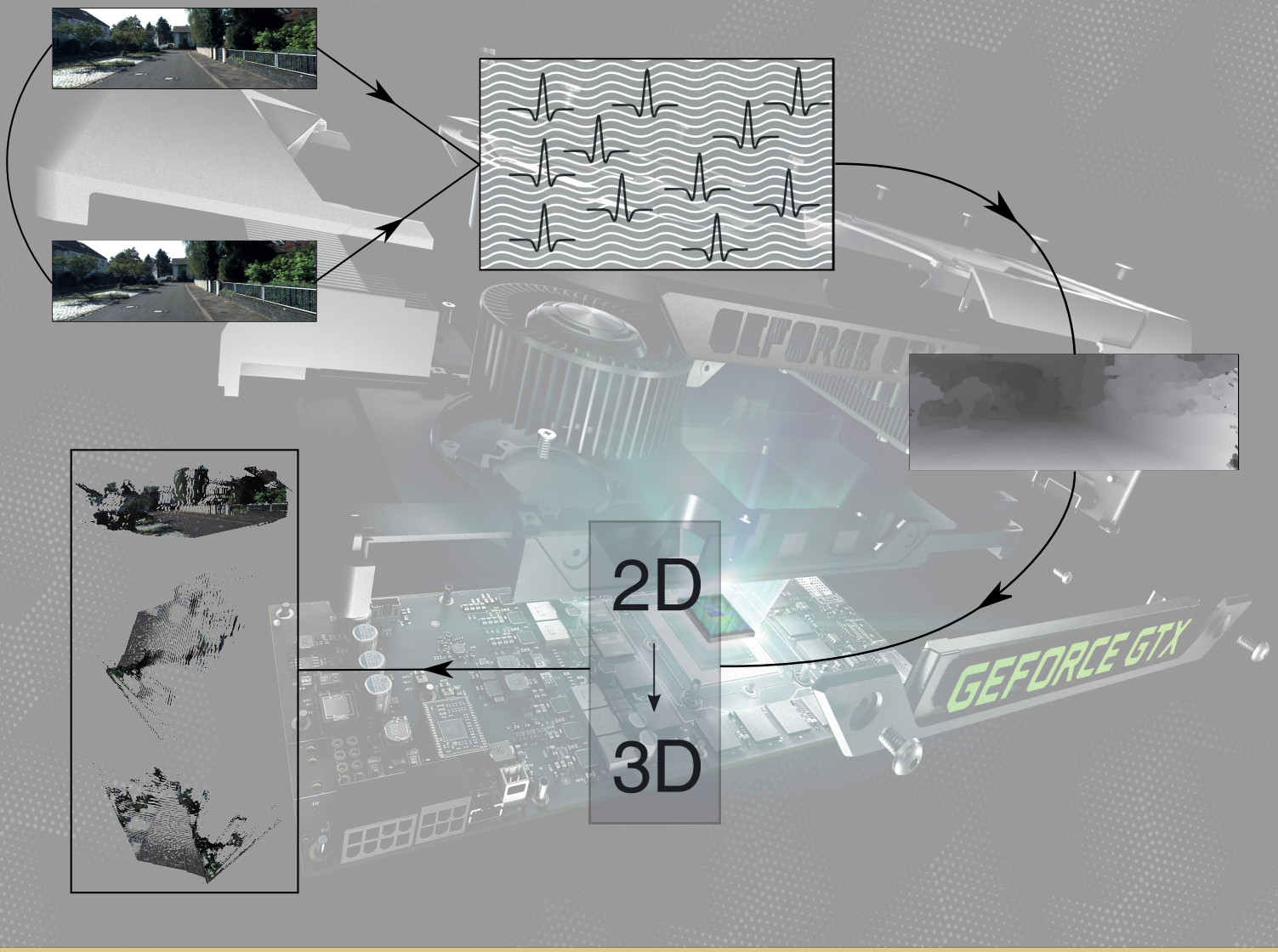


Ricardo Gabriel Lebre Calhau dos Santos Ralha

Real-time symmetry-based dense stereo matching for 3D reconstruction

Dissertação submetida para a satisfação parcial dos requisitos do grau de
Mestre em Engenharia Electrotécnia e de Computadores, Área de Especialização em Computadores

Fevereiro de 2015

· U C ·

Universidade de Coimbra

Real-time symmetry-based dense stereo matching for 3D reconstruction

Ricardo Gabriel Lebre Calhau dos Santos Ralha

Dissertação para obtenção do Grau de Mestre em

Engenharia Electrotécnica e de Computadores

Orientador:	Doutor Gabriel Falcão Paiva Fernandes
Co-Orientador:	Doutor João Pedro de Almeida Barreto

Júri

Presidente:	Doutor Vítor Manuel Mendes da Silva
Orientador:	Doutor Gabriel Falcão Paiva Fernandes
Vogal:	Doutor Nuno Miguel Mendonça da Silva Gonçalves

Fevereiro de 2015

Agradecimentos

Queria começar por agradecer ao meu orientador, o Professor Gabriel Falcão, que acreditou em mim e me motivou ao longo destes dois longos semestres, ao ”Jamaro” pela contínua paciência que apresenta a aturar-me e por tantas vezes ter ficado a lanchar sozinho e ao João Andrade pelas lições improvisadas e pelo auxílio nos momentos de aperto.

A todos os meus colegas de laboratório pelo fantástico ambiente de trabalho criado e por não me deixarem esquecer que a minha tese ia ser apresentada em Janeiro, o meu muito obrigado.

Agradeço também aos meus melhores amigos que me acompanharam nas várias fases da minha vida. Apesar das várias mudanças, ainda estamos juntos para o que der e vier.

Uma palavra para os meus colegas de curso que me ”integraram” e me ajudaram ao longo destes 5 anos. São muitos jantares, muitos convívios, muitos momentos que nunca esquecerei.

Finalmente, queria agradecer à minha família. Esteve sempre do meu lado nos momentos difíceis e ajudou-me a tomar as decisões que me levaram a estar aqui hoje. Estou-vos para sempre grato.

Esta dissertação foi realizada no âmbito dos projectos ”RECI/EEI-AUT/0181/2012 - AMS-HMI12: Apoio à Mobilidade Suportada por Controlo Partilhado e Interfaces Homem-Máquina Avançados”, ”Centro-07-ST24-FEDER-002028: Diagnosis and Assisted Mobility for People with Special Needs” financiados pela Fundação para a Ciência e Tecnologia (FCT), FEDER e programas QREN e COMPETE. Foi também realizada no âmbito da Unidade UID/EEA/50008/2013, com o apoio financeiro do quadro financiamento que estiver em vigor (FCT/MEC através de fundos nacionais e, quando aplicável, co-financiado pelo FEDER, no âmbito do Acordo de Parceria PT2020).

