UNIVERSIDADE D
COIMBRA

António Fernando Crisóstomo Fraga

Parallel Face Detection

Dissertação no âmbito do Mestrado Integrado em Engenharia Eletrotécnica e de computadores orientada pelo Professor Doutor Gabriel Falcão Paiva Fernandes e a pelo Professor Doutor Luís Filipe Barbosa de Almeida Alexandre presentada à Faculdade de Ciências e Tecnologias da Universidade de Coimbra no Departamento de Engenharia Eletrotécnica e de Computadores

Outubro de 2020

PARALLEL FACE DETECTION

António Fernando Crisóstomo Fraga

Dissertação para obtenção do Grau de Mestre em Engenharia Eletrotécnica e de Computadores

Orientador: Doutor Gabriel Falcão Paiva Fernandes
Co-Orientador: Doutor Luís Filipe Barbosa de Almeida Alexandre

Júri

Presidente: Doutor Jorge Nuno de Almeida e Sousa Almada Lobo
Orientador: Doutor Gabriel Falcão Paiva Fernandes
Vogal: Doutor Pedro Alexandre Dias Martins

Outubro de 2020

Imagination is more important than knowledge. Knowledge is limited. Imagination encircles the world.

- Albert Einstein

Agradecimentos

Em primeiro lugar, gostaria de expressar o meu agradecimento ao Professor Doutor Gabriel Falcão Paiva Fernandes e ao Professor Doutor Luís Filipe Barbosa de Almeida Alexandre pela sua mentoria, dedicação e disponibilidade que demonstraram ao longo do desenvolvimento desta dissertação, tendo tido um papel primordial na conclusão da mesma.

Em segundo lugar, gostaria de agradecer ao Instituto de Telecomunicações por disponibilizar todos os meios de trabalho necessários.

Gostaria de dedicar esta tese aos meus pais, agradecendo-lhes a educação que me deram e por nunca terem deixado de acreditar em mim e no meu sucesso. Por isto, e por tudo o que já fizeram por mim, estarei eternamente grato.

Agradeço às minhas irmãs, pelo companheirismo e pela sua presença assídua em tudo, mesmo nos tempos mais difíceis.

Para além disto, não podia deixar de referir os amigos que Coimbra me deu: Louro, Costa, Matos, Bernardo, JC, Filipe, Janela, Veríssimo, Lousada e Francisco pela companhia e amizade durante este percurso.

Quero agradecer também aos meus amigos de infância: Guilherme, Inês, Pedro e Tomás pela sua amizade incondicional e por sentir que, passe o tempo que passar, estarão sempre presentes e disponíveis. Também quero agradecer à Margarida pelo seu apoio durante o desenvolvimento desta tese.

Por fim, um especial agradecimento ao João Azevedo pela sua disponibilidade, tendo sido essencial para alcançar este resultado.

Muito Obrigado a todos

Abstract

Face detection is typically used millions of times per day in many different contexts and the resolution of the images has seen a significant increase. These high-resolution images can be a very defiant challenge in sequentially based architectures since with the rise in the number of pixels the overall performance of this type of implementation decreases drastically.

This thesis describes the implementation of a framework of the Viola-Jones in parallel architectures such as GPUs and low-power GPUs. They emerge as natural candidates for the acceleration that we seek, offering a very high computational power and core numbers that enable the process of such large amounts of data in parallel.

It also shows the parallelization and optimization of the implementation utilizing the advantages offered by these architectures to achieve an overall performance boost and speedup in high-resolution images when comparing to sequential architectures.

An analysis of the results shows the successful implementation and the influence that the GPU resources available (power, CUDA cores, etc.) have on the overall GPU speedup as well as in its performance. This parallel face detector implementation was able to obtain a global speedup as high as 33 times in 8k images in comparison with the sequential version.

Keywords

CUDA, Parallel Programming, Viola-Jones Face Detector Framework, Low-Power Graphic Processing Units, GPGPU

Resumo

O reconhecimento de faces em imagens é atualmente feito em grande escala e as imagens utilizadas tende a ser cada vez mais de resolução mais elevadas. Isto pode ser um desafio complicado em arquiteturas sequenciais, pois, com o aumento do número total de pixels das imagens, o desempenho geral desse tipo de implementações tende a diminuir drasticamente.

A tese apresentada descreve a implementação da *framework* Viola-Jones em arquiteturas paralelas. Desta forma, as arquiteturas paralelas (GPUs e GPUs de baixo consumo), emergem como a solução ideal já que oferecem elevados valores de poder computacional e números de cores que beneficiam o processamento de grandes quantidades de data em paralelo. Utilizando, assim, as vantagens destas arquiteturas para uma paralelização e otimização específica a esta implementação, obtendo, portanto, uma melhoria significativa na *performance* em comparação a arquiteturas sequenciais em imagens de alta resolução.

Por sua vez, também é realizada uma análise dos resultados desta implementação, que acaba por ser bem-sucedida em diversas GPUs, com o objetivo de fazer uma análise conclusiva da influência dos recursos de GPU disponíveis (Power, CUDA cores, etc.) na aceleração geral da GPU. De referir ainda que este detetor de caras baseado em arquiteturas paralelas foi capaz de obter uma aceleração global de até 33 vezes superior em imagens de 8k em comparação com a versão sequencial inicialmente implementada.

Palavras Chaves

CUDA, Programação Paralela, Framework de deteção de caras Viola-Jones, Unidades de processamento gráfico de baixa potência, GPGPU

Contents

1. Introduction 1

1.1. Motivation 2
1.2. Objectives 3
1.3 Dissertation Outline 4

2. Viola-Jones Face Detector 5

2.1. Viola-Jones Framework 6
2.2. Integral Image 8
2.3. Cascade Classifier 9
2.4. Baseline CPU Implementation 10
2.4.1. Detecting Faces 10
2.4.1.1. Nearest Neighbor 10
2.4.1.2. Integral Image 10
2.4.1.3. Cascade Classifier 10
2.4.1.4. Scale Image Invoker 11
2.5. Related Work 11

3. GPU architecture and CUDA programming model 13

3.1. CPU Architecture 14
3.1.1. CPU Memory 15
3.2. GPU Architecture 16
3.2.1. CUDA 17
3.2.2. Memory Hierarchy 19
3.2.3. CUDA optimization techniques 20

4. Parallelization and optimization of the Viola-Jones 23

4.1. Detecting Faces CPU results 24

4.2. GPU implementation structure 25
4.3. Functions parallelization and optimization 28
4.3.1. Nearest Neighbor and Cascade Classifier 28
4.3.2. Integral Image 29
4.3.3. Scale Image Invoker 34
5. Experimental Results 35
5.1. Methodology 36
5.2. Test apparatus 36
5.3. System Setup 38
5.4. Parallel Detecting Faces 39
5.4.1. Nearest Neighbor 39
5.4.2. Integral Image 40
5.4.3. Cascade Classifier 41
5.4.4. Scale Image Invoker 42
5.5. Overall Speedup Contribution 42
5.6. Energy Efficiency Analysis 45
6. Conclusion & Future Work 47
6.1. Conclusion 48
6.2. Future work 48

List of figures:

2.1 – Three rectangle *Haar Feature* [4] ………...……………………………………………. 6

2.2 – Two rectangle *Haar Feature* [4] ……………………..……………………………………. 6

2.3 – Integral Image of point 1 will the sum of all the pixels belonging to the rectangle A ... 7

2.4 – *Integral Image* 4 is the sum of pixels from the rectangles A + B + C + D ……………. 7

2.5 – Sub-window that passes through the image and example of four Haar Features that are applied to this sub-window [3] ……………………………………………………….…..... 8

2.6 – Input Image exemple ………………………………………………………………………. 9

2.7 – Output obtained from the input image 2.6 ……………………………………………….. 9

2.8 – Data flow of Detecting Faces ………………………………..…………………….... 11

3.1 – Example of a dual-core CPU architecture ……………………………………………… 15

3.2 – CPU and GPU Multiprocessors architecture ………………………………………….. 16

3.3 – CUDA Processing Flow ………………………………..……………………………..…. 18

3.4 – GPU Memory Layout, based on [11] …………………………………………………… 19

3.5 – Grid and Block layout, based on [16] ………………………………………………… 20

3.6 – Data transfer speeds in the GPU and CPU ……………………………….....………. 21

4.1 – Data transfer speeds in the GPU and CPU ……………………………………..………. 26

4.2 – Structure of the GPU Viola-Jones Implementation ………………………….………… 26

4.3 – Parallelization of Cascade Classifier ………………………………………………..… 28

4.4 – Transpose kernel[9] ……………………………………………....…………………..… 29

4.5 – Scan kernel[9] ………………………………………………………………………….... 29

4.6 – Illustration of the work done for images with width or height bigger than two times the maximum number of threads that the GPU used can launch ………………………………. 30

4.7 – Integral Image GPU implementation scheme ………………………………………….. 31

4.8 – CUDA ScanSQ kernel ……………………..……………………………………………… 32

4.9 – New CUDA Scan kernel ……………………………………………………………….. 32

4.10 – New CUDA Transpose kernel …………………………………………………….. 32

4.11 – Parallelization of Scale Image Invoker ...……………………………………………… 33

5.1 – Nearest Neighbor speedup in relation to the total number of pixels ………………… 39

5.2 – Integral Image speedup in relation to the total number of pixels …………………….. 39

5.3 – Cascade Classifier speedup in relation to the total number of pixels ………………. 40

5.4 – Scale Image Invoker speedup in relation to the total number of pixels ..…………… 41

5.5 – Time spent in memory allocation and transfers in relation to the total number of pixels 42

5.6 – Execution times in relation to the total number of pixels 43

5.7 – Detecting Faces GPU speedup in relation to the total number of pixels 43

List of tables:

2.1 – Previous Related Work in Parallelization of Viola-Jones 13

3.1 – Types of Function available in CUDA API 18

5.1 – The three systems used to test the implementation 36

5.2 – Specifications of the three GPUs 37

5.3 – Summary of time, speedup and energy efficiency 44

5.4 Previous Related Work in Parallelization of Viola-Jones in comparison with results obtained 45

List of Acronyms

GPU Graphics Processing Unit

CPU Central Processing Unit

GPGPU General-Purpose computing on Graphics Processing Units

CUDA Compute Unified Device Architecture

ALU Arithmetic Logic Unit

SM Streaming Multiprocessor

HPC High-Performance Computing

OpenCL Open Computing Language

DRAM Dynamic Random-Access Memory

OS Operating System

1 Introduction

Contents

1.1 Motivation
1.2 Objectives
1.3 Dissertation Outline

Face detection is typically used millions of times per day in a context that ranges from mobile phones to personal computers. The use of fast algorithms for this type of applications implies the process of large amounts of data. Typically, CPUs are limited in terms of cores and memory bandwidth being mainly used in sequential operations. The main objective of this work was the development of a fine tuned, fast and parallel face detector. GPUs emerge as natural candidates for acceleration as they offer a very high computational power and core numbers that enables the process of such large amounts of data in parallel. We have developed and optimized a version of the Viola and Jones framework obtained from “Rapid Object Detection using a Boosted Cascade of Simple Features” [2], and originally implemented in CPU. The approach followed exploits distinct GPU contexts that range from high-end till low-power core constrained GPUs and the results achieved. By using the advantages offered by these types of parallel architectures, we were able to achieve performance boosts and significantly decrease execution times compared to the CPU counterpart.

1.1 Motivation

The introduction of computers and their evolution led to the appearance of digital image processing. In this area, several tasks arise according to the needs that have been surfacing over the years. A specific case was the need to detect human faces on smartphones and mobile devices in the early 2010s. These mobile devices, which have limitations in terms of energy and computation made it necessary for the application to take these factors into account and to maintain low energy consumption rates while obtaining relatively low computation times, in order to achieve real-time detection.

In 2001, an article by Paul Viola and Michael Jones, titled "Rapid Object Detection using a Boosted Cascade of Simple Features", was published at the Computer Vision and Pattern Recognition Conference that unveiled an innovative perspective, because it introduced new approaches, for instance, the use of techniques like Cascade Classifier and Integral Image. This approach focused on improving performance with a high success rate.

Since then, several applications have appeared based on these innovative approaches. Also, the continuous evolution and appearance of articles focused on the general improvement of the Framework, as well as general reviews and comparisons of the algorithms used for the detection of faces have appeared over

the years [6], [8]. Other approaches, such as the use of convolution neural networks have also emerged with good results in terms of success rates, but due to the high computational need, and the energy restrictions of mobile devices, it impeded their use, being that currently, Viola-Jones manages to give a good answer to these factors.

Thus, it is intended to implement an application that uses parallel architectures, maintains a high success rate and decreases the overall execution time of the application.

1.2 Objectives

Firstly, we proposed the implementation of a framework that has the main purpose of detecting human faces in an image, based on the Viola-Jones Object Detection framework [3]. This implementation will be done using the sequential architecture of the CPU. Afterwards, tests will be performed, collecting important data for the future comparison between sequential and parallel implementations. Information such as the total execution time, each main function execution time, success rate, and the false-positive rate will be gathered.

This implementation will focus on the creation and development of an application with relatively lower execution times when compared to the first ones obtained in the tests performed, as well as a more efficient computational performance both in terms of energy and in the rate of success by making use of the specialized optimization and parallelization of parallel architectures.

In summary, this work proposes the exploration of parallel architectures for the implementation of a face detector, making a comparative analysis between different GPUs and CPU implementations while maintaining high success rates and low execution times.

