE SiPS*, Oct 2007, pp. 273–278.

[269] E. Boutillon, L. Conde-Canencia, and A. Ghouwayel, "Design of a GF(64)-LDPC Decoder based on the EMS Algorithm," *IEEE TCS—I*, vol. 60, no. 10, pp. 2644–2656, 2013.

[270] X. Zhang and F. Cai, "Efficient Partial-Parallel Decoder Architecture for Quasi-Cyclic Nonbinary LDPC Codes," *IEEE TCS–I*, vol. 58, no. 2, pp. 402–414, 2011.

[271] T. Lehnigk-Emden and N. Wehn, "Complexity Evaluation of Non-binary Galois Field LDPC Code Decoders," in *Proc. IEEE Symp. on Turbo Codes & Iter. Inf. Processing*, 2010.

[272] Intel, "Microprocessor Quick Reference," 2012. [Online]. Available: http://www.intel.com/pressroom/kits/quickreffam.htm

[273] Khronos Group, *OpenCL 1.2 Specification*. Khronos Group, 2011.

[274] F. Zhang, Xinmiao and Cai, "Reduced-Complexity Check Node Processing For Non-Binary LDPC Decoding," in *IEEE Workshop on Signal Processing Systems*, 2010, pp. 70–75.

[275] P. U. Adrian Voicila, David Declercq, François Verdier, Marc Fossorier, "Low-complexity, Low-memory EMS algorithm for non-binary LDPC codes," in *ICC*, 2007, pp. 671–676.

[276] G. G. Alban Goupil, Maxime Colas and D. Declercq, "FFT-based BP Decoding of General LDPC Codes over Abelian Groups," *IEEE Transactions on Communications*, vol. 55, no. 4, pp. 644–649, 2007.

[277] Adrian Voicila, David Declercq, François Verdier, Marc Fossorier, and Pascal Urard, "Low-Complexity Decoding for Non-Binary LDPC Codes in High Order Fields," *IEEE Transactions on Communications*, vol. 58, no. 5, pp. 1365–1375, 2010.

[278] "How do I interpret the Logic Utilization number reported in the Quartus II Fitter report?" solution ID: rd05172012_146. [Online]. Available: https://www.altera.com/support/support-resources/knowledge-base/solutions/rd05172012_146.html

[279] M. Stephenson, J. Babb, and A. Amarasinghe, "Bitwidth Analysis with Application to Silicon Compilation," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2000.

[280] J. Llosa, A. González, E. Ayguadé, and M. Valero, "Swing Modulo Scheduling: A Lifetime-Sensitive Approach," in *In IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques (PACT'96*, 1996, pp. 80–86.

[281] M. Frigo, Steven, and G. Johnson, "The design and implementation of fftw3," *Proceedings of the IEEE*, pp. 216–231, 2005.

[282] Nvidia, "Nvidia cuda fast fourier transform library (cufft)," 2015. [Online]. Available: https://developer.nvidia.com/cufft

[283] S. He and M. Torkelson, "Designing pipeline FFT processor for OFDM (de)modulation," in *Signals, Systems, and Electronics, 1998. ISSSE 98. 1998 URSI International Symposium on*, 1998, pp. 257–262.

[284] E. H. Wold and A. M. Despain, "Pipeline and parallel-pipeline FFT processors for VLSI implementation," *IEEE Transactions on Computers*, vol. C-33(5), pp. 414–231, 1984.

[285] M. Hasan, T. Arslan, and J. Thompson, "A novel coefficient ordering based low power pipelined radix-4 FFT processor for wireless LAN applications," *Consumer Electronics, IEEE Transactions on*, vol. 49, no. 1, pp. 128–134, 2003.

[286] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark Silicon and the End of Multicore Scaling," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA '11. New York, NY, USA: ACM, 2011, pp. 365–376.

[287] I. J. Chang, D. Mohapatra, and K. Roy, "A Priority-Based 6T/8T Hybrid SRAM Architecture for Aggressive Voltage Scaling in Video Applications," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 21, no. 2, pp. 101–112, Feb 2011.

[288] S. Ganapathy, A. Teman, R. Giterman, A. Burg, and G. Karakonstantis, "Approximate Computing with Unreliable Dynamic Memories," in *NEWCAS Conference (NEWCAS), 2015 13th IEEE International*, June 2015.

[289] Ngassa, C. Kameni and Savin, V. and Declercq, D., "Unconventional behavior of the noisy min-sum decoder over the binary symmetric channel," in *ITA workshop*, 2013.

[290] G. Karakonstantis, C. Roth, C. Benkeser, and A. Burg, "On the exploitation of the inherent error resilience of wireless systems under unreliable silicon," in *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, June 2012, pp. 510–515.