Abstract

SymStereo is a novel matching cost function capable of evaluating the possibility of two pixels being a match by measuring symmetry, unlike traditional dense stereo estimation algorithms that use photo-similarity. Particularly, the LogN variant of the SymStereo framework provides encouraging results when dealing with slanted surfaces. Nonetheless, the computational complexity required by this technique is problematic. Therefore, this thesis proposes a fully functional real-time parallel pipeline that uses dense stereo based photo-symmetry to process 3D maps for images with slant-dominated surfaces. This is of interest for multiple areas in computer vision involved in scene reconstruction of urban data sets and also for tracking in robotics or intelligent autonomous vehicles. The analysis and manipulation of the various matching cost and aggregation parameters resulted in real-time reconstructions with higher image quality for images with slanted surfaces. Also, a multiple Graphics Processing Unit (GPU) parallelization pipeline is proposed, which is capable of calculating from 2 up to 132 volumes per second for high- and low-resolution images, respectively.

Keywords

Dense stereo estimation, SymStereo, 3D Reconstruction, High resolution images, Parallel Processing, Multiple-GPU processing

Resumo

O SymStereo é uma nova ferramenta que, através de foto-simetria, avalia a possibilidade de dois pixeis serem correspondentes, ao contrário dos métodos tradicionais de estimação estéreo densa que utilizam foto-similaridade. Em particular, a variante LogN do SymStereo obtém bons resultados para imagens com superficies com inclinação. No entanto, a complexidade computacional desta técnica pode ser problemática. Desta forma, é proposto um pipeline, paralelo e em tempo real, que utiliza foto-simetria baseada em estimação estéreo densa para processar mapas 3D para imagens com superfícies dominadas por slant. Este estudo é de interesse para várias áreas de visão por computador como a reconstrução de zonas urbanas e na reconstrução de mapas para sistemas robóticos ou veículos autónomos. A análise e manipulação dos vários parâmetros, tanto do LogN como do capítulo da agregação, resultaram em reconstruções em tempo real com qualidade superior para imagens com inclinação. O pipeline apresentado é constituído por múltiplas GPU e é capaz de calcular 2 e 132 volumes por segundo para imagens de alta e baixa resolução, respectivamente.

Palavras Chave

Estimação estéreo densa, SymStereo, Reconstrução 3D, Imagens de alta resolução, Processamento paralelo, Processamento em múltiplas GPU

Contents

List of Figures

List of Tables

List of Acronyms

API	Application Programming Interface
ALU	Arithmetic and Logic Unit
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DSI	Disparity Space Image
DRAM	Dynamic Random Access Memory
DTFT	Discrete-time Fourier transform
GPU	Graphics Processing Unit
IDTFT	Inverse discrete-time Fourier transform
OpenCL	Open Computing Language
OS	Operating System
PC	Personal Computer
PCIe	Peripheral Component Interconnect Express
RAM	Random Access Memory
ROM	Read Only Memory
SIMT	Single Instruction Multiple Thread
SM	Streaming Multiprocessor
SP	Streaming Processor

1

Introduction

Contents

3D views can be obtained from depth maps and depth maps can be calculated with two (sparse or dense stereo) or more images (multiview stereo). Sparse stereo extracts potentially matchable image locations (edges or object discontinuities) and then searches for corresponding locations in the other image [1], while multiview and dense stereo try to find the corresponding pixels for every pixel in the reference image [2, 3].

Regarding dense stereo, the matching cost function assesses the likelihood of two pixels being in correspondence. Most matching functions use a technique that measures photo-similarity (or dissimilarity) between two stereo images to compute the matching costs. However, a recent study proposed the measure of symmetry instead of photo-similarity [4]. This method is named SymStereo and originated a new family of symmetric dense matching cost functions.

1.1 Motivation

SymStereo is implemented in three functions: *SymBT* (modification of the *BT* metric [5] for measuring symmetry instead of similarity, *SymCen* (non-parametric symmetry metric based in the *Census* [6] transform) and *logN* (uses a bank of log-Gabor wavelets for quantifying symmetry) [4]. The SymStereo variant *logN* outperforms the aforementioned functions in the particular case of images with slanted surfaces [4].

Despite the good results, the *logN* metric is a compute-intensive algorithm when running on a conventional Central Processing Unit (CPU). Therefore, we propose to accelerate the code by introducing support for parallel computing on Graphics Processing Units (GPUs). Currently, there are several frameworks that allow performing parallel computing, namely the Compute Unified Device Architecture (CUDA) [7] and the Open Computing Language (OpenCL) [8]. CUDA is exclusive to Nvidia's GPUs while OpenCL is supported by a vast set of processors. Despite being limited to a number of GPUs, CUDA is highly optimized to Nvidia's architectures and typically performs 10 to 30% better than OpenCL on the same task [9]. The CUDA framework was used to parallelize the *logN* metric, allowing superior throughput performance when compared to its sequential counterpart. The main motivation consists of achieving more than one scene processed per second, while maintaining the high quality of the 3D reconstructed scene.

1.2 Objectives

This thesis focus on developing a hybrid dual-GPU real-time stereo pipeline for the construction of 3D maps using SymStereo. In order to improve the quality of reconstructed scenes, visual enhancement post-processing algorithms such as local aggrega-

tion, left-right consistency check and occlusion pixel filling were designed to obtain depth maps with less imperfections and, consequently, higher-quality 3D maps. The complete pipeline is parallelized using the CUDA computing language, in order to fully exploit the processing power of two top performer Nvidia GTX Titan GPUs.

1.3 Main contributions

One kind of autonomous system that typically has to process large amounts of images to make decisions about odometry and trajectory is the robot. Many of those images, in particular for indoor systems, have high levels of slant. The real-time processing of these images allows the system to make decisions also in real-time, assuming, this way, a level of importance which depends on the motion requisites of the autonomous vehicle.

By taking advantage of the processing capabilities of two GPUs and one CPU, a full working real-time pipeline is presented, capable of calculating up to 2 and 132 volumes/scenes per second for high- and low-resolution images, respectively. Also, the log-Gabor parameters were refined, namely the number of scales, shape-factor, scaling step and center frequency of the mother wavelet, as well as the aggregation window, for performing an exhaustive generation of disparity maps. Therefore, an extensive analysis regarding quality versus processing time was performed.

This work has produced two manuscripts. One is already accepted for publication, while the other has been submitted in March 2015:

- Ricardo Ralha, Gabriel Falcao, Joao Andrade, Michel Antunes, Joao Pedro Barreto, and Urbano Nunes, ”Distributed dense stereo matching for 3D reconstruction using parallel-based processing advantages” [Accepted] in IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2015;

- Ricardo Ralha, Gabriel Falcao, Joao Amaro, Vasco Mota, Michel Antunes, Joao Pedro Barreto, and Urbano Nunes, ”Parallel refinement of 3D slanted scenes using dense stereo induced from symmetry” [Submitted] in IEEE Transactions on Circuits and Systems for Video Technology (TCSVT), 2015.