1.3 Dissertation Outline

The organization chosen for this thesis is done with six chapters:

Chapter one - introduction into the topic addressed, the motivation behind it, and the main objectives that are intended to be achieved with this work;

Chapter two – in depth look into the Viola-Jones Object Detection Framework, its structure, and the most important functions for its implementation;

Chapter three - presentation of both CPU and GPU architectures and a comparison between them as well as an introduction to GPGPU (General Purpose Graphics Processing Unit), in specific the CUDA programming model;

Chapter four - an explanation behind the optimization and parallelization made in the GPU implementation of the Viola-Jones Object Detection;

Chapter five - presentation and analysis of the results obtained;

Chapter six - conclusions taken from the work done, as well as the possible future development of the work already done

2 Viola-Jones Face Detector

Contents

2.1 Viola-Jones Object Detection Framework
2.2 Integral Image
2.3 Cascade Classifier
2.4 Baseline CPU implementation
2.5 Related Work

This section briefly describes the Viola-Jones object detection algorithm, including discussion regarding its current limitations and the innovative approaches developed under the context of the proposed work.

2.1 Viola-Jones Object Detection Framework

Viola-Jones object detection framework was one of the first to appear in the field of object detection, proposed in 2001 by Paul Viola and Michael Jones in the article Rapid Object Detection using a Boosted Cascade of Simple Features [2]. This article was driven by the high demand for a solution to the problem of detecting human faces in computer images, and with innovative approaches such as Integral Image and Cascade Classifiers, it achieved a significant increase in performance and a very considerable decrease in execution time, maintaining a high success rate compared to the solutions available at the time. Thus, it was possible to obtain results with a good percentage of success in real-time.

2.2 Integral Image

The classification method used is based on simple feature values. These are called Haar Features, and in this case, they are based on the assumption that all human faces have common characteristics with each other. One of the simplest to identify, in a monochrome image, is that the eye area is darker than the upper part of the cheeks. Another is that the nose area is lighter than the area of the two eyes, and, using these and other feature is how the Haar features are created (Fig 2.1 and Fig 2.2 Illustrates Haar features resulting from these two examples).

This way, comparing these tonality differences present in the image it is possible to perceive whether, in fact, in the area of the image in question, there is possibly a human face. But if a small number of filters are used to detect these Haar characteristics, there will be several false positives. To avoid this problem, and to obtain a high success rate, multiple filters are used. Only after successfully passing through multiple filters is a human face detected.

Figure 2.1 *– Three rectangles Haar Feature [4]*

Figure 2.2 – *Two rectangle Haar Feature [4]*

Furthermore, as several filters have to be used, this comparison must be made as quickly as possible, to be able to obtain reasonable execution times and processing several frames per second. For this reason, it was proposed by Viola and Jones, the use of Integral Images, an algorithm that quickly and efficiently adds the values of the pixels within a given rectangular region. Thus, a point (x, y) represents the sum of the values of all the pixels above, and to the left of this point, that is, taking Fig. 2.3 as an example, the Integral Image of point 1 will be the sum of all pixels in rectangle A.

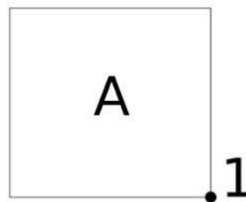

Figure 2.3 *– Integral Image of point 1 will the sum of all the pixels belonging to the rectangle A*

Therefore, and following this logic, the value of Integral Image of point 2 will be the sum of rectangles A and B, and the value in point 3 will be rectangle A plus C, and, finally, the value in point 4 will be the sum of rectangles A, B, C, and D.

This is in fact very important since the difference in tonality is calculated by comparing sums of pixels of rectangular areas, as this approach is more effective than making the comparison pixel by pixel.

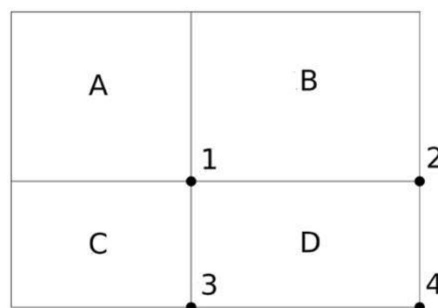

Figure 2.4 *– Integral Image 4 is the sum of pixels from the rectangles A + B + C + D*

Since the features of Haar are composed of two, three, or four rectangles (Fig. 2.1, Fig. 2.2), it is possible to obtain computation times for these filters much shorter than those previously achieved.

This improvement in computation time is extremely considerable in the final execution time of the framework, due to the huge number of times that these filters are

applied since these are not all applied at once in the image, but applied whenever the sub-window is moved. This sub-window passes through the entire image scanning for potential human faces (Fig. 2.5).

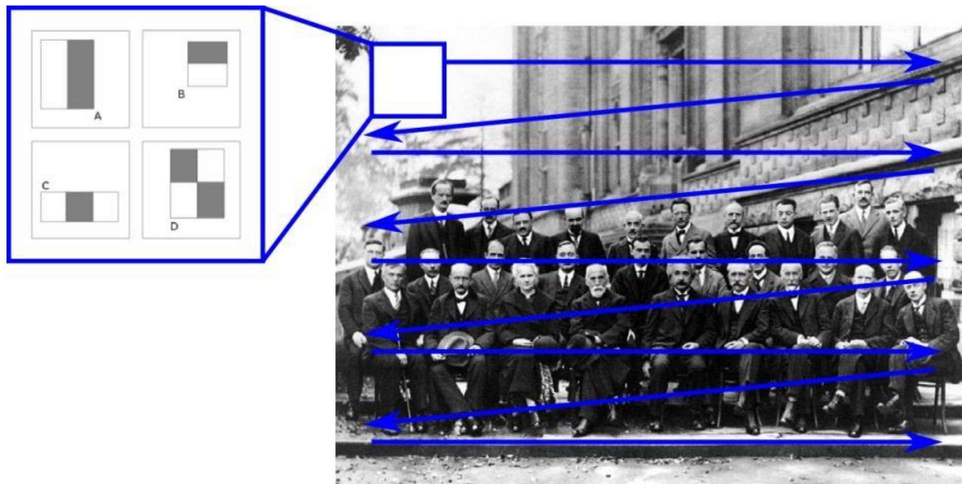

Figure 2.5 - Sub-window that passes through the image and example of four Haar Features that are applied to this sub-window [3]

2.3 Cascade Classifier

The Cascade Classifier algorithm was another of the innovations presented by this Framework, which had an enormous influence on the fact that the results obtained were much better in comparison to the results of other options available at the time in terms of detecting human faces, since this one, in addition to being able to increase the detection performance of the Framework, it also helps to decrease significantly the total execution time.

As mentioned in section 2.2, the image is traversed by a sub-window that is shifted until it completely crosses the image, and over it, a set of filters is applied to detect the Haar Features (Fig. 2.5), but instead of all these filters being applied all at once in this sub-window, creating high computational demand, Viola and Jones suggested that there would be different levels of classifiers, with increasing complexity, as well as an increasing number of filters applied per step.

Therefore, the first level of the classifier is not very complex, having a reduced number of filters, but it still manages to reject a large portion of the image's sub-windows that should be rejected and detects almost all positive sub-windows. As such, there is no need to apply very complex filters to sub-windows that clearly do not have a human face in their area since even with the simplest filters they do not pass this initial stage, making this approach much more efficient and faster.

In the case of the result being positive in the first classifier, the evaluation of the same sub-window is automatically triggered by the second classifier, which in turn, if positive, triggers the evaluation by the third classifier and so on until the most complex classifier, which will confirm the presence of a human face in the sub-window in question.

At any time a negative result in any stage leads to the immediate exclusion of the sub-window.

Finally, as the sub-window has to go through the evaluation of all these stages, with increasing complexity, it makes this process of evaluating and detecting a human face extremely selective. Thus, the percentage of false positives found is very low.

2.4 Baseline CPU Implementation

To acquire deeper knowledge about this Framework a simplified version of it was implemented in CPU[3]. With this sequential implementation, it is possible to understand the areas where parallelization and optimization are possible and will have better results. Therefore, a more informed decision will be possible when choosing the functions that should use GPGPU, CUDA in specific. As the work that has to be performed is now known, and the GPU will only be used in the tasks that its architecture is more efficient than the CPU.

In the CPU implementation, it is requested as input an image in PGM format (grayscale image format), which will be processed to detect any human faces present in the image. This way, the output of this implementation will be a copy of the original image with the faces appearing with a square drawn in their area, as it is possible to observe in the example Fig. 2.6 and Fig. 2.7.

Figure 2.6 *– Input Image example*

Figure 2.7 *– Output result of Figure 2.6*

It is important to note that in this implementation, as it is simplified, a pre-trained cascade classifier is used, with a total of twenty-five stages, containing 9 to 211 filters in

each stage. Even though it is simplified, the success rate presented during the tests was satisfactory, with the failed cases having as main cause the same problem presented by the original article by Viola and Jones: difficulty in detecting side way faces, due to the non-detection of the Haar Features by the applied filters, since the entire face is necessary to not be rejected by the cascade classifier.

For a better understanding of the future work that will be performed for the parallelization of this baseline CPU version using the GPU, a deeper look at each of the main functions that composes this implementation will be taken in the following sub-sections.

2.4.1 Detecting Faces

The Detecting Faces function performs all the work that has to be done for the face detection, and it has to be executed multiple times since the work computed is done for different scales of the image pyramid. This pyramid starts from the original scale and downscales a number of times depending on the size of the original image. The number of downscales performed will be the number of iterations of the Detecting Faces function.

They are four main functions being called by Detecting Faces they are Nearest Neighbor, Integral Image, Cascade classifier and Scale Image Invoker. A quick look at the work that each computes is now presented.

2.4.1.1 Nearest Neighbor

This is the first main function to be called by the detecting faces and has the purpose of downsampling the input image for each level of the image pyramid. This downsampled image will then be used by the other main functions in the computation of their work.

2.4.1.2 Integral Image

The image downsampled by Nearest Neighbor is then passed to the integral image function, that calculates the integral image and the squared integral image of this image. This is done for every iteration of the Detecting faces.

2.4.1.3 Cascade Classifier

The Cascade Classifier is the function that sets the image for the haar filters of all twenty-five stages of the pre-trained cascade classifier. It loads all the necessary

filters with the index of the four corners of the filter rectangle so to be easily compared with the integral image values that have already been calculated.

2.4.1.4 Scale Image Invoker

The actual computation of the filters is only done in this function, here a window is shifted through the image with the current scale being processed with the cascaded filter. For each shift the window has to go through the cascade filter again making this function the most computation heavy out of the four. The work performed by this function is divided in three functions the Scale Image Invoker that calls Run Cascade Classifier, that then calls Eval Weak Classifier.

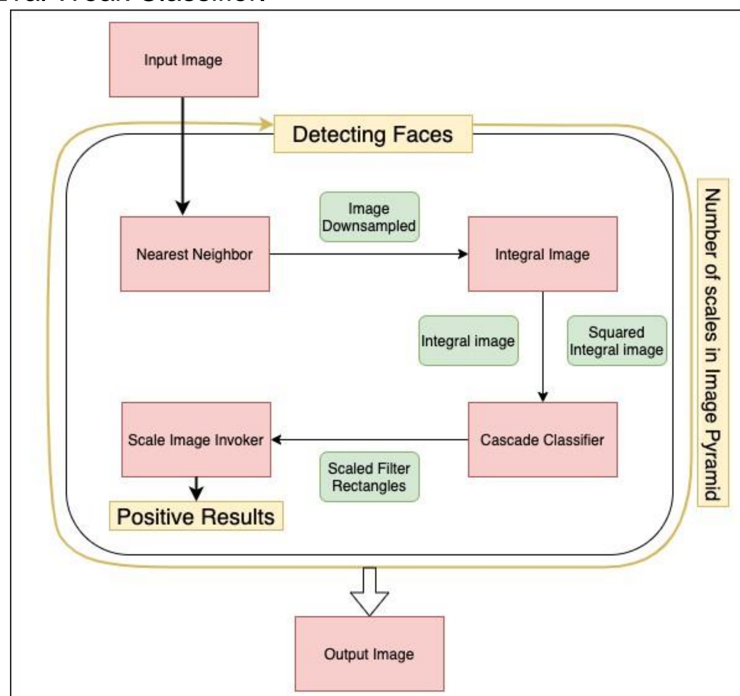

Figure 2.8 – *Data flow of Detecting Faces*

In figure 2.8, it is possible to see the data flow associated with all main functions of Detecting Faces and how the output produced by one function is then used by the following function being called. Only after all scales of the image pyramid have been computed that the positive face detection results are grouped and the output is then written in the output image of the implementation.

2.5 Related Work

The Viola-Jones face detection framework to this day is still one of the more widely used face detection algorithms in research studies, mainly because of its simplicity and the positive results it still produces. For this reason many of the attempts

of implementing face detectors in parallel architectures are based in this famous algorithm. Some related work in this area was explored to provide a reference point for the work that would be performed.

Other attempts of improving the accuracy of the Viola-Jones were present in the literature before, using different methods and technics. We highlight one where the authors propose the initial usage of a skin color filter as well as object extraction to filter out sections of the image that don't seem to have a face present [22]. This combination overcame the problem presented by the skin filter that suffered from a large number of false negatives. This approach achieved its goal and a decrease in both false negative rate and the false positive rate was noticeable.

Although much of the work that has been presented is focused in the accuracy, the main objective of this thesis is the increase in performance and speed. In this area, many approaches have focused in the usage of different hardware. One such approach used Verilog HDL to implement a face detector in a physical FPGA [23]. Having very positive results, achieving very fast detection but with a clear limitation of usability as they only work on the hardware they were developed for, being a disadvantage compared to generic GPU-based optimized implementations.