[291] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture Support for Disciplined Approximate Programming," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII. New York, NY, USA: ACM, 2012, pp. 301–312.

[292] M. Ardakani and F. R. Kschischang, "Gear-shift Decoding," *IEEE Trans. Commun.*, vol. 54, no. 7, pp. 1235–1242, 2006.

[293] K. Cushon, S. Hemati, C. Leroux, S. Mannor, and W. J. Gross, "High-Throughput Energy-Efficient LDPC Decoders Using Differential Binary Message Passing," *IEEE Trans. Signal Processing*, vol. 62, no. 3, pp. 619–631, 2014.

[294] C. Roth, C. Studer, G. Karakonstantis, and A. Burgi, "Statistical Data Correction for Unreliable Memories," in *Signals, Systems and Computers, 2014 48th Asilomar Conference on*, Nov 2014, pp. 1890–1894.

[295] K. Cushon, S. Hemati, S. Mannor, and W. J. Gross, "Energy-efficient gear-shift LDPC decoders," in *Proc. IEEE Int. Conf. on Application-specific Systems, Architectures and Processors*. IEEE, 2014, pp. 219–223.

[296] X. Chen, Q. Huang, S. Lin, and V. Akella, "FPGA-based Low-complexity High-throughput Tri-mode Decoder for Quasi-cyclic LDPC Codes," in *Annual Allterton Conf. on Communication, Control and Computing*, 2009, pp. 600–606.

[297] T. Mohsenin, H. Shirani-mehr, and B. M. Baas, "LDPC Decoder with an Adaptive Wordwidth Datapath for Energy and BER Co-Optimization," *Hindawi Journal of VLSI Design*, vol. 2013, no. 1, pp. 1–14, 2013.

[298] C. Kirsch and H. Payer, "Incorrect systems: It's not the problem, It's the solution," in *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, June 2012, pp. 913–917.

[299] S. Bhunia and S. Mukhopadhyay, *Low-power variation-tolerant design in nanometer silicon*. Springer, 2011.

[300] Z. Chishti, A. Alameldeen, C. Wilkerson, W. Wu, and S.-L. Lu, "Improving cache lifetime reliability at ultra-low voltages," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, Dec 2009, pp. 89–99.

[301] C. Wilkerson, H. Gao, A. R. Alameldeen, Z. Chishti, M. Khellah, and S.-L. Lu, "Trading off Cache Capacity for Reliability to Enable Low Voltage Operation," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ser. ISCA '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 203–214.

[302] S.-T. Zhou, S. Katariya, H. Ghasemi, S. Draper, and N. S. Kim, "Minimizing total area of low-voltage SRAM arrays through joint optimization of cell size, redundancy, and ECC," in *Computer Design (ICCD), 2010 IEEE International Conference on*, Oct 2010, pp. 112–117.

[303] F. Frustaci, M. Khayatzadeh, D. Blaauw, D. Sylvester, and M. Alioto, "SRAM for Error-Tolerant Applications With Dynamic Energy-Quality Management in 28 nm CMOS," *Solid-State Circuits, IEEE Journal of*, vol. 50, no. 5, pp. 1310–1323, May 2015.

[304] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Analysis and Characterization of Inherent Application Resilience for Approximate Computing," in *Proceedings of the 50th Annual Design Automation Conference*, ser. DAC '13. New York, NY, USA: ACM, 2013, pp. 113:1–113:9.

[305] M. K. Qureshi, D. H. Kim, S. Khan, P. Nair, and O. Mutlu, "AVATAR: A Variable-Retention-Time (VRT) Aware Refresh for DRAM Systems," in *Dependable Systems and Networks (DSN), 2015 45th Annual IEEE/IFIP International Conference on*, June 2015.

[306] M. Shoushtari, A. BanaiyanMofrad, and N. Dutt, "Exploiting Partially-Forgetful Memories for Approximate Computing," *Embedded Systems Letters, IEEE*, vol. 7, no. 1, pp. 19–22, March 2015.

[307] Y. S. Park, D. Blaauw, D. Sylvester, and Z. Zhang, "Low-Power High-Throughput LDPC Decoder Using Non-Refresh Embedded DRAM," *IEEE Journal of Solid-State Circuits*, vol. 49, no. 3, pp. 783–794, March 2014.

[308] M. May, M. Alles, and N. Wehn, "A Case Study in Reliability-aware Design: a Resilient LDPC Code Decoder," in *Proceedings of the conference on Design, Automation and Test in Europe*. ACM, 2008, pp. 456–461.