1.4 Dissertation outline

This thesis consists of 6 chapters. After the introduction, chapter 2 features the GPU and CPU architectures and the functioning of the CUDA framework. Chapter 3 focuses on the basic principles of stereo algorithms and the theory that involves the development of the *logN* algorithm. In chapter 4, the implementation and parallelization of the stereo

pipeline on the GPU is explained. In chapter 5, the experimental results are exposed and chapter 6 presents some conclusions about the work developed under the scope of this thesis.

2

The GPU architecture and CUDA Programming Model

Contents

In the early 2000's, Central Processing Units (CPUs) had reached physical and clock speed walls. This led to a change in the CPU design paradigm [10]. Meanwhile, Graphics Processing Units (GPUs) were only used to perform multimedia tasks like video playbacks and graphics rendering. The parallel computing nature of these devices encouraged the development of novel parallel programming interfaces that allowed exploiting GPUs for scientific computation as well.

This chapter discusses Intel's Haswell CPU and Nvidia's Kepler GPU architectures and also the Compute Unified Device Architecture (CUDA) framework.

2.1 The CPU architecture

In 1965, "Moore's Law" [11] defined the growth trajectory for the hardware and semiconductor industries. But as technology evolved, new challenges arose. The further increasing of the transistor's clock speed eventually stalled and the rising number of transistors in CPUs led to heat dissipation constraints. Thus, to solve these problems, the industry opted to change the process design established and started combining cores. With the new multicore solutions, new mechanisms were introduced. For example, multiprocessing enabled the Operating System (OS) to schedule processes to different cores, distributing the workload. This was the industry's solution to continue supplying the ever demanding Personal Computer (PC) users market.

In this section, the Haswell CPU Architecure, represented in Fig. 2.1, is addressed since it is the architecture of the host system used to perform experimental tests in this thesis - Intel's i7 4790k.

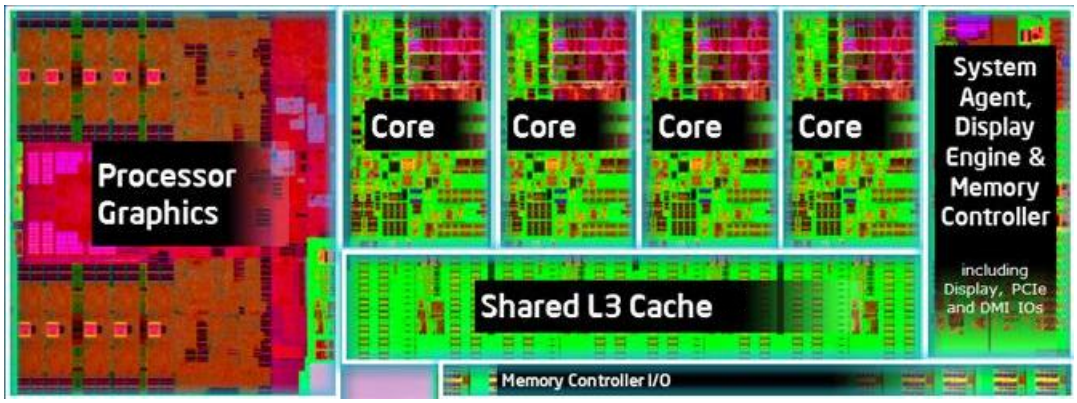


Figure 2.1: Haswell Generic Architecture. 4 cores, processor graphics and a shared L3 cache between them [12].

2.1.1 The x86-64 multicores

There are two important mechanisms that make up each one of the CPU cores, the control unit and the Arithmetic and Logic Unit (ALU). The L1 and L2 caches that complete each core are addressed in the next segment.

Control unit The control unit is responsible for interpreting the instructions that are passed on to that core. By decoding the instruction, the control unit can, for example, fetch other instructions from the memory or discover the memory addresses of the operands for an operation and send the data to the arithmetic and logic units, so that the desired operation may be performed.

ALU The ALU is responsible for the arithmetic operations that occur on that core including additions, subtractions, multiplications, bit shifting, logic operations such as AND, OR, NOT and boolean comparisons. By supporting floating-point operations, ALUs are complex components that enable a large array of operations.

2.1.2 System memory hierarchy

One of the main bottlenecks with computers is the communication latency that exists between the CPU and main memory. With the objective of reducing the impact of this problem, a memory hierarchy exists in the system. With the introduction of the multicore architecture, this memory hierarchy, which started fairly simple, is now a complex mechanism that greatly enhances the user experience. The memory hierarchy can be split in two regions: top level and bottom level.

Top level On the top of the hierarchy there are three cache levels. The L1 and L2 caches are exclusive to each core and have 64 KB and 256KB of capacity, respectively. The L1 cache is faster than the L2 cache seeing that, commonly, the memory capacity is inversely proportional to its access speed. The L3 cache is a shared memory between all cores and has a capacity of 8MB. This cache is bigger but slower than the two previously mentioned ones. Despite the small capacity, due to space limitations, these caches make-up in speed as the CPU access time is much faster compared to the time that it takes to access the main memory.

Bottom level On the bottom of the hierarchy is located the main memory, four 8GB Dynamic Random Access Memory (DRAM) sticks that combine for a 32GB memory. Besides the main memory, the system also has a Read Only Memory (ROM) containing

important data to be used when loading the OS. While Random Access Memorys (RAMs) are volatile (contents are lost when the machine is powered off) and allow the CPU to read and write at will, ROMs are non-volatile (retain contents after the machine is powered off), read-only memories.

2.2 The GPU architecture

By demanding increasingly complex graphics in real-time, the gaming industry defines the growth pace of GPUs. This leads to more threads running simultaneously, more cores and more efficient memory control. As a result of this evolution, the gap between CPUs and GPUs is increasing each year. Nevertheless, it is still possible (and desirable) for both of them to coexist and work together [10]. The CUDA programming model was invented by Nvidia to support the joint CPU/GPU execution on Nvidia's devices [7].

In this section, the Kepler GPU Architecure, shown in Fig.2.2, is explained since it is the architecture of the two GeForce GTX Titan GPUs used in this thesis.

Figure 2.2: Kepler Generic Architecture, 14 Streaming Multiprocessors with the L2 cache [13].