There are also several papers that utilized parallel architectures, GPUs in specific, and the advantages associated with them to obtain significant speedups in comparison with CPU based implementations. In specific Kong J. and Deng Y. [25], presented the highest average speed up across several resolutions (23x). This method was based in the assignment of one detection window to one GPU thread combined with the implementation of the skin color filtering method that was mentioned above.

With this information gathered, the main goal of this thesis remains the same, focusing on the relative speed up of the implementation in comparison with the baseline CPU version. Using the most recent CUDA versions and techniques, we are able to utilize the increased resources that are now available in the latest GPUs to achieve better results in terms of execution time. The direct comparison with these related works is a difficult to perform accurately as the configurations of the cascade classifier and other parameters of this parallel face detector are not specified, but they are still a useful guideline to follow to understand the success of our implementation and for that, table 2.1 is constructed. A normalized value of performance is presented by multiplying the FPS values by the resolution of the images used (Performance per Pixel), as well this was done per Core and per Watt for a fair comparison between the related works.

Table 2.1 *- Previous Related Work in Parallelization of Viola-Jones*

Paper	**FPS***	**Perfor. per Pixel***	**Perfor. per Core***	**Perfor. per Watt***
Reference	[Frames Per Sec]	[MPixels/Sec]	[KPixels/Sec/Core]	[KPixels/Sec/W]
FPGA [3]	16,08	4,94	NA	1 266,61
GPU [4]	35,00	7,56	19,68	228,99
GPU [5]	37,91	11,65	4,04	47,53
GPU [6]	10,00	2,62	6,83	79,44
GPU [7]	**51,02**	66,87	**278,64**	283,36
GPU [8]	3,00	1,22	0,79	5,30
GPU [9]	9,35	2,87	5,98	13,12

3 GPU architecture and CUDA programming model

Contents

3.1 CPU Architecture
3.2 GPU Architecture

This chapter analyzes the main properties of the CPU and GPU architectures, exposing their main advantages and disadvantages. Also, it elaborates on the reasoning behind the usage of the GPU and its programming model CUDA to achieve the main goal of an overall performance boost and a very significant time reduction on the Viola-Jones implementation addressed before.

3.1 - CPU Architecture

The central processing unit (CPU), is known as the key component of any computer system since it contains all the circuitry needed to interpret and execute program instructions. Since it has the responsibility for logic, basic arithmetic, controlling, and input/output(I/O) operations.

In historical terms, the first computers were physically wired to perform a specific task, and CPUs are normally defined as a device to execute software. As such, the earliest versions of computers weren’t considered to have a CPU.

The first machine to have a CPU was presented in August of 1949 and it was called the EDVAC [21]. This was the first computer designed to perform a certain number of instructions and operations and the programs that these instructions composed were stored in high-speed computer memory rather than physically wired, as in previous machines.

An equally important evolution in CPU history was the advances in technology starting in the era of discrete transistor mainframes till today's era of using integrated circuits to manufacture CPUs. These advances paved the way for the generalization of designs of multi-purpose processors that are produced in mass quantities instead of custom designs for particular applications that were used in the early stages of the CPU history.

Another important milestone was the introduction of multi-core processors, in the beginning, only one core processor was produced consequently they were only able to execute one instruction at a time. Only after many years passed that the first multi-core processor was developed. In theory, dual-core processors would be close to twice as powerful as the single-core processor but in practice, this does not happen, and only about a 50% increase in performance is observed.

The increase of the number of cores (quad-core, octa-core, etc…) was a success, and manufacturers quickly understood it provided a significant performance boost in comparison with its predecessor. This increase, brought to the CPUs the ability to handle even bigger workloads than before, giving it a clear performance boost, mainly in programs that supported multithreading. In today’s market, CPUs still have a limited

number of cores since its main function hasn't changed and they are still optimized to perform sequential operations using significant amounts of local memory (cache) that as very low latency to achieve great performance.

The exponential growth of performance observed in the CPU history (supported by Moore's law), is now coming to a standstill and concerns have arisen about the limits of integrated circuit transistor technology. So to continue the performance growth that CPU has seen since the 1960s when the law was created, expansion and research in the usage of parallelism and the increase in the number of cores present in the CPU is now the new trend in processor development.

3.1.1 – CPU Memory

To reduce the time and energy cost to access data from the main memory, the CPU has a hardware cache that is a small and very fast memory, located very close to the processor's cores. This cache is usually organized as a hierarchy of different cache levels, in most processors, they are 3 levels, L1, L2, and L3, where in terms of speed L1 is the fastest and L3 the slowest. Here copies of data from the main memory are stored, reducing the need for fetching data in higher latency and lower speed memory such as the DRAM or the Hard Drive hence enhancing the overall performance of the operations executed by the CPU.

The overall design of the CPU memory structure can be observed in the figure below:

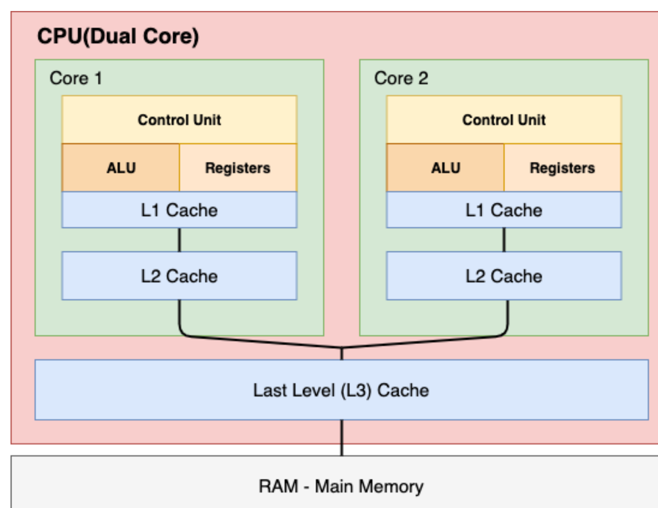

Figure 3.1 *– Example of a dual-core CPU architecture*

3.2 - GPU Architecture

In comparison, the GPU (graphics processing unit) was since the beginning used to accelerate memory-intensive work such as texture mapping and so they were projected to have an architecture completely different from the CPU, as its purpose was completely different as well and was designed with this in mind.

As it's visible in figure 3.2, the CPU has bigger ALUs and Control units as well as bigger cache memory resulting in a bigger and more developed core than the ones in the GPU. Therefore, the CPU has the advantage in more complex tasks that have to be executed sequentially. On the other hand, in tasks that can be divided into simpler operations the numerous smaller cores available in the GPU can be used to parallelize the operations and execute them all at once. This way, taking the most advantage of the hundreds to thousands of small cores that can launch and handle thousands of threads simultaneously. Hence, the GPU architecture is perfectly suited for parallel work and the CPU for sequential work.

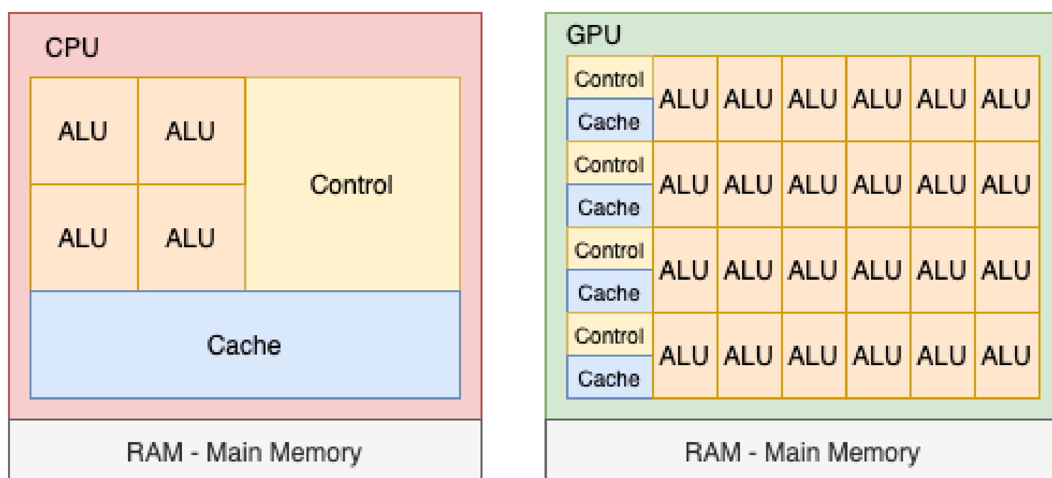

***Figure 3.2** – CPU and GPU Multiprocessors architecture*

In the beginning, these cores had a simple and specific task of rendering but with technology advances registered in this area, GPUs began to process more complex tasks such as realistic 2D and 3D scenarios, and with the popularity of the gaming and multimedia industry rising, this technology arrived in the personal computer.

Today, modern GPUs have become very efficient at manipulating computer graphics and image processing. Besides these tasks, since GPUs offer very high computational power different and new applications began to be presented to harvest and take advantage of this power to process high amounts of data with parallelization (e.g. Deep learning, artificial intelligence, etc…). Considering their highly parallel

structure, GPUs are way more efficient than general CPUs for algorithms that process large amounts of data in parallel. The advances and the explosive growth of research and usage of Deep learning in recent years were in part thanks to the evolution of general GPUs.

To conclude, GPUs with their very large core count and high bandwidth are opening a new range of possibilities for the development of more GPU-accelerated applications. This is where GPGPU (General-Purpose Computing on Graphics Processing Units) development platforms were introduced and play a key role, helping the harvesting of the computational power available by the GPU to perform and accelerate tasks normally only handled by the CPU, instead of only handling the graphics renderings of the computer.

In section 3.2.1, a deeper look into the different GPGPUs available and why the decision to use CUDA, in specific, was made, and how the platform boosts the performance of applications.

3.2.1 – CUDA

Nvidia, one of the biggest producers of GPUs worldwide developed a GPGPU platform named CUDA to help the development of such applications on their GPUs. In 2008, the OpenCL (open source) was another framework that was developed and presented by Apple to the Kronos Group, a technology consortium that now maintains it. Both OpenCL and CUDA are the most used software frameworks for GPGPU (General-Purpose Computing on Graphics Processing Units) helping the acceleration of many applications throw parallel computing using tasks and data-based parallelism.

The biggest difference between them is that CUDA being a proprietary framework from Nvidia, it has top quality support to app developers and has constant development allowing better performance results than OpenCL[10], the only limitation is that CUDA is only supported in NVIDIA GPUs. This better performance attributed to CUDA made it the obvious choice for the optimization and parallelization of the Viola-Jones implementation.

CUDA API at the start only supported the C language, but now it supports Java, C++, and Python and allows the programmer to write kernels, that are functions that are executed on the Device (GPU). An important definition in this framework is that the host is the CPU and the device the GPU.

In the CUDA API, they are three types of functions:

- **Host functions** - run in the host and are only callable by the host, essentially a typical CPU function;

- **Global functions** - executed in the device but can only be called by the host, this type of function requires a configuration at the time of calling. The configuration has as parameters the number of threads per block and the number of blocks per grid, as well as the specification of the dimension of the blocks and grids;
- **Device functions** - executed and called by and from the device, i.e., has to be called inside a global or device function.

As it is visible in the table below:

Table 3.1 – Types of Function available in CUDA API

Type of functions	Executed on	Only callable by
__device__ float DeviceFunction()	device	device
__global__ void GlobalFunction()	device	host
void HostFunction()	host	host

Another important point, is that the main memory is separate from the GPU memory and so it is not accessible by the GPU, nor for writing or reading and so there is a processing flow that has to be followed for the correct functioning of the CUDA API. The correct order and the visual illustration of the CUDA processing flow is presented in figure 3.3.

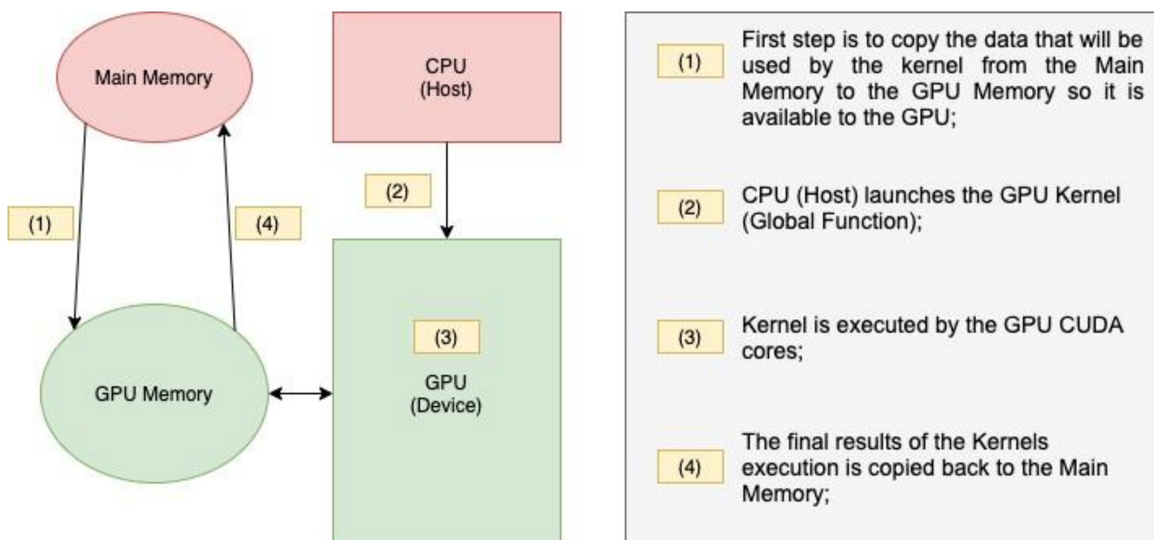

***Figure 3.3** – CUDA Processing Flow*

Normally the tasks executed in parallel on the GPU are simple, but more complex and difficult tasks can be run as well. Considering, that the programming model from the CUDA framework is based on the premise that all problems can be split into simpler sub-problems or tasks that after this division can be parallelized and executed by multiple threads at the same time. Hence, taking advantage of the high number of CUDA cores available to achieve a performance boost in comparison with the sequential execution of the CPU.