[309] Ngassa, C. Kameni and Savin, V. and Declercq, D., "Analysis of Min-Sum based Decoders Implemented on Noisy Hardware," in *IEEE Asilomar*, 2013.

[310] Ngassa, C. Kameni and Savin, V. and Declercq, D., "Min-sum-based decoders running on noisy hardware," in *IEEE GLOBECOM*, 2013.

[311] C. Novak, C. Studer, A. Burg, and G. Matz, "The effect of unreliable LLR storage on the performance of MIMO-BICM," in *Signals, Systems and Computers (ASILOMAR), 2010 Conference Record of the Forty Fourth Asilomar Conference on*, Nov 2010, pp. 736–740.

[312] A. Hussien, M. Khairy, A. Khajeh, K. Amiri, A. Eltawil, and F. Kurdahi, "A combined channel and hardware noise resilient Viterbi decoder," in *Signals, Systems and Computers (ASILOMAR), 2010 Conference Record of the Forty Fourth Asilomar Conference on*, Nov 2010, pp. 395–399.

[313] Y. Emre and C. Chakrabarti, "Memory error compensation techniques for JPEG2000," in *Signal Processing Systems (SIPS), 2010 IEEE Workshop on*, Oct 2010, pp. 36–41.

[314] A. Balatsoukas-Stimming and A. Burg, "Density Evolution for Min-Sum Decoding of LDPC Codes Under Unreliable Message Storage," *Ieee Communications Letters*, vol. 18, no. 5, pp. 849–852, 2014.

[315] A. Teman, G. Karakonstantis, R. Giterman, P. Meinerzhagen, and A. Burg, "Energy versus data integrity trade-offs in embedded high-density logic compatible dynamic memories," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2015*, March 2015, pp. 489–494.

[316] ARM, "big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7; Improving Energy Efficiency in High-Performance Mobile Platforms," 2013. [Online]. Available: https://www.arm.com/files/pdf/big_LITTLE_Technology_the_Futue_of_Mobile.pdf

[317] A. Moors, T. Rompf, P. Haller, and M. Odersky, "Scala-virtualized," in *Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation*, ser. PEPM '12. New York, NY, USA: ACM, 2012, pp. 117–120.

[318] T. Rompf and M. Odersky, "Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs," *Commun. ACM*, vol. 55, no. 6, pp. 121–130, Jun. 2012.

[319] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala: A Comprehensive Step-by-step Guide*, 1st ed. USA: Artima Incorporation, 2008.

[320] Pervasive Parallelism Laboratory, "Delite." [Online]. Available: http://stanford-ppl.github.io/Delite/index.html

[321] N. George, H. Lee, D. Novo, T. Rompf, K. Brown, A. Sujeeth, M. Odersky, K. Olukotun, and P. Ienne, "Hardware system synthesis from Domain-Specific Languages," in *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, Sept 2014, pp. 1–8.

[322] M. Ringenburg, A. Sampson, I. Ackerman, L. Ceze, and D. Grossman, "Monitoring and debugging the quality of results in approximate programs," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XX. ACM, 2015, pp. 399–411.

[323] X.-Y. Hu, E. Eleftheriou, and D. Arnold, "Regular and irregular progressive edge-growth tanner graphs," *Information Theory, IEEE Transactions on*, vol. 51, no. 1, pp. 386–398, Jan 2005.

[324] J. Bao, Y. Zhan, J. Wu, and J. Lu, "Design of efficient low rate QCARA GLDPC codes," in *Wireless Mobile and Computing (CCWMC 2009), IET International Communication Conference on*, Dec 2009, pp. 213–216.

[325] J. Lin, J. Sha, Z. Wang, and L. Li, "Efficient Decoder Design for Nonbinary Quasicyclic LDPC Codes," *Circuits and Systems I: Regular Papers, IEEE Transactions on*, vol. 57, no. 5, pp. 1071–1082, May 2010.

[326] F. J. MacWilliams and N. J. A. Sloane, *The Theory of Error-Correcting Codes*. North-Holland Mathematical Library, 1978.

[327] V. Pless, *Introduction to the Theory of Error-Correcting Codes*. Wiley-Interscience Series in Discrete Mathematics, 1982.

[328] J. J. Sylvester, "Thoughts on inverse orthogonal matrices, simultaneous sign successions, and tesselated pavements in two or more colours, with applications to Newton's rule, ornamental tile-work, and the theory of numbers," *Philosophical Magazine*, vol. 34, pp. 461–475, 1867.

[329] J. Hadamard, "Résolution d'une question relative aux déterminants," *Bulletin des Sciences Mathématiques*, vol. 17, pp. 240–246, 1893.