2.2.1 Streaming Multiprocessors

GPUs are composed by Streaming Multiprocessors (SMs). Each SM is formed by the basic processing unit of a GPU, Streaming Processors (SPs) or CUDA cores. Besides CUDA cores, a SM also possesses its own control unit, for decoding and issuing instructions and to schedule the threads' execution and on-chip memory composed by shared memory and L1 cache. The SM is able to process several threads concurrently, issuing and executing the same instructions for a great number of threads and performing similar work on different data elements. The GeForce GTX Titan, using the Kepler architecture, has 14 SMs with 192 CUDA cores per SM. Regarding kernel execution, Nvidia follows a Single Instruction Multiple Thread (SIMT) policy. It uses the warp, a batch of 32 threads that execute in parallel. Such grouping exists to dispatch the same instruction to all 32 threads, following a common datapath.

2.2.2 Memory Hierarchy

There are two regions of memory distinguishable in a GPU device: the on-chip memory and the off-chip memory.

On-chip memory The on-chip memory consists of the shared memory and L1 cache. In this architecture, each SM has a $64kB$ on-chip memory that can be configured as $48kB$ of shared memory and $16kB$ of L1 cache, the other way around or as $32kB/32kB$. Thus, the memory can be adjusted and suited for any application's needs.

Besides this, each SM also has a read-only data cache that is $48kB$ long. Fig.2.3 depicts the on-chip memory.

Off-chip memory When transferring data to the GPU, the CPU can only interact with the off-chip memory. The off-chip memory includes global memory and constant memory. Constant memory is a cached $64kB$ read-only memory region, useful for allocating initialized values that are accessed several times without need for update, while global memory is a $6GB$ read-write memory that can transfer data with any thread on the GPU.

Besides the off-chip memory, the GPU also has a $1536kB$ L2 cache. The off-chip memory can be seen in Fig.2.4.

2.2.3 Scalability and Scheduling

An application developed to run on a GPU has to be partitioned in order to be processed by the device. The lowest level of the hierarchy is the thread, an instance that executes the same code segment applied to different data elements. A group of threads

form the mid-level of the hierarchy which is named a block. And finally, a group of blocks forms the highest level, a grid. This hierarchy is observable in Fig.2.5.

The blocks created have to be distributed to the SMs to be scheduled (Fig.2.6). Since a SM dispatches warps of 32 threads, the number of threads per block has to be a factor of 32. Only this way it is possible to achieve the best occupation. Each cycle, the SM dispatches a number of warps equal to the number of blocks allocated to the SM.

Figure 2.3: Kepler Streaming Multiprocessor model with 192 CUDA cores, the on-chip memory and read-only data cache [13].

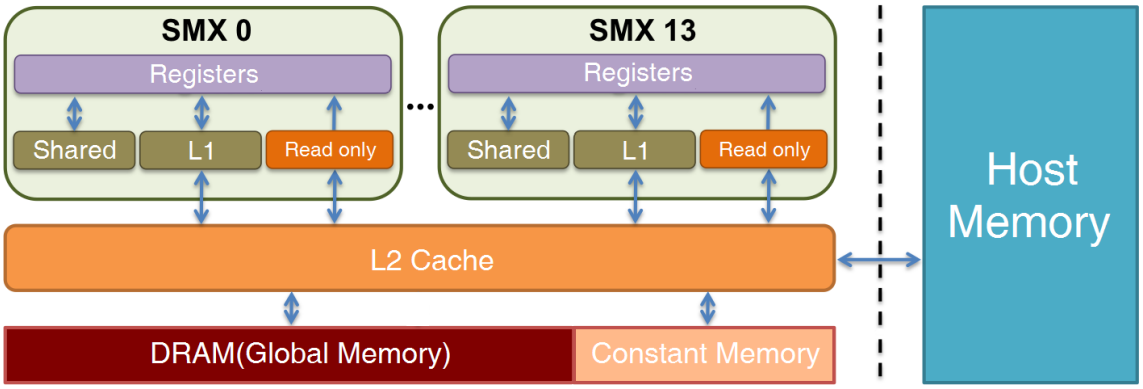


Figure 2.4: Kepler off-chip and on-chip Memories.

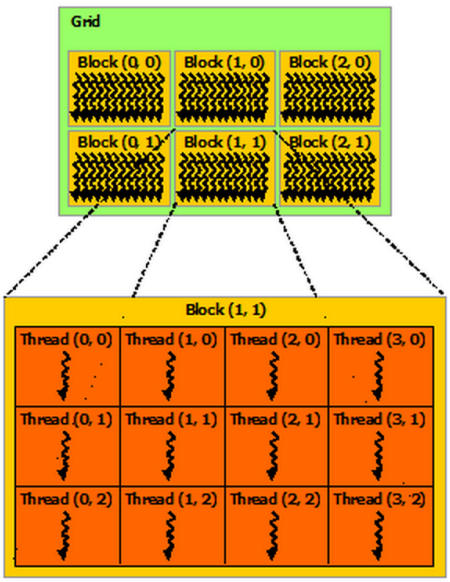


Figure 2.5: Grid organization [7].

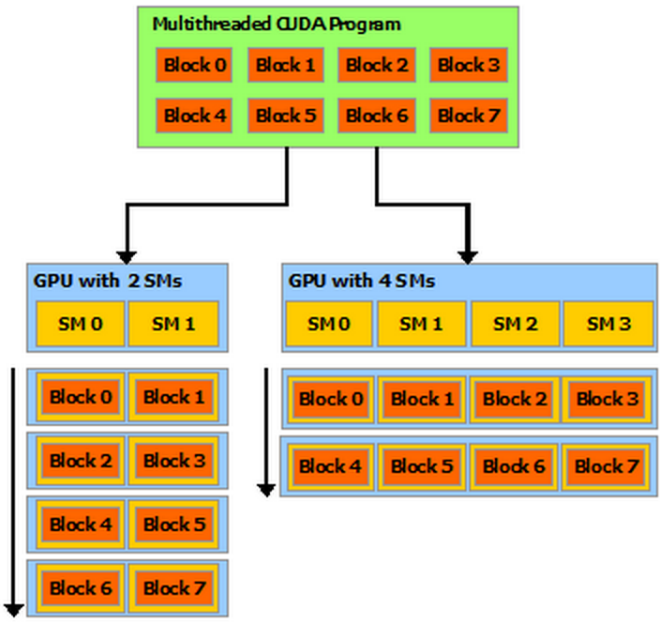


Figure 2.6: Block execution on different generation devices [7].

2.3 The CUDA Programming Model

In order to turn GPUs into fully programmable devices, programming languages were necessary. With the objective of having a dedicated language, Nvidia created the CUDA programming model, which only works with the company's devices, like the GeForce GTX Titans used in this thesis. To achieve maximal efficiency, the user must be aware of the capacities of the device. This way, the CUDA programming language can be seen as a high-level language that requires a low hardware abstraction by the user.