3.2.2 GPU Memory Hierarchy

The CUDA threads during runtime, upon the launch of a kernel, have the ability to access data from different memory spaces as illustrated in Figure 3.4. Each thread has its own private local memory, plus the shared memory that is visible to all threads from the same block. Also, all threads have access to global memory. Registers and local memory have the fastest data transfer speeds, then the shared memory and the slowest being the global memory. This decreasing data transfer speed should be taken to account when developing a CUDA application, as memory accessibility is a key part of the optimization of a CUDA implementation. This and other CUDA related optimization techniques will be discussed in greater depth in the next section.

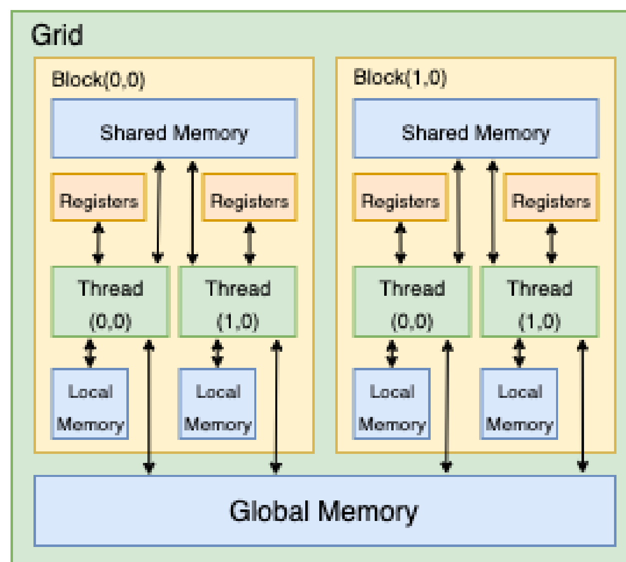

***Figure 3.4** – GPU Memory Layout, based on [11]*

It is also important to understand the hierarchy between grid, blocks, threads and their layout. As visible in Figure 3.5 (below), a grid is an assembly of blocks that is an assembly of threads in itself. The blocks can be created and organized as one-dimensional, two-dimensional (2D), or three-dimensional (3D) grid of thread blocks. This

is not only just for readability, but also for exploiting 2D/3D locality in the blocked shared memory, which provides much faster memory accesses. This decision is made depending on the size of the data being processed and as the number of thread blocks and the number of threads in each block is determined and defined by the size of data to be processed or limited by the number of processors in the system as depending on the GPU there is a maximum number of threads each block can launch. Given that, the same processor core has to have all threads of a block contained in itself and sharing its limited memory resources. In today's average GPU each thread block may contain up to 1024 threads [16].

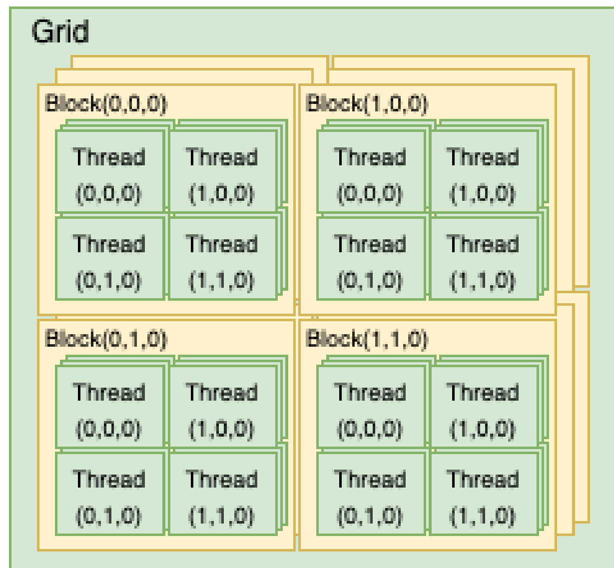

***Figure 3.5** – Grid and Block layout, based on [16]*

3.2.3 CUDA optimization techniques

Before the development of the CUDA implementation, and to utilize to the fullest the advantages offered by the GPU resources, it is important to understand some optimization techniques already developed by NVIDIA[16]. After the study and research, the optimizations that were better suited and were indeed used in this CUDA implementation were memory and instruction optimizations.

Out of the two, memory optimizations have the most significant and visible impact in terms of performance since the transfer times and access times can have a big difference between the memories used. Hence, the optimizations implemented tried to follow two important goals:

- Firstly, the usage of the fastest memories possible maximizing the available bandwidth. One clear example of this, is the usage of shared memory, as it lies on-chip, it has lower latency and higher bandwidth than the global

memory. As it is shared by all threads belonging to the same block (visible in figure 3.4), it is used mainly for inter-block thread communication and reduce global memory transfers. One disadvantage is that when multiple threads try to read or write onto the same memory bank, a bank conflict happens and these accesses are serialized for them to still be able to be executed but the performance suffers. Different strategies such as padding or changing the address patterns can be used to counter this problem.

- Secondly, minimizing data transfers between the host (CPU) and device (GPU) and the other way around. Only doing the transfers that are indeed necessary and mitigating wasted or useless ones. Considering that the bandwidth between the main memory and device memory is the slowest of all available, as it is possible to see in figure 3.6. So all the data that is to be used by the device should be passed by the programmer to the device, in the least number of transfers possible.

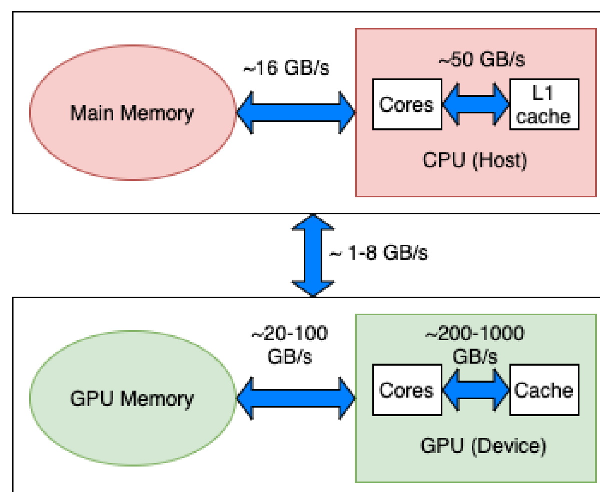

***Figure 3.6** – Data transfer speeds in the GPU and CPU*

At last, the instruction optimizations' main goal is to diminish the number of instructions needed to execute the same amount of work. To achieve this, integrated functions and fast math libraries that are available by Nvidia in CUDAtoolkit are to be used.

4 Parallelization and optimization of the Viola-Jones

Contents

4.1 Detecting Faces CPU results
4.2 GPU implementation structure
4.3 Functions parallelization and optimization

The following chapter analyzes the parallel approach developed, in particular the specific parallelization of each functionality of the original/baseline Viola-Jones algorithm. It aims for an overall performance boost and execution time reduction of every function of the newly proposed GPU implementation in comparison with the baseline CPU implementation.

4.1 Detecting Faces CPU results

Before advancing to the parallelization, the different functions that composed the Detecting Faces had the time that each took in the CPU implementation measured, to see which of them were taking the longest and the percentage of time that was spent on each of them (Chart 4.1). This allowed to concentrate the parallelization work on the functions that were taking the longest as the time reduction in these functions would have the most impact in the overall execution time.

As a result, all main functions that belonged to the function Detecting Faces were timed and the results are shown below.

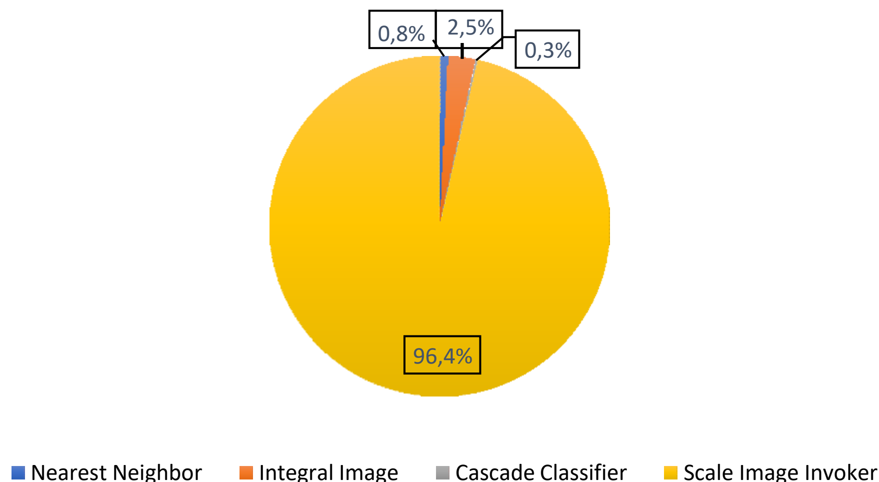

Chart 4.1 – Percentage of each function in the total execution time of Detecting faces

Taking into account the results showed above it is obvious that the first function to take a look at optimizing is Scale Image Invoker as it takes an average of about 96% of the computation time of Detecting Faces. This was foreseeable as it calls Run Cascade Classifier that then calls Eval Weak Classifier, as such, it takes in total the execution time of the three functions combine.

Furthermore, in CUDA implementations it is important to take into consideration the Amdahl's law [22] which states that the theoretical possible speedup obtained by optimizing a function is limited by the fraction of time that the function is taking previously. Using the Amdahl's law formula and considering *n* the number of execution threads equal to 1024 with B being the fraction of the algorithm that is strictly serial equal to 0,05 (as the work performed is highly parallelizable), the theoretical speedup of Scale Image Invoker can be calculated as shown below:

$$S = \frac{1}{B + \frac{1}{n}(1 - B)} = \frac{1}{\frac{1}{20} + \frac{1}{1024}(1 - \frac{1}{20})} = 19{,}64$$

Formula 1 *– Amdahl's law formula and Scale Image Invoker theoretical speedup calculation*

4.2 GPU implementation structure

With this information gathered the next step was to make a decision regarding which functions should be transferred to the device (GPU), whether in terms of possible optimization and parallelization of its work or overall execution time.

Some important considerations that were taken before deciding which functions should be passed to the device (GPU) were:

Understanding that as seen in table 3.1 and explained in section 3.2.1, there are three different function types in CUDA: host, device, and global functions. One of the main points presented and important to remember now is that the global function can´t be called inside the device kernel and the device function can't be called by host functions. As such, when thinking of passing the Scale Image Invoker to the GPU and so changing it to a global function we have obligatorily to change both Run Cascade Classifier and Eval Weak Classifier to device functions so that they can continue to be called by Scale Image Invoker.

Another important aspect is that as device functions are called already inside the kernel, global function when they are called create a device kernel, and as such the launching options such as the number of threads, blocks or the dimension of the grid are already decided and cannot be changed during runtime.

With the bandwidth for memory transfers between device and host being the lowest (8Gb/s), the optimization and parallelization in the device could be not worth it since the time spent in the data transfers between the two can be higher than the time reduction obtained from the device optimization.

So, a deeper analysis of the data used and the amounts of data transfers that would be necessary if only the Scale Image Invoker and the two functions that it calls,

were passed to the device (GPU). And it was visible that for every iteration (total number of filter stages, in this case, is 25) the data would have to be updated in each iteration in both host and device, making it a clear problem in reducing the execution time.

So, when making the decision on which functions should then be passed to the device, the ones that used and updated the same data were the ones selected. Hence, reducing the data transfer Host to Device and Device to Host to the strict minimum necessary.

This way, after the image loading and before any iteration is made, all the data used and updated by the device is transfer and made available in the GPU memory at the start of Detecting Faces.

This way, after the image loading and before any iteration is made, all the data used and updated by the device is transfer and made available in the GPU memory at the start of Detecting Faces.

At the end of Detecting Faces, after all the computation is completed the output results are passed back to the Host, where they are used to draw the rectangles on the faces detected, i.e., writing the output.

With this approach, the time spent transferring data between the two is maintained to the bare minimum necessary as it is extremely important for the achievement of the main goal of reduction in the execution time.

As such, the proposed processing flow of the Detecting Faces CUDA implementation is as shown in the image below.