In this section, a simple example is used to explain the working flow of CUDA. The example consists of a parallel summation between two single floating-point tables (A and B) with the result being stored in matrix C.

2.3.1 Program Execution

CUDA applications can de divided in two parts, one than runs on the host, the CPU, and another one that runs on the device, the GPU. While the host runs the part responsible for memory allocations and flow control, the device processes the parallel kernels, where data operations are made.

CUDA host code Fig.2.7 shows a typical scheme of the host section code of a CUDA application. If there is more than one compatible device, CUDA allows the programmer to choose the device to work with. By default, device 0 is chosen. Initially, it is necessary to allocate space in the device's global memory. Afterwards, the data segment in the host is transferred to the device. This way, when the kernel is called, the GPU already has the data available to perform the necessary calculations. After kernel execution, the CPU orders the data back to its memory. Finally, the memory space allocated needs to be deallocated.

CUDA device code The GPU is responsible for running the parallel kernels. Basically, kernels specify the amount of work each thread performs. In Fig.2.8 it is possible to obverse a kernel that performs the matrix summation.

The kernel is build in a way that allows each thread to be responsible for an index of the matrix. If memory accesses are coalesced (data pre-aligned on the device's memory), all indexes are calculated and stored at the same time.

```
// Host code

int main()
{
    int i;
    int N = 1024;
    size_t size = N * sizeof(float);

    // Allocate vectors in host memory
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);
    float* h_C = (float*)malloc(size);

    // Initialize input vectors
    for(i=0;i<N;i++)
    {
        h_A[i]=i*3;
        h_B[i]=i*4;
    }

    // Allocate vectors in device memory
    float* d_A;
    cudaMalloc(&d_A, size);
    float* d_B;
    cudaMalloc(&d_B, size);
    float* d_C;
    cudaMalloc(&d_C, size);

    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 64;
    int blocksPerGrid = N/threadsPerBlock;
    dim3 dimBlock(threadsPerBlock,1,1);
    dim3 dimGrid(blocksPerGrid,1,1);

    VecAdd<<<dimGrid, dimBlock>>>(d_A, d_B, d_C, N);

    // Copy results from device memory to host memory
    // h_C contains the results in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    // Free host memory
    free(h_A);
    free(h_B);
}
```

Figure 2.7: Matrix summation host code [7].

```
// Device code - Kernel definition

__global__ void VecAdd(float* A, float* B, float* C, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    if (i < N)
        C[i] = A[i] + B[i];
}
```

Figure 2.8: Matrix summation device code [7].

2.3.2 GPU efficiency

One of the objectives of the programmer is to achieve the best efficiency from the GPU. Occupancy can measure this efficiency. It can be defined as

$$\text{Occupancy} = \frac{\text{Number of Active Warps per SM}}{\text{Maximum Number of Active Warps per SM}}. \tag{2.1}$$

Occupancy limiting factors Three factors can limit occupancy: the number of threads per block, the number of registers per thread and shared memory per block. The programmer is responsible to find the best combination of threads per block that can maximize the occupancy. For example, being a first rate GPU, the GeForce GTX Titan has 2^{16} 32 bit registers per SM. Let's assume a configuration with 256 threads per block, 32 registers per thread and 4 KB of shared memory per block. The occupancy here is 100% seeing that the maximum number of active warps this GPU allows is 64 and there are 64 warps running. Now, consider each thread uses 64 registers. The occupancy would drop to 50% since the maximum number of active warps could only be 32. In this case, not even changing the number of threads per block would help, as the maximum occupancy for 64 registers per thread is 50%. Presume now the initial configuration but with 8 KB of shared memory per block. The occupancy would be 75%, with 48 active warps. If the number of threads per block is risen to 512, the occupancy returns to 100%, with less active blocks but enough shared memory for all of them.

There are various configurations that can be used by the programmer to maximize the efficiency of the device. It all depends on the application's needs and the capacity of the programmer to take full advantage of the resources at his/her disposal.

2.3.3 Code optimization techniques

The real challenge of developing parallel CUDA code programs lies in using all the resources available on the device in the most efficient way. In order to do so, there are known techniques that can significantly enhance the program's performance:

- Shared memory: As mentioned before, shared memory can be used by all threads scheduled to a SM. When using it, the data handled by the kernel must be transferred from the GPU's main memory to the SM's shared memory. It is smaller and faster which allows for reduced access times, compared to the device's main memory;

- Pinned memory allocations: Data transfers from CPU to GPU can have a significant impact on the final processing time. It is fundamental to minimize this impact in order to achieve better results. When allocating data in the host memory, the system makes a pageable allocation by default. However, the device is not able to transfer data directly from pageable memory. To transfer that data, a temporary pinned array is created on the host memory so that the data can be transferred from the pageable section to the array and only then it is transferred to the device memory [7]. To avoid this, pinned allocations in the host can be made, saving time in data transfers;

- Thread Divergence: Divergent conditions, such as the if statements, can influence the final results. When in the presence of a conditional statement, the warp has to verify both paths in all threads, adding one more cycle. This way, whenever possible, it is important to avoid divergent branches;

- Loop Unrolling: In the presence of a loop, the warp needs to verify, each cycle, the established condition to perform a jump. By unrolling the loop, these problems are avoided. The unrolling can be made manually or by the CUDA compiler;

- Coalesced Memory Accesses: When all the threads in a warp perform a memory load, the device detects if the addresses being read are consecutive or separated by a constant multiple of 16 or 32. Thanks to the Peripheral Component Interconnect Express (PCIe) 3.0, with a bus of 128-bit, used by most GPUs, multiple memory accesses can be coalesced to a single memory access, decreasing the number of memory transfers;

- Asynchronous Memory Transfers: By using CUDA streams, it is possible to do memory transfers while executing kernels. This increases the throughput of the program, minimizing the impact of memory transactions.

2.4 Conclusions

This chapter describes the architectures that compose the host and devices. The Haswell CPU architecture is an efficient architecture composed of several (2, 4, 6 or 8) cores and a memory hierarchy. It is mainly used to process serial applications but is also prepared for parallel computing. The Kepler GPU is a highly parallel architecture comprised of 14 SMs with 192 CUDA cores each and a memory hierarchy to support efficient data processing.

CUDA is a highly optimized programming language developed by Nvidia for exploiting general-purpose processing on GPUs. Programmers need specific knowledge about the Kepler architecture in order to develop code that allows an efficient exploitation of all the resources available.

The next chapter focuses on the various phases of dense stereo algorithms and the logN metric in particular, which presents a complex and thus compute-intensive nature.

3

Stereo Algorithms and the logN metric

Contents

This Chapter discusses dense stereo algorithms and all the steps they usually comprise to generate a 3D map. Since it is an integral part of this work, the *logN* algorithm is discussed in detail.