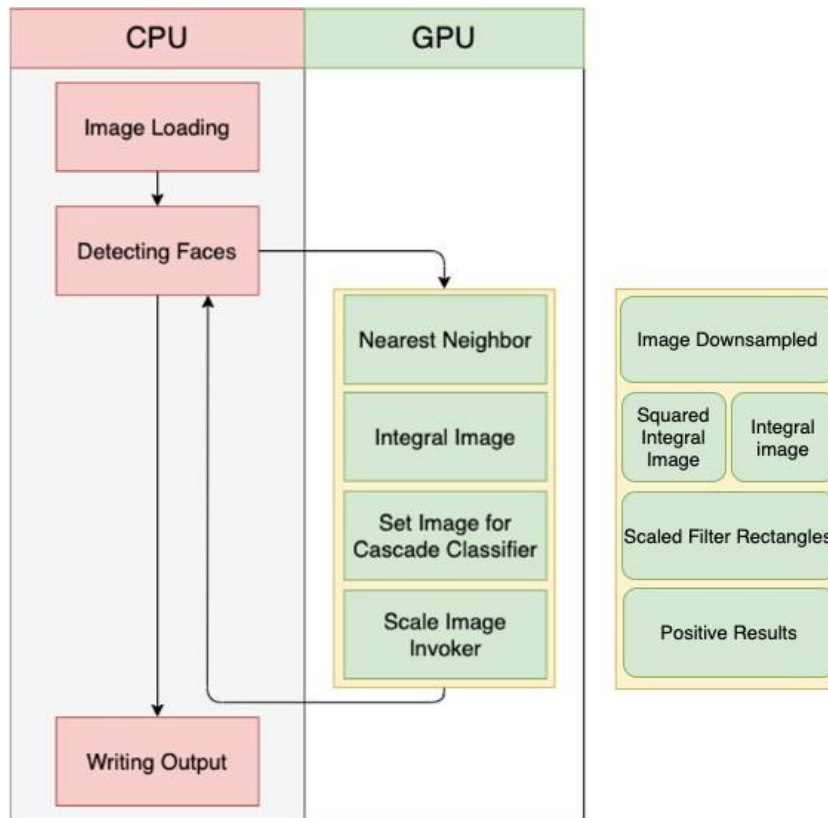

***Figure 4.1** – Proposed processing flow of Detecting Faces CUDA implementation*

A detailed observation of the final scheme of the GPU implementation that was obtained is shown below. Offering a better understanding of the structure of this application, with all functions that composed Detecting Face and their types being displayed.

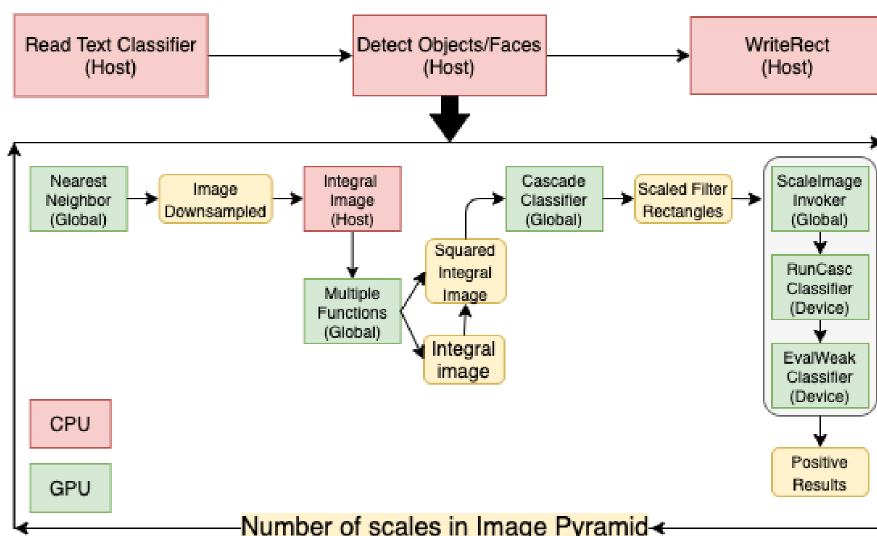

***Figure 4.2** – Structure of the GPU Viola-Jones Implementation*

As shown in figure 4.2, out of the main function of Detecting Faces the Integral Image was the odd one out, considering that its implementation in the GPU had to be decomposed in different device functions as it will be explained in greater depth in section 4.3.1. Thus to optimize the threads usage and obtain better results time-wise, these functions were divided into different kernel calls (global functions). This decision opens up the ability to launch the optimized kernel settings for each function, taking into account the work it does at each iteration and launching only the necessary number of threads to obtain the most optimal implementation possible. As such, the computation work it executes is still all performed in the device.

4.3 Functions parallelization and optimization

In the next section, a more detailed look into each function will now take place, explaining in greater detail the optimization and parallelization made in each one of the newly global and device functions as well as the thinking process behind the decisions taken to obtain a significant speedup in all images, and expecting a bigger increase as the resolutions of the image grow.

4.3.1 Nearest Neighbor and Cascade Classifier

In Nearest Neighbor and Cascade Classifier, the work being computed was simple and for this reason, a division of the work performed was made into threads taking advantage of the parallel computation of the GPU.

To prevent any thread starvation the division made was homogenous. This would decrease the optimization of the implementation since some threads would be doing less work than others and the parallelization would not be optimal as the work is not completed till the last threads have concluded its work.

In the case of the cascade classifier, the parallelization was done as each stage of the cascade classifier is done by a block and each thread of the block in question does the work needed for each filter that belongs to the stage as it is illustrated in the figure below.

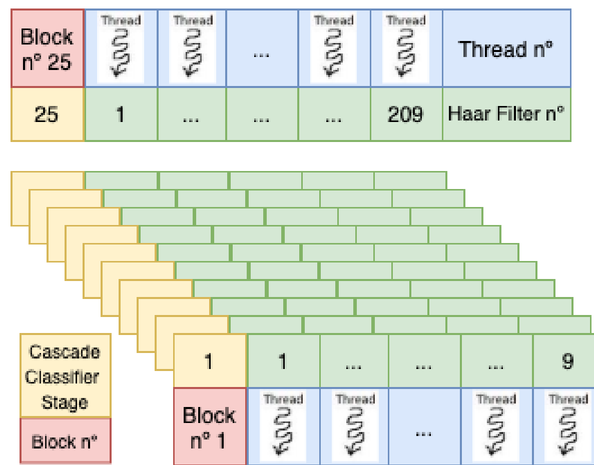

Figure 4.3 – *Parallelization of Cascade Classifier*

4.3.2 Integral Image

Integral Image is a well-known algorithm in computer vision as it is very useful for image processing with box filters since it fairly helps with very high efficiency and quick way to calculate the total sum of data in a specific area.

As such, some CUDA implementations of this algorithm have already been presented as a solution to optimize and reduce the time it takes to calculate these sums of data. Another factor for different papers and studies being presented is that the parallelization isn't the easiest, because different operations needed in this algorithm are sequential. One clear example is the dependence on the previous data since it is a sum of all the previous summed areas plus the area in question. To resolve this problem, scientific papers were presented that overcame it, this, and other problems that were encountered. In particular, the paper "Efficient Integral Image Computation on the GPU" [9] by Berkin Bilgic, Berthold K.P. Horn, and Ichiro Masaki that was presented in 2010 was a clear reference to the algorithm implementation that will be presented.

Taking advance of the parallel characteristics of the GPU, a SCAN function was created to sweep and scan the values of an entire row of the image using each thread to calculate two iterations using temporary shared memory, a type of memory available in the GPU that is visible to all threads within a block. This way, all threads can access the calculations made by others from the same block as them. This approach made it possible to use input images up to 2048 pixels in width, with most of the graphic cards available in 2020 since most can launch up to 1024 threads. In the year of publishing (2010) of the paper referenced fewer threads were available by most GPUs and so only smaller images could be used. But since there is a need to scan the rows as well as the columns, and the scan function only scans the values of the rows, there is a need to transpose the array with the results from the scan function. Another CUDA

function called transpose was presented by the paper in question to perform that task. After scanning the rows, the results from this first scan are transposed by this function and scanned again, and now the columns are scanned as rows, and the total image is now scanned. To finalize, and have the integral image completed the last transpose is needed so that the final result is the same row and columns as the original input. Now, the integral image calculation is finally completed and saved in an output array. Both of the functions are shown in Figures 4.3 and 4.4 (below).

CUDA Code: Transpose kernel for the GPU

```
__global__ void transpose(float *input, float
*output, int width, int height)
{
  __shared__ float temp[BLOCK_DIM][BLOCK_DIM+1];

  int xIndex = blockIdx.x*BLOCK_DIM + threadIdx.x;
  int yIndex = blockIdx.y*BLOCK_DIM + threadIdx.y;

  if((xIndex < width) && (yIndex < height))
  {
    int id_in = yIndex * width + xIndex;
    temp[threadIdx.y][threadIdx.x] = input[id_in];
  }

  __syncthreads();

  xIndex = blockIdx.y * BLOCK_DIM + threadIdx.x;
  yIndex = blockIdx.x * BLOCK_DIM + threadIdx.y;

  if((xIndex < height) && (yIndex < width))
  {
   int id_out = yIndex * height + xIndex;
   output[id_out] = temp[threadIdx.x][threadIdx.y];
  }

}
```

Figure 4.4 *– Transpose kernel [9]*

CUDA Code: Scan kernel for the GPU

```
__global__   void   scan(float   *input,   float
*output, int n)
{
  extern __shared__ float temp[];
  int tdx = threadIdx.x; int offset = 1;

  temp[2*tdx]   = input[2*tdx];
  temp[2*tdx+1] = input[2*tdx+1];

  for(int d = n>>1; d > 0; d >>= 1)
  {
    __syncthreads();
    if(tdx < d)
    {
      int ai = offset*(2*tdx+1)-1;
      int bi = offset*(2*tdx+2)-1;
      temp[bi] += temp[ai];
    }
    offset *= 2;
  }

  if(tdx == 0) temp[n - 1] = 0;

  for(int d = 1; d < n; d *= 2)
  {
    offset >>= 1; __syncthreads();
    if(tdx < d)
    {
      int ai = offset*(2*tdx+1)-1;
      int bi = offset*(2*tdx+2)-1;

      float t  = temp[ai];
      temp[ai] = temp[bi];
      temp[bi] += t;
    }
  }
  __syncthreads();
  output[2*tdx]   = temp[2*tdx];
  output[2*tdx+1] = temp[2*tdx+1];
}
```

Figure 4.5 *– Scan kernel [9]*

Both functions presented have limitations, in the case of the transpose function, there is a need for the input array to have the same number of rows and columns. This is, the input image has to be a square. On the other hand, the scan function as previously stated has to receive as input an image of dimensions not greater than the maximum number of threads that the GPU used can launch multiplied by two, seeing that each thread is responsible for the scan of data concerning two pixels. In most mid-tier modern GPUs, this value is about 1024 threads per block, and as such this concludes to a maximum image size of 2048x2048p. Another limitation of the scan function is that the width and height of the input image have to be the power of 2. This is because the input has to be dividable by two since each thread is responsible for 2 pixels, limiting the possible images that could be used.

To counter these limitations, firstly before any calculation is made with the data, the input array is padded with zeros to the dimension of the closest power of 2, i.e., if an image is 2010x1003p it will be padded to 2048x 1024p. Opening the opportunity to the usage of the function scan in non-power of 2 dimensioned images.

Secondly, to solve the problem presented when the images had the width or height bigger than the maximum number of threads multiplied by two (most cases 2048p), a division of the rows using two or more blocks is made. If the image is bigger than 4096p then 3 blocks are needed, since each block can only launch 1024 threads (2048p processed) and so on. Each block is to be considered as a segment. The shared memory used by the scan function is shared only by the threads that belong to the same block so the calculations made in each segment are independent of other segments. But the scan function should scan and sum the entire row and with the division made only the sum of each segment is made. Knowing the index of each block, we save the values of the last iteration of each block in an auxiliary array to after the scan function is completed, sum these values to all the values in all the segments with a higher block index.

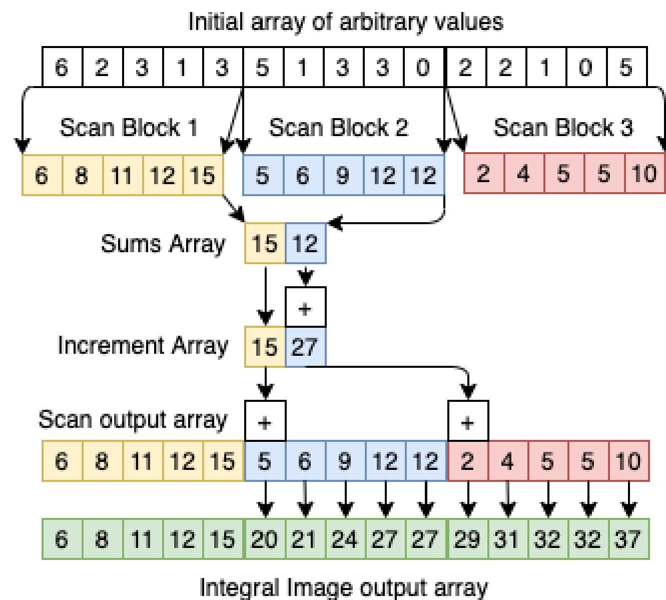

***Figure 4.6** – Illustration of the work done for images with width or height bigger than two times the maximum number of threads that the GPU used can launch*

Using this segment division and padding, the scan function can now scan any image of any size. For images smaller than 2048p, the padding still happens to the closest power of 2 and for optimized kernel call, the number of threads launched is equal to the width and after transpose equal to the height of the image. This approach tries to generalize the usage of the scan and transpose functions proposed by the paper, that was as well limited since its input had to be squared (same width and height) and was

changed to accept any padded input. Thus, this CUDA implementation for the Integral Image algorithm presents an optimized and parallelized approach taking advantage of the shared memory available in the GPU and can expect higher speedup results in bigger images with no limitation in terms of the input image.