This way, the reader is given an overview of the principles of stereo algorithms. If the reader is interested in knowing more, he can refer itself to [14].

3.1 Stereo Algorithms Phases

Stereo algorithms generally include the following steps:

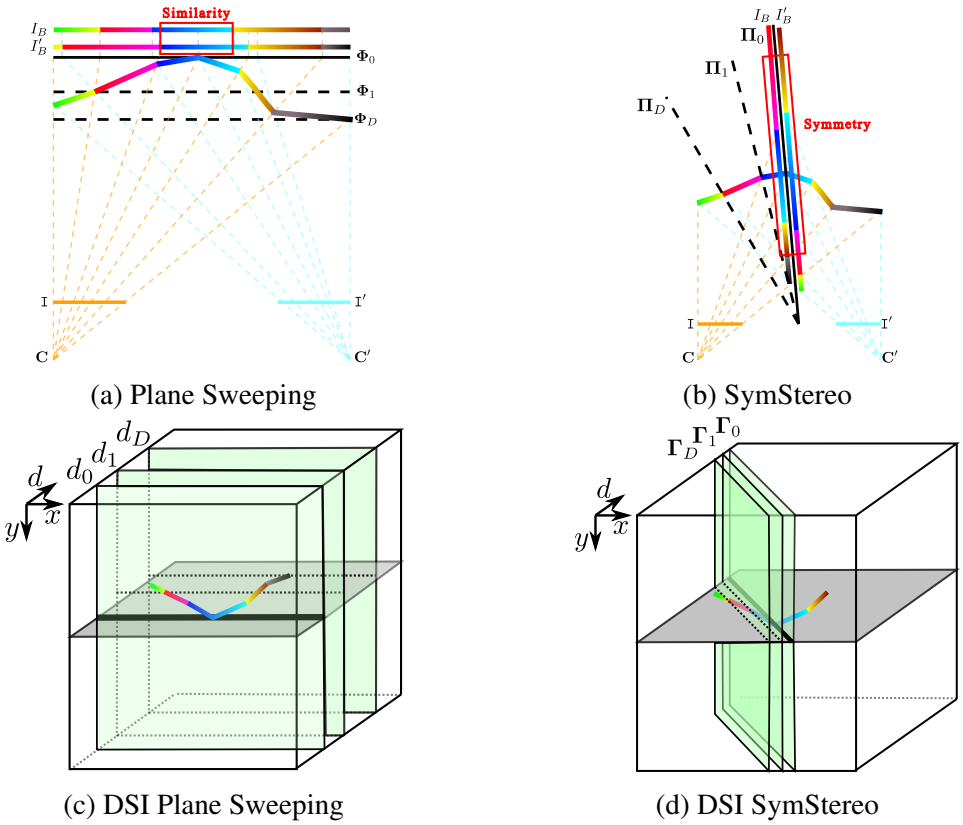
1. Matching cost computation;
2. Cost (support) aggregation;
3. Disparity computation;
4. Disparity refinement;
5. Disparity to 3D maps conversion.

This section addresses the algorithms used in each phase to calculate the final 3D map.

3.1.1 Matching cost computation - The LogN algorithm

In this first step, the matching function receives data from left and right images, which are acquired by a stereo camera. The purpose of the algorithm is to compute the matching costs by verifying the possibility of two pixels, one from each image, corresponding to each other. This evaluation is performed across all possible disparities and pixel locations. By doing this, the Disparity Space Image (DSI) [15] is created. The DSI is a 3D volume that, for each pair pixel-disparity associates the corresponding matching cost.

New advancements show that by using symmetry rather than photo-similarity to evaluate the likelihood of two pixels being a match, stereo disparity estimation improves [4]. In Fig.3.1 are represented the differences between conventional stereo matching by plane sweeping [16] and SymStereo. By examining images 3.1a and 3.1c, it is noticeable that each possible disparity d_i is associated with a virtual plane Φ_i, meaning that photo-similarity between I_b and I'_b, that are the results of back-projecting I and I' onto Φ_i, is implicitly measured by the chosen matching cost. By observing images 3.1b and 3.1d, in SymStereo, the virtual planes Π_i that pass between the cameras intersect the scene structure. Thus, the back-projection images are reflected with respect to the curve where that intersection occurs (mirroring effect). Also, each plane Π_i in 3.1b corresponds to an oblique plane Γ_i in 3.1d which means that by choosing the appropriate set of virtual cut planes Π_i, the entire DSI domain can be covered.



(a) Plane Sweeping

(b) SymStereo

(c) DSI Plane Sweeping

(d) DSI SymStereo

Figure 3.1: Plane sweeping vs SymStereo [4].

The objective of *logN* consists of calculating the matching costs. To do so, *logN* uses a bank of log-Gabor filters to detect symmetric and anti-symmetric energy. The matching cost will be the joint energy calculated by combining the symmetric energy with its anti-symmetric equivalent. The N in *logN* stands for the number of wavelet scales that are used to filter the input images.

The matching function can be divided in two phases: the filtering and the computation of the joint energy. However, before describing the algorithm, it is important to understand the geometry comprising stereo vision.

Epipolar Geometry Epipolar geometry is essential for any stereo algorithm. Fig.3.2 depicts two pinhole cameras, each one with its own center of projection, O_L for the left camera and O_R for the right camera. Each one has an image plane, A and B, that contains the projections x_L and x_R, respectively, of X, the point of interest. Since the two centers of projection are distinct, they both can be projected into the other camera's image plane (e_L is the representation of O_R in the left camera image plane and e_R is the representation of O_L in the right camera's image plane). These image points are called epipoles or epipolar points.

The line $O_L - X$ is seen as a point by the left camera because it is in line with the

center of projection. However, by the perspective of the right camera, $O_L - X$ is seen as a line in image plane B. This line, $e_R - x_R$ is called an epipolar line. Symmetrically, the line $O_R - X$ seen by the right camera as a point is seen as an epipolar line e_L–x_L by the left camera.

Epipolar geometry can be simplified if the two image planes, A' and B', coincide. In this case, both epipolar points, e'_R and e'_L, are located in the infinite. Thus, the epipolar lines e'_R–x'_R and e'_L–x'_L also coincide. Because both images are horizontally aligned, when searching for corresponding points in aligned images, the search only needs to be done along an horizontal line. If the cameras cannot be positioned in a way that allows both planes to coincide, an image rectification process is possible [17].