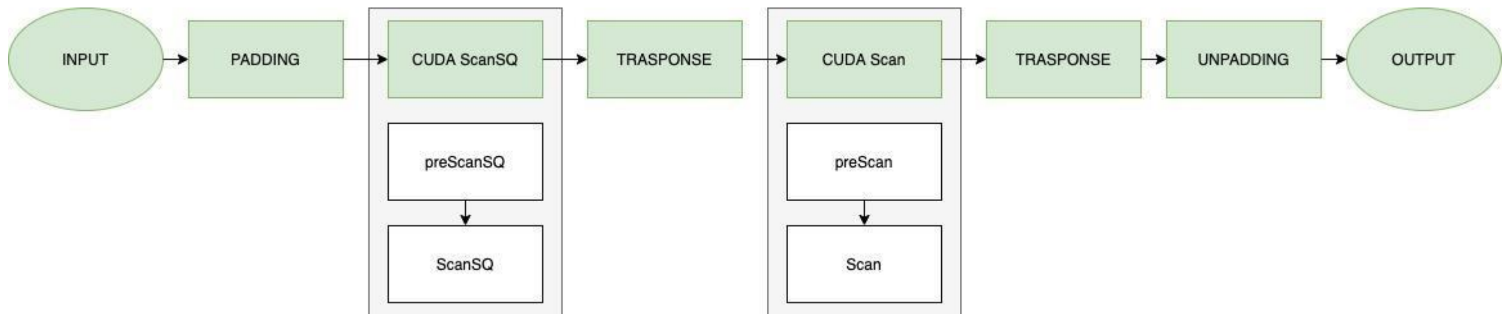

Figure 4.7 *– Integral Image GPU implementation scheme*

The figure 4.6, illustrates the order of operations and global function calls needed to achieve the desired result. As in this implementation, it had to calculate the Sum of the data as well as its Square Sum. To achieve this, the scan function was altered so to do both of the calculations at the same time as they use the same data to do its calculation. Hence, the first CUDA SCAN call is in fact the ScanSQ that does both the sum and the square sum. Right away, both are Transposed and scan again but this time only with Scan that does only the sum. To finalize both are again transposed and the Square Sum and Sum of the entire image are concluded.

Below, both Scan and ScanSQ function that were developed for this Integral Image algorithm, based on the functions presented in the paper “Efficient Integral Image Computation on the GPU” [9] by Berkin Bilgic, Berthold K.P. Horn, and Ichiro Masaki are displayed. As well as, the Transpose function that suffered an alteration, as already explained to work with any image resolution.

CUDA Code: Scan and Square Scan kernel for the GPU

```
__device__ void SQScan(int *input, int *output,int *inputSQ, int *outputSQ,
int n, int *sums, int *sumsSQ , int new_width){

    extern __shared__ int buffer[];
    int *temp = &buffer[0];
    int *tempSQ = &buffer[new_width];
    int tdx = threadIdx.x;
    int offset = 1;

    temp[2 * tdx] = int(input[2 * tdx]);
    temp[2 * tdx + 1] = int(input[2 * tdx + 1]);
    tempSQ[2 * tdx] = int(input[2 * tdx])* int(input[2 * tdx]);
    tempSQ[2 * tdx + 1] = int(input[2 * tdx + 1])* int(input[2 * tdx + 1]);

    for (int d = n >> 1; d > 0; d >>= 1) {
        __syncthreads();
        if (tdx < d) {
            int ai = offset * (2 * tdx + 1) - 1;
            int bi = offset * (2 * tdx + 2) - 1;
            temp[bi] += temp[ai];
            tempSQ[bi] += tempSQ[ai];
        }
        offset *= 2;
    }
    __syncthreads();
    if (tdx == 0) {
        unsigned int block_id = gridDim.y * blockIdx.x + blockIdx.y;
        sums[block_id] = temp[n-1];
        temp[n - 1] = 0;
        sumsSQ[block_id] = tempSQ[n-1];
        tempSQ[n - 1] = 0;
    }
    for (int d = 1; d < n; d *= 2) {
        offset >>= 1;
        __syncthreads();
        if (tdx < d) {
            int ai = offset * (2 * tdx + 1) - 1;
            int bi = offset * (2 * tdx + 2) - 1;

            int t = temp[ai];
            temp[ai] = temp[bi];
            temp[bi] += t;

        int tSQ = tempSQ[ai];
            tempSQ[ai] = tempSQ[bi];
            tempSQ[bi] += tSQ;
        }
    }
    __syncthreads();
    output[2 * tdx] = temp[2 * tdx] + input[2 * tdx];
    output[2 * tdx + 1] = temp[2 * tdx + 1] + input[2 * tdx+1];
    outputSQ[2 * tdx] = tempSQ[2 * tdx] + input[2 * tdx]*input[2 * tdx];
    outputSQ[2 * tdx + 1] = tempSQ[2 * tdx + 1] + input[2 * tdx + 1]*input[2 * tdx + 1];
}
```

***Figure 4.8** – New CUDA ScanSQ kernel*

CUDA Code: New Scan kernel for the GPU

```
__device__ void Scan(int *input, int *output,int *inputSQ, int *outputSQ,
int n, int *sums, int *sumsSQ , int new_width) {

    extern __shared__ int buffer[];
    int *temp = &buffer[0];
    int *tempSQ = &buffer[new_width];
    int tdx = threadIdx.x;
    int offset = 1;

    temp[2 * tdx] = int(input[2 * tdx]);
    temp[2 * tdx + 1] = int(input[2 * tdx + 1]);

    tempSQ[2 * tdx] = int(inputSQ[2 * tdx]);
    tempSQ[2 * tdx + 1] = int(inputSQ[2 * tdx + 1]);

    for (int d = n >> 1; d > 0; d >>= 1) {
        __syncthreads();
        if (tdx < d) {
            int ai = offset * (2 * tdx + 1) - 1;
            int bi = offset * (2 * tdx + 2) - 1;
            temp[bi] += temp[ai];
            tempSQ[bi] += tempSQ[ai];
        }
        offset *= 2;
    }
    __syncthreads();
    if (tdx == 0) {
        unsigned int block_id = gridDim.y * blockIdx.x + blockIdx.y;
        sums[block_id] = temp[n-1];
        temp[n - 1] = 0;
        sumsSQ[block_id] = tempSQ[n-1];
        tempSQ[n - 1] = 0;
    }
    for (int d = 1; d < n; d *= 2) {
        offset >>= 1;
        __syncthreads();
        if (tdx < d) {
            int ai = offset * (2 * tdx + 1) - 1;
            int bi = offset * (2 * tdx + 2) - 1;

            int t = temp[ai];
            temp[ai] = temp[bi];
            temp[bi] += t;

            int tSQ = tempSQ[ai];
            tempSQ[ai] = tempSQ[bi];
            tempSQ[bi] += tSQ;
        }
    }
    __syncthreads();
    output[2 * tdx] = temp[2 * tdx] + input[2 * tdx];
    output[2 * tdx + 1] = temp[2 * tdx + 1] + input[2 * tdx+1];
    outputSQ[2 * tdx] = tempSQ[2 * tdx] + inputSQ[2 * tdx];
    outputSQ[2 * tdx + 1] = tempSQ[2 * tdx + 1] + inputSQ[2 * tdx + 1];
}
```

***Figure 4.9** – New CUDA Scan kernel*

CUDA Code: New Transpose kernel for the GPU

```
__global__ void CoalescedTranspose(const int *input, int *output) {
    __shared__ int tile[TRANSPOSE_TILE_DIM][TRANSPOSE_TILE_DIM + 1];

    int x = blockIdx.x * TRANSPOSE_TILE_DIM + threadIdx.x;
    int y = blockIdx.y * TRANSPOSE_TILE_DIM + threadIdx.y;
    int width = gridDim.x * TRANSPOSE_TILE_DIM;

    for (int j = 0; j < TRANSPOSE_TILE_DIM; j += TRANSPOSE_BLOCK_ROWS)
        tile[threadIdx.y+j][threadIdx.x] = input[(y + j) * width + x];

    __syncthreads();

    x = blockIdx.y * TRANSPOSE_TILE_DIM + threadIdx.x;
    y = blockIdx.x * TRANSPOSE_TILE_DIM + threadIdx.y;
    width = gridDim.y * TRANSPOSE_TILE_DIM;

    for (int j = 0; j < TRANSPOSE_TILE_DIM; j += TRANSPOSE_BLOCK_ROWS)
        output[(y+j)*width + x] = tile[threadIdx.x][threadIdx.y + j];
}
```

***Figure 4.10** – New CUDA Transpose kernel*

4.3.3 Scale Image Invoker

An important factor that had to be taken into account in this implementation was that Scale Image Invoker calls Run Cascade Classifier that then calls Eval Weak Classifier. So the parallelization was made knowing that each thread will have to perform the work from both functions.

With this in mind, the work was divided so that each thread launched was responsible for the execution of the work of both functions. As this work was performed in each pixel of the image a 2D block was used, with the pixel being represented by the x and y from the CUDA block. Each thread will be doing computation work of one image pixel that in this case is the work that would be performed at each sliding window of the sequential version, as each window was the size of one pixel. This way, all threads will be go through the cascade classifier at the same time, possible as there is now data dependency on the cascade filters. This way the total number of pixels from the image is processed all at the same time instead of one pixel at a time as it was with the sliding window in the sequential form that was practiced by the CPU as it can be observed in figure 4.11.

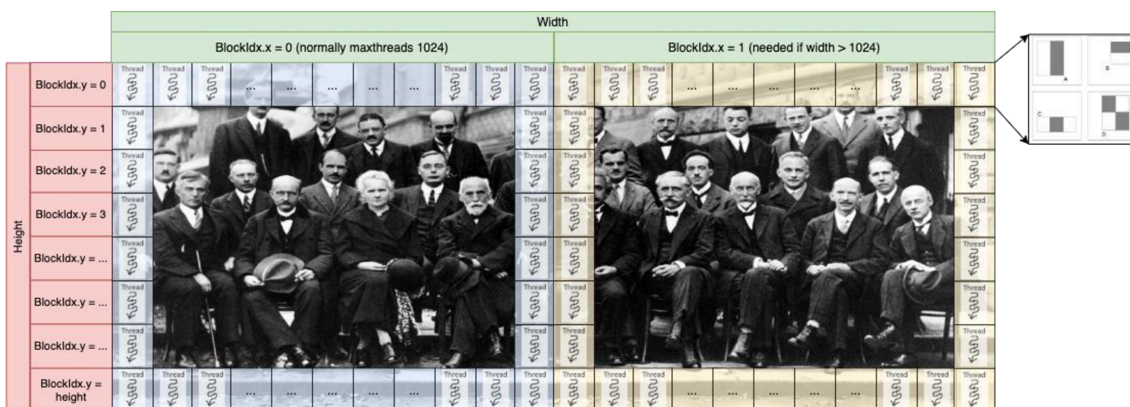

Figure 4.11 *– Parallelization of Scale Image Invoker*

With this implementation it is expected to achieve the transformation of a very timely function with a high sequential workload into a paralyzed function with a high-performance boost in comparison to the CPU version and a big reduction of the total execution time is expected.

To conclude, with this GPU Viola-Jones Implementation optimization and parallelization an overall performance boost across all the functions mentioned is expected to be visible with a big-time reduction and a high value of GPU Speedup. The goal of the next chapter will be to correlate the results obtained with the increase of the total number of pixels in the image processed to understand and conclude if in fact the parallelization and optimization were well performed.

5 Experimental Results

Contents

5.1 Methodology
5.2 Test apparatus
5.3 System Setup
5.4 Parallel Detecting Faces
5.5 Overall Conclusion
5.6 Energy Efficiency Analysis

The main purpose of this chapter is to analyze and present the results obtained from the parallelization and optimization of the Detecting Faces function that executes a great part of this Viola-Jones Implementation, as explained in the previous chapter. The functions that compose Detecting Faces and that were modified and optimized are the Nearest Neighbor, Integral Image, Scale Image Invoker, and Cascade Classifier. Therefore, these four functions were the ones that their execution was timed in both CPU and GPU implementations, as well as, the total execution time of the Detecting Faces function. Thus, a direct comparison of both implementations can be made and the GPU speedup calculated.

5.1 - Methodology

Each test was repeated five times for each test image. A group of test images was assembled, from a very small and low-resolution image (300x160p) to 8k(7680x4320) images. This way, a conclusion could be taken regarding the performance of this GPU implementation as the number of pixels were increasing and get to values that top-of-the-line smartphones and computers use at this moment in time. While the tests were run, the time that each function took, and the total execution time for each test image was recorded. After the five repetitions, for all time measurements, the standard deviation and the average values were calculated, hence obtaining more reliable and consistent results. The accuracy of detection of the faces present in the test image was as well compared with the CPU implementation and, as expected the results were equal. With this data and information, the comparison between both could be made and conclusions be taken from this optimized and parallelized GPU implementation.

5.2 - Test apparatus

In the beginning, these tests were performed using a test bench (Setup A) consisting of an average GPU, the GTX 960. With 1024 CUDA cores it's considered to be a low-performance device, but the versatile and cost-efficient option to the average home computer. After, and to ensure a more complete and consistent result set, it was of interest to test this implementation with a different range GPUs. Therefore, two new test systems were used, one with a high-end GPU that has more resources than a typical GPU and a low power option, that emphasizes power consumption over performance. To achieve this goal, setup B) and C) were used to perform the same tests with the same images and they are presented in the table below:

Table 5.1 - The three systems used to test the implementation developed

Setup	GPU	CPU	RAM
a)	GTX 960	Intel Core i7-4790 @ 3.60 GHz	16 GB
b)	RTX 2080 Ti	Intel Core i7-4790K @ 4.00 GHz	32 GB
c)	Jetson TX2's GPU	ARM - Cortex A57 @ 2GHz	-

Mostly intended for HPC, the RTX 2080 Ti opens up the ability to analyze the performance boost obtained with the availability of more power, more CUDA cores, and higher clocks speeds. Furthermore, as power is a very important factor in any technology and can be a limitation in many system setups, the test was run as well on a Jetson TX2, that has a custom-built GPU, with the purpose of maximizing energy efficiency and consumption on low power applications.