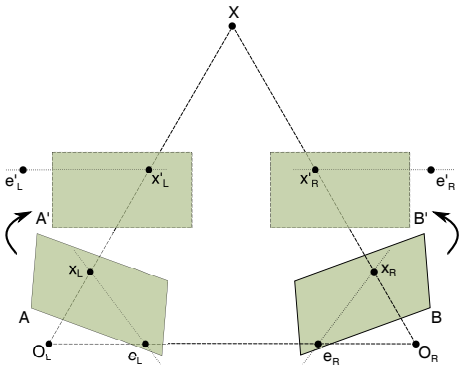


Figure 3.2: Representation of Epipolar Geometry for non-aligned and aligned image planes.

Camera alignment is important in LogN since the search for corresponding pixels is done only along an horizontal line. Thus, the algorithm input images must be captured with aligned cameras or rectified *a posteriori*.

Filtering Phase Consider two images, I and I' and two epipolar lines, one on each image, $I(q_0)$ and $I'(q_0)$. In this phase, the stereo pair will be filtered with N log-Gabor wavelets. Since these are 1D analytical filters and filtering occurs in the spectral domain, a 1D Fourier transform has to be applied to both images. Being $\mathscr{I} = \mathscr{F}(I)$ the Fourier Transform along the epipolar lines of I and G the matrix of coefficients of the filter, $\mathscr{I}.G_k$ is calculated, for a given wavelet k, where G_k represents a wavelet with the same length as the epipolar line.

In order to return to the time domain, an Inverse Fourier Transform is applied. For a general pixel location q_0 and wavelet k, it can be deduced that

$$s_k(q_0) + \boldsymbol{i}a_k(q_0) = \mathscr{F}^{-1}(\mathscr{I}(q_0).G_k), \tag{3.1}$$

$$s'_k(q_0) + \boldsymbol{i}a'_k(q_0) = \mathscr{F}^{-1}(\mathscr{I}'_f(q_0).G_k). \tag{3.2}$$

I'_f is I' flipped horizontally, as it needs to be flipped before the filtering procedure.

Energy Calculation Phase After filtering the two images, the joint energy needs to be calculated. In order to properly implement this procedure, the right-side signal needs to be shifted by an amount d_i that depends on the virtual cut plane Π_i [4]. The flipped image becomes

$$\widehat{I}(q_0) = I'_f(q_0 - d_i), \tag{3.3}$$

where d_i is a shift amount depending on the choice of the virtual plane Π_i.

With this, the symmetry (s^S and a^S) and anti-symmetry (s^A and a^A) coefficients can now be calculated for a generic epipolar line q_0 with

$$s_k^S(q_0) + \boldsymbol{i} a_k^S(q_0) = \big(s_k(q_0) + s'_k(q_0 - d)\big) + \boldsymbol{i}\big(a_k(q_0) + a'_k(q_0 - d)\big), \tag{3.4}$$

$$s_k^A(q_0) + \boldsymbol{i} a_k^A(q_0) = \big(s_k(q_0) - s'_k(q_0 - d)\big) + \boldsymbol{i}\big(a_k(q_0) - a'_k(q_0 - d)\big). \tag{3.5}$$

With the image being symmetric about the pixel location q_0, the real components s^S and s^A typically take high values and the imaginary components a^S and a^A assume small values [18]. Taking this into account, the symmetry (E^S) and anti-symmetry (E^A) energies can be established for the N wavelet scale responses

$$E^S(q_0) = \frac{\sum_{k=0}^{N-1} |s_k^S(q_0)| - |a_k^S(q_0)|}{\sum_k \sqrt{(s_k^S(q_0))^2 + (a_k^S(q_0))^2}}, \tag{3.6}$$

$$E^A(q_0) = \frac{\sum_{k=0}^{N-1} |a_k^A(q_0)| - |s_k^A(q_0)|}{\sum_k \sqrt{(s_k^A(q_0))^2 + (a_k^A(q_0))^2}}. \tag{3.7}$$

The joint energy E comes as a combination of the symmetry and anti-symmetry energies

$$E = E^A \times E^S. \tag{3.8}$$

In Fig.3.3 it is possible to observe the various stages of the *logN* metric.

Wavelet scales To understand the wavelet scales of the log-Gabor filters, the role of each parameter must be addressed. Fig. 3.4 illustrates the relation of the parameters with the space-frequency response of the filter.

There are four parameters that define the wavelets: the shape-factor Ω, the center frequency of the mother wavelet ω_0 , the scaling step s, and the total number N of wavelets. The shape-factor is related with the bandwidth of the filter and defines a contour in the (ω, σ) domain. The first wavelet scale G_0 is uniquely defined by the center frequency ω_0 and the shape-factor Ω and s sets the distance between center frequencies of successive wavelet scales along the contour. If the stereo pair is dominated by textured regions, logically, the parameters must be chosen in order to work on the top-left corner of the (ω, σ) plane where the frequency of the log-Gabor wavelets is higher. But if the pair

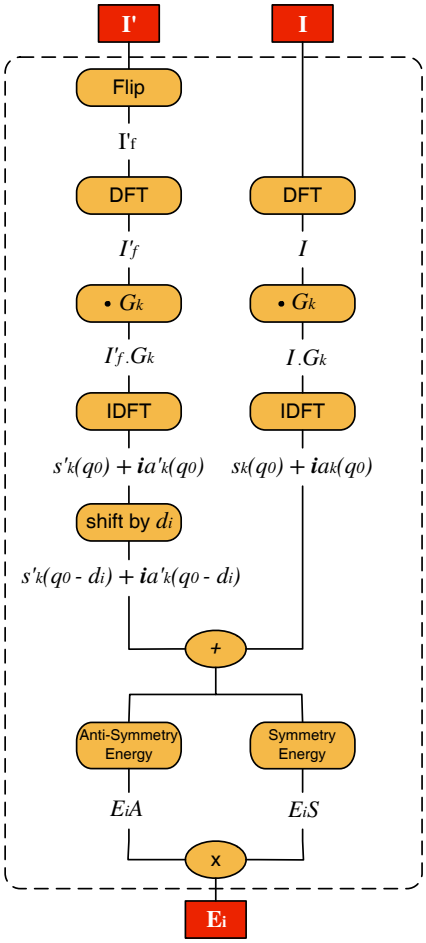


Figure 3.3: The LogN algorithm.

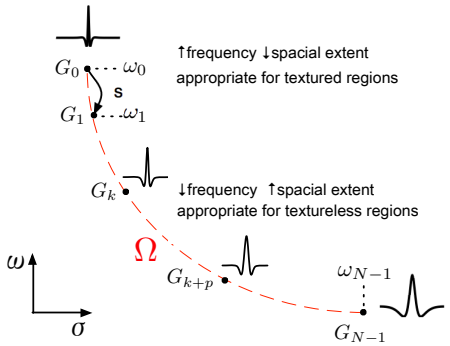


Figure 3.4: (Qualitative) space-frequency behaviour of the log-Gabor wavelets G_k [19] . The horizontal axis refers to the spatial support σ of the filter kernel, while the vertical axis concerns the response frequency ω.

is comprised mostly of textureless regions, then the work must be done on the bottom-right corner of the plane where the frequency is lower. More details on the design of the log-Gabor filters can be found in [19].