With these tests in different GPUs and different test benches (Table 5.1), it was possible to construct a spectrum of power consumption and analyze the impact that power has on the performance of this GPU implementation, as well as other resources like the number of CUDA cores available. The detailed overview of each GPU can be seen in the following table:

Table 5.2 – Specifications of the three GPUs

	GTX 960	RTX 2080 Ti	Jetson TX2
Architecture	Maxwell 2.0	Turing	Pascal
Memory Size	4 GB	11 GB	8 GB (shared with CPU)
Memory Type	GDDR5	GDDR6	LP-DDR4
Memory Bus	128 bit	352 bit	128 bit
Bandwidth	112.2 GB/s	616.0 GB/s	59.7 GB/s
Base Clock	1127 MHz	1350 MHz	854 MHz
Boost Clock	1178 MHz	1545 MHz	1465 MHz
Memory Clock	1753 MHz	1750 MHz	1866 MHz
TDP	120 W	250 W	7.5W - 15W
SM Count	8	68	2
L1 Cache (per SM)	48 KB	64 KB	48 KB
L2 Cache	1024 KB	5.5 MB	512 KB
FP32(float) performance	2.413 TFLOPS	13.45 TFLOPS	750.1 GFLOPS
FP64 (double) performance	75.39 GFLOPS	420.2 GFLOPS	23.44 GFLOPS
Cuda Cores	1024	4352	256
Transistor	2,940 million	18,600 million	N/A
Process Size	28 nm	12 nm	16 nm

5.3 System Setup

The system setup setup A), B) used for the test and for the results of the sequential version a controlled environment in Ubuntu 20.04.1 LTS. The setup C) was the only one that differs using an older version of Ubuntu the 16.04 version.

The latest version of the implementation presented consists of a parallel solution made in CUDA and developed entirely from the serial version (CPU)[3] and, like so, does not deviate from the premises of the algorithm. Thus, the output of each function was inspected to confirm whether it matched the serial version.

In terms of the compilers used, for the sequential version the GNU Compiler Collection (GCC) was used and for the parallel version, the NVIDIA CUDA Compiler (NVCC) was used.

The time measurements used to calculate the total execution times for each function in order to be as accurate as possible were made using the Chrono library, which is a flexible collection for C++ that tracks time with varying degrees of precision. In this case, depending on the time the function took, the degree of precision was changed between milliseconds or microseconds.

5.4 Parallel Detecting Faces

Times were measured in all three test systems (Table 5.1), and a comparison between the speedup obtained with the RTX2080 Ti (4086 CUDA cores), the GTX 960 (1024 CUDA cores), and the Jetson TX2 (256 CUDA cores) was built. Furthermore, as they vary with respect to in power consumption, CUDA cores, clock speeds and memory (Table 5.2), an analysis of the influence of these resources on the overall performance boost of the implementation presented was also performed.

Secondly, as the main focus of the implementation were the functions Nearest Neighbor, Integral Image, Cascade Classifier, and Scale Image Invoker since they composed Detecting Faces and the work they perform was done by the GPU, taking advantage of its parallel architecture to obtain an overall performance boost. A separate view at each function GPU speedup will be taken as well for a more detailed look into the overall speedup and execution times of this CUDA implementation in comparison to the sequential version.

5.4.1 Nearest Neighbor

The nearest neighbor execution time obtained in the CPU was already one of the lowest in the implementation. Still, and as visible in figure 5.1 with the parallelization and optimization made in the Nearest Neighbor Kernel implementation, a big GPU speedup was observed in all test systems. As well, with the increase in the number of pixels, the speedup growth tends to be exponential.

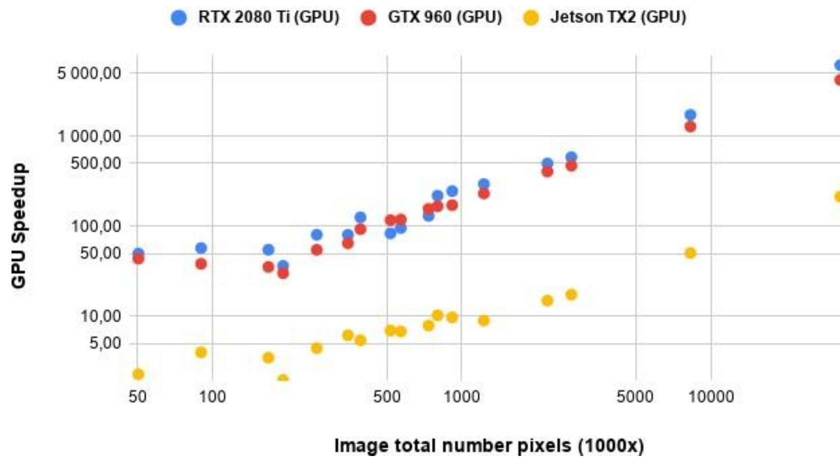

__Figure 5.1__ – Nearest Neighbor speedup in relation to the total number of pixels

5.4.2 Integral Image

In the case of the Integral Image function, that was decomposed in different global functions, had multiple kernels being launched in its execution. This combined with the need to allocate and free memory in the device in each iteration makes the execution time and consequently the GPU speedup lower than expected. Having negative results across the board, only at the biggest image of the test set (8k image) does the speedup in the 2080 Ti achieve a positive result.

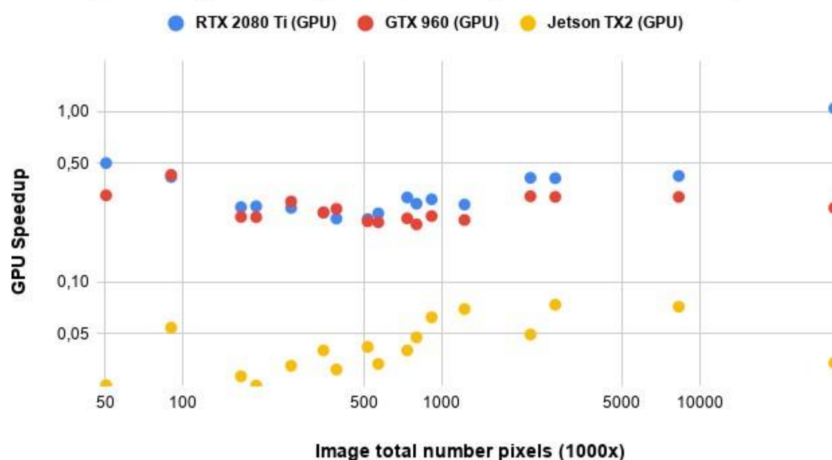

__Figure 5.2__ – Integral Image speedup in relation to the total number of pixels

Even with this negative record, the fact that the data is being altered and saved in the device memory gives the Detecting Faces implementation an advantage since there is no need for data transfers between the host and the device. These transfers would have to happen with every iteration of the Detecting Faces and would consequently diminish the overall speedup of this CUDA implementation.

5.4.3 Cascade Classifier

In comparison with the other two functions that obtained positive speedup (Nearest Neighbor and Scale Image Invoker), this is the only one that the speedup does not scale with the increase of the total number of pixels. This is a consequence of the Cascade Classifier having always the same dimension independent of the resolution of the image. Therefore, the work computed in this function will always be the same and its speedup in comparison to the sequential implementation will tend to be in the 10 to 20 range for both RTX 2080 Ti and the GTX 960. In the case of the Jetson TX2, the speedup varies between 1,5 and 4 (Figure 5.3).

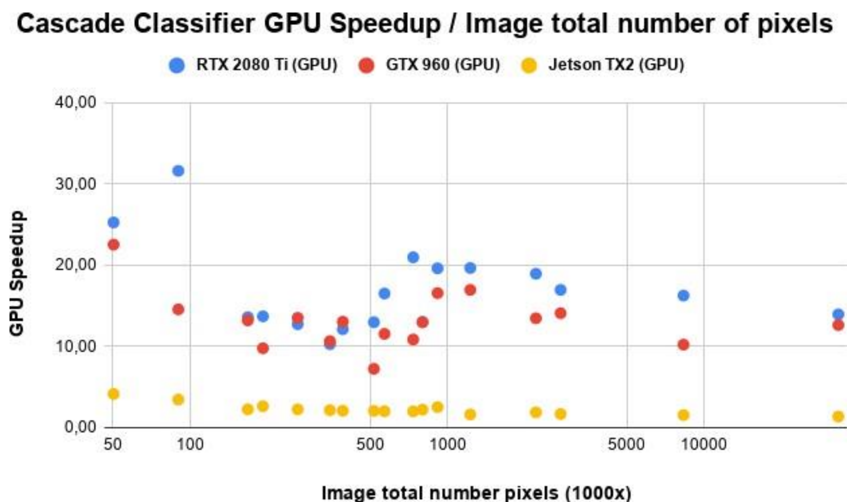

Figure 5.3 *– Cascade Classifier speedup in relation to the total number of pixels*

5.4.4 Scale Image Invoker

Scale Image Invoker being the most time-consuming function of the CPU implementation would always play a key part in the achievement of the main goal presented for this implementation, obtaining an overall speedup and performance boost. For this reason, a huge speedup presented by its GPU implementation was crucial.

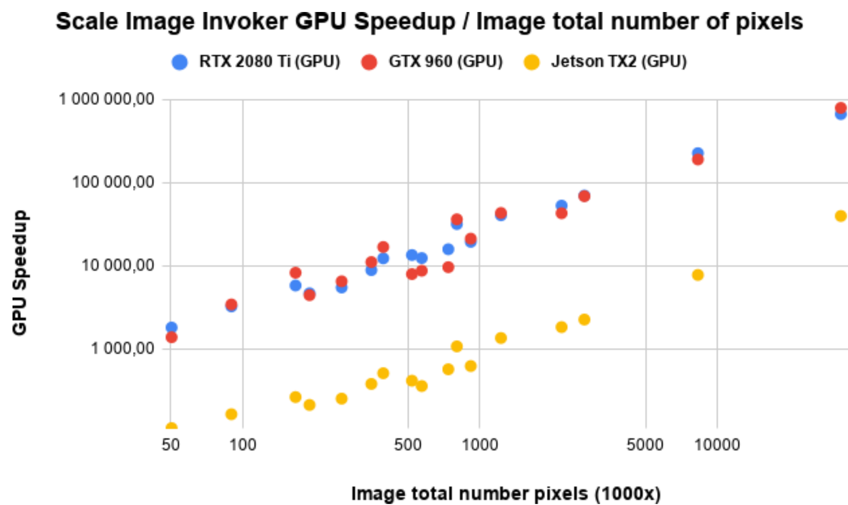

Figure 5.4 *– Scale Image Invoker speedup in relation to the total number of pixels*

As shown, a huge difference in the performance between the CPU and GPU implementation was achieved and the speedup values were the highest ones across all function's implementation. This is explained by the fact that in the sequential version each pixel was processed at a time and as the image resolution grows so did the function execution time. However, in the case of the GPU, this work was parallelized and optimize using its grid to simplify the problem, bringing huge gains in every device speedup (as much as 500 000).

5.5 Overall Speedup Contribution

Finally, an important consideration to make is that the results presented so far, do not take into account the elapsed time to allocate device memory and the data transfers from the host to the device, in preparation for the computation of the four functions that compose Detecting Faces in the GPU. This necessity is a bottleneck in most CUDA implementations and it was an important goal to reduce them to the minimum necessary, as already explained in previous sections. Therefore, the time it took to perform this groundwork was as well measured and it can be examined in Figure 5.5.

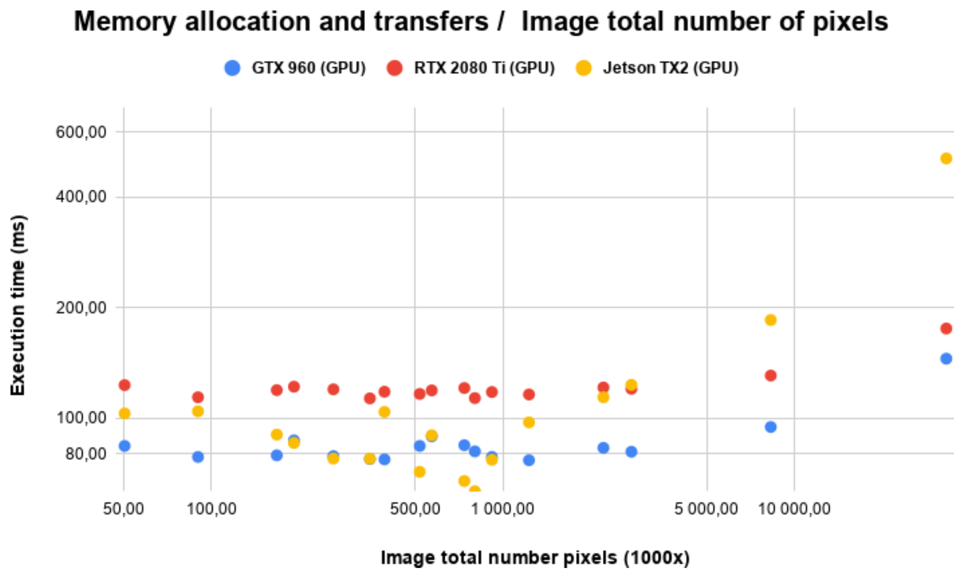

***Figure 5.5** – Time spent in memory allocation and transfers in relation to the total number of pixels*

As seen in figure 5.5, this CUDA implementation independently of the image size will always have a minimum of about 80 milliseconds of execution time. In the RTX 2080Ti this value is closer to 110 milliseconds, as it is always necessary for these memory allocations and transfers to happen with every image. Only at the two biggest images of the set, it is possible to see an increase in the time spent, as the data size begins to rise to larger values. In the case of the Jetson TX2, given its very limited memory resources and bandwidth this increase is even more noticeable.

Consequently, it is foreseeable that the speedup in smaller images that took less than 80 milliseconds in the CPU implementation, to be negative. However, the GPU speedup results already presented in all functions, except the integral image, were very positive. Furthermore, in the functions Scale Image Invoker and Nearest Neighbor, with an increasing number of pixels, an exponential growth of the speedup was noticed. Therefore, an overall speedup and drastic reduction in the total execution time of the Detecting Faces functions was expected and confirmed as it can be witnessed in Figures 5.6 and 5.7.

Figure 5.6 – *Execution times in relation to the total number of pixels*

In the matter of performance across all machines, the best overall time results were recorded by the RTX 2080 Ti, as it was expected, achieving a speedup of up to thirty-three in comparison to the serial version. The GTX 960 follows closely in execution time and consequently in the speedup as well, till the 8k(7680x4320) image where it is visible the limitation of its resources in comparison with the RTX 2080 Ti. Finally, the Jetson TX2 whose results were always expected to be worse than the other two GPUs, but still manage after the 1 million total pixel mark to obtain a better result than both sequential implementation at a much lower energy cost.

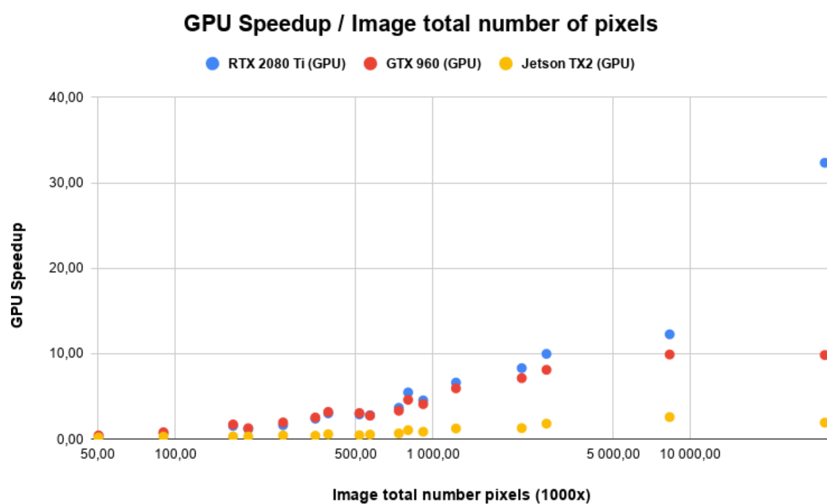

Figure 5.7 – *Detecting Faces GPU speedup in relation to the total number of pixels*

These values show that across every device tested, the implementation achieves a very positive GPU speedup result after passing the memory allocation and transfers bottleneck and as the number of pixels grows, the speedup grows as well. This happens until the resources available in the device start to be insufficient for the workload demand and therefore the speedup stagnates. This isn't reflected in the high-end GPU, the RTX 2080 Ti. As the limiting factors of the GPU end up being its grid size and processing power, where there is a big difference between the RTX 2080 Ti and the GTX 960. Only in a bigger image where finally the size of data being processed is enough to achieve the GTX 960 resource limitation, can the RTX 2080Ti showcase its sovereignty in these parameters over the GTX 960 resulting in a big speedup difference and continuing its exponential growth instead of stagnation as the GTX 960 and Jetson encounter.

All in all, it can be considered that the implementation is in fact well optimized and parallelized as the GPU speedup values were, as already said and shown, very positive and had the tendency to increase exponentially when there wasn't any hardware limitation.

5.6 Energy Efficiency Analysis

In terms of performance, as it was visible in the results, the Jetson TX2 being a low-power GPU cannot compete with the other GPUs used, as it prioritizes energy efficiency over performance. Hence, for it to be a fair comparison an analysis of the energy efficiency (FPS/Watt) was performed for all test systems.

Table 5.3 *- Summary of time, speedup and energy efficiency - *Higher is better*

Image Resolution	RTX 2080 Ti			GTX 960			Jetson TX2		
	FPS	Speedup*	FPS/W*	FPS	Speedup*	FPS/W*	FPS	Speedup*	FPS/W*
2k (2048x1080)	**31,1**	9,9	0,4	25,3	8,12	0,5	5,3	1,8	**0,7**
4k (3840x2160)	15,4	12,2	0,2	12,1	9,92	0,2	2,4	2,5	0,3
8k (7680x4320)	10,2	**32,3**	0,1	3,5	9,85	0,06	0,3	1,9	0,07

The energy efficiency analysis shown in table 5.3, reveals that the Jetson TX2 obtains positive results in terms of frames per second divide by watt. As it has higher values than the GTX 960 in all three images chosen and in comparison, with the best overall performing GPU, the RTX 2080 Ti it only has lower value in the biggest image of the test set, where the high-end GPU is able to obtain excellent results of nearly 32x speedup in comparison with the sequential implementation.

*Table 5.4 - Previous Related Work in Parallelization of Viola-Jones in comparison with results obtained - *Higher is better*

Paper	FPS	Perfor. per Pixel	Perfor. per Core	Perfor. per Watt
Reference	[Frames Per Sec]	[MPixels/Sec]	[KPixels/Sec/Core]	[KPixels/Sec/W]
FPGA [3]	16,08	4,94	NA	1 266,61
GPU [4]	35,00	7,56	19,68	228,99
GPU [5]	37,91	11,65	4,04	47,53
GPU [6]	10,00	2,62	6,83	79,44
GPU [7]	**51,02**	66,87	**278,64**	283,36
GPU [8]	3,00	1,22	0,79	5,30
GPU [9]	9,35	2,87	5,98	13,12
[This work w/ GTX960]	12,15	100,78	98,42	839,81
[This work w/ RTX 2080 Ti]	10,02	**332,44**	76,39	1 329,76
[This work w/ Jetson TX2]	2,45	20,32	79,38	**2 540,16**

Using the same model as Table 2.1 we can compare the results obtained to other related work in this area. As visible in table 5.4, using the same normalized parameters of comparison (Performance per second, per core and per watt) we can conclude the success of our implementation with higher values in contrast with those of the related work presented in terms of energy efficiency (Jetson TX2) and performance per pixel (RTX 2080 Ti).

6 Conclusion & Future Work

Contents

6.1 Conclusion
6.2 Future work

6.1 Conclusion

The final implementation obtained does in fact represent a clear optimization of the sequential Viola-Jones Framework. As the main goal proposed in this thesis is achieved and an overall performance boost is obtained in the implementation of Viola-Jones face detector in parallel architectures. Using CUDA optimization techniques as well as parallelization to obtain high speedup values in both GPUs in comparison to the sequential version. Opening up the possibility of real-time detections, even in an image as big as 8k the detection of faces was done in less than a second.

In the case of the low-power GPU (Jetson TX2), even with its resources and memory limitations, it achieved similar results to CPU implementation, while consuming a very significant lower value in power.

All in all, with the implementation in both GPUs (GTX 960 and RTX 2080 Ti) and the low-power Jetson TX2 GPU, that vary in many aspects between them as expressed before, gave the possibility to elaborate conclusive analyses of the impact that power and other important GPU resources (CUDA cores, memory bandwidth, etc.) have on the overall performance and speedup of the implementation presented and conclusions could as well be taken about the energy efficiency.

6.2 Future Work

One of the emerging platforms from the computer vision scientific area is the usage of convolution neural networks (CNNs) to process images. This has recently seen a great evolution and interest in the scientific community. Thus, an integration of such technologies in this implementation could have been developed as the hardware limitations that once were a great problem are now being lifted with the appearance of ever more powerful GPUs.

Another possible future development in this work could be the introduction of new features and filters for the detection of different objects to complement the face detection already implemented.

Bibliography

[1] Jain, V., & Patel, D. (2016). A GPU based implementation of robust face detection system. Procedia Computer Science, 87, 156-163.

[2] Viola, P., & Jones, M. (2001, December). Rapid object detection using a boosted cascade of simple features. In Proceedings of the 2001 IEEE computer society conference on computer vision and pattern recognition. CVPR 2001 (Vol. 1, pp. I-I). IEEE.

[3] "Viola-Jones Face Detection", 2012. [Online] Available at: https://sites.google.com/site/5kk73gpu2012/assignment/viola-jones-face-detection, [Last Access: 27/10/2019];

[4] "Digital image processing". [Online] Available at: https://en.wikipedia.org/wiki/Digital_image_processing, [Last Access: 12/1/2020];

[5] Dang, K., & Sharma, S. (2017, January). Review and comparison of face detection algorithms. In 2017 7th International Conference on Cloud Computing, Data Science & Engineering-Confluence (pp. 629-633). IEEE.

[6] Zhou, Y., Liu, D., & Huang, T. (2018, May). Survey of face detection on low-quality images. In 2018 13th IEEE International Conference on Automatic Face & Gesture Recognition (FG 2018) (pp. 769-773). IEEE.

[7] Sharifara, A., Rahim, M. S. M., & Anisi, Y. (2014, August). A general review of human face detection including a study of neural networks and Haar feature-based cascade classifier in face detection. In 2014 International Symposium on Biometrics and Security Technologies (ISBAST) (pp. 73-78). IEEE.

[8] Bilgic, B., Horn, B. K., & Masaki, I. (2010, June). Efficient integral image computation on the GPU. In 2010 IEEE Intelligent Vehicles Symposium (pp. 528-533). IEEE.

[9] "Opencl vs CUDA". [Online] Available at: https://create.pro/opencl-vs-cuda/, [Last Access: 12/6/2020];

[10] Derek Anderson, "Processing organization scheme and memory layout for the GPU using CUDA". [Online] Available at: https://www.researchgate.net/figure/Processing-organization-scheme-and-memory-layout-for-the-GPU-using-CUDA-Global-Constant_fig1_221399113, [Last Access: 25/7/2020];

[11] Pipat Methavanitpong, "Heterogenous Parallel Programming". [Online] Available at: https://www.slideshare.net/pipatmet/hpp-week-1-summary, [Last Access: 12/10/2020];

[12] NVIDIA Corporation, "NVIDIA CUDA C programming guide," 2019, version 10.2. [Online] Available at: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, [Last Access: 9/8/2020];

[13] NVIDIA Corporation, "GPU Gems 3 – Chapter 39". [Online] Available at: https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda, [Last Access: 6/9/2020];

[14] NVIDIA Corporation, "NVIDIA Turing architecture in depth". [Online] Available at: https://developer.nvidia.com/blog/nvidia-turing-architecture-in-depth/, [Last Access: 12/6/2020];

[15] NVIDIA Corporation, "NVIDIA CUDA C++ best practices guide," 2019, version 10.2. [Online] Available at: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html, [Last Access: 28/7/2020];

[16] NVIDIA Corporation, "Fundamental Optimizations in CUDA". [Online]. Available: http://developer.download.nvidia.com/GTC/PDF/1083_Wang.pdf, [Last Access: 24/9/2020];

[17] Nvidia, "NVIDIA Jetson TX2: Thermal Design Guide," [Online] Available at: http://developer.nvidia.com/embedded/dlc/jetson-tx2-thermal-design-guide, May. 1, 2017, [Last Access: 19/9/2020];

[18] NVIDIA, "Nvidia turing gpu architecture," 2018. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/ technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.Pdf, [Last Access: 24/9/2020];

[19] M. Harris, "How to optimize data transfers in cuda c/c++," December 2012. [Online]. Available: https://devblogs.nvidia.com/ how-optimize-data-transfers-cuda-cc/, [Last Access: 17/8/2020];

[20] “Central Processing Unit”. [Online] Available at: https://en.wikipedia.org/wiki/Central_processing_unit , [Last Access: 12/1/2020];
[21] Carla Tardi, “Moore’s Law” [Online]. Available: investopedia.com/terms/m/mooreslaw.asp , [Last Access: 17/4/2020];
[22] Benhallou, K., Kech, M., Ouamri, A., & Benhallou, K. An efficient face detection based on improved viola and jones. International Journal of Engineering and Technology,[Online]. Available: http://ijens. org/Vol_14_I_03/147803-2626-IJET-IJENS. pdf.
[23] Cho, J., Benson, B., Mirzaei, S., & Kastner, R. (2009, July). Parallelized architecture of multiple classifiers for face detection. In 2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors (pp. 75-82). IEEE.
[24] Kong, J., & Deng, Y. (2010, August). GPU accelerated face detection. In 2010 International Conference on Intelligent Control and Information Processing (pp. 584-588). IEEE.
[25] Bhatia, A. R., Patel, N. M., & Chauhan, N. C. (2016, October). Parallel implementation of face detection algorithm on GPU. In 2016 2nd International Conference on Next Generation Computing Technologies (NGCT) (pp. 674-677). IEEE.
[26] Wai, A. W. Y., Tahir, S. M., & Chang, Y. C. (2015, November). GPU acceleration of real time Viola-Jones face detection. In 2015 IEEE International Conference on Control System, Computing and Engineering (ICCSCE) (pp. 183-188). IEEE.
[27] HB fredj, S. Sqhair, C. Souani (2020, September). An Efficient Parallel Implementation of Face Detection System Using CUDA. In 2020 5th International Conference on Advanced Technologies For Signal and Image Processing (ATSIP). IEEE.
[28] Shivashankar J. Bhutekar et. al, “Parallel Face Detection and Recognition on GPU,” International Journal of Computer Science and Information Technologies Vol. 5 (2) , pp. 20132018, 2014.
[29] Ren Meng et. al, "Acceleration Algorithm for CUDA-based Face Detection,”2013 International Conference on Signal Processing, Communication and Computing, 2013, pp 1-5.