3.1.2 Cost aggregation and disparity computation

There are two types of stereo algorithms: local and global. Local algorithms rely on a window-based approach and use an aggregation step [20, 21] while global algorithms tend to solve a global optimization problem by finding the best disparity that minimizes a global cost function that is composed by data and smoothness terms [22–24]. LogN is a local stereo algorithm thus, it needs an aggregation cost step to aggregate the determined matching costs.

To estimate the correct disparity for a pixel, the sum of the matching costs is calculated over a square window. This is done for every disparity value. The final disparity chosen will be the disparity associated with the smallest sum of values calculated over the square window.

3.1.3 Disparity refinement

The disparity refinement stage is used to correct some disparities that were wrongly computed and to fill pixels that are occluded on the depth map. Occluded pixels are pixels only visible in one of the images. The circumvent of these inconveniences is divided in two sub-stages: left-right consistency check and filling of occluded pixels.

Left-Right Consistency Check Two disparity maps are necessary for this stage, one computed using the left image as reference and the other with the right image. By subtracting the disparities of corresponding pixels on each depth map, it is verifiable that if the absolute value is superior than a given threshold, then the pixel is considered occluded. For our pipeline, the threshold adopted is three.

Filling of Occluded Pixels The algorithm starts by finding the first occluded pixel, beginning in the top left corner of the disparity map. Then, a 4-way search is performed to find the first non-occluded pixel in each way, left, right, up and down. The disparity selected is the median between the four values that were found. If no value is found in one of the ways, the median is calculated between the other three values and so on. This is done for all occluded pixels in the disparity map.

The pixels that have already been filled are considered in the calculation of the disparities for the next occluded pixels. Therefore, due to these dependencies between pixels,

this stage is confined to the Central Processing Unit (CPU). The algorithm is depicted in Fig.3.5.

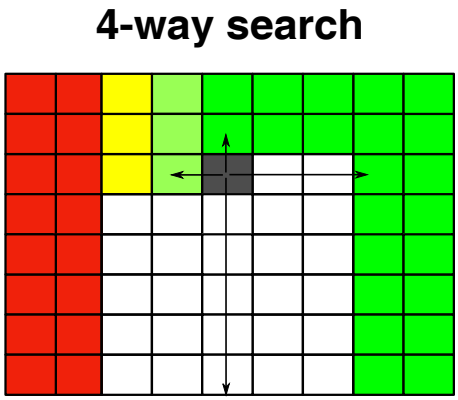


Figure 3.5: Representation of a 4-way search.

3.1.4 From Disparity maps to 3D views

To calculate the 3D volume, 2D coordinates are mapped into 3D by

$$Z = (f * b)/D, \tag{3.9}$$

$$X = ((x - c_x) * Z)/f, \tag{3.10}$$

$$Y = ((y - c_y) * Z)/f, \tag{3.11}$$

where f is the focal length (in pixels), b represents the distance between the two lens (in meters), c_x and c_y are the image centers (in pixels) and D is the disparity of the pixel.

3.2 Conclusions

This chapter provides the reader with a better understanding of stereo algorithms and all phases that they comprise. Stereo algorithms are composed of five stages: matching cost computation, cost aggregation, disparity computation, disparity refinement and disparity to 3D conversion. Of all phases, the first one is the most important. The *LogN* metric is essential in this investigation since it responds extremely well to slanted surfaces, but it presents a high computational complexity, which is incompatible with real-time execution on conventional CPUs. Therefore, it can be accelerated and conveniently supported by parallel programming.

The next chapter explains how the pipeline is parallelized and takes advantage of the many resources made available by the Graphics Processing Unit (GPU) to perform in real-time.

4

The SymStereo pipeline on the GPU

Contents

With the goal of parallelizing the pipeline of 3D volumes computation in real-time, a hybrid architecture was exploited using a Central Processing Unit (CPU) and two Graphics Processing Units (GPUs). The parallel pipeline of SymStereo is depicted in Fig. 4.1. In order to reduce the pipeline processing time, some previously mentioned code opti-

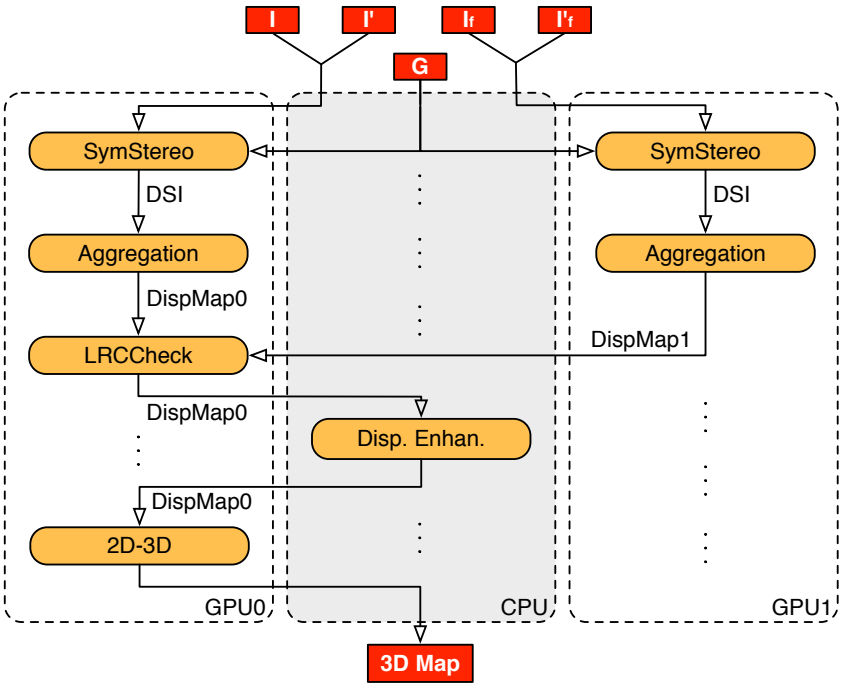


Figure 4.1: SymStereo Pipeline representation, where I and I' are the left and right images, I_f and I'_f are the left and right images flipped and G represent the Gabor coefficients.

mization techniques like pinned memory allocations, thread divergence, loop unrolling and coalesced memory accesses were extensively utilized.

The use of shared memory was also tried but since the necessary data blocks are larger than the shared memory size, that at the finest level of thread-granularity would force several memory swaps to occur, every time this performance enhancing technique is applied, the kernels' efficiency decreases due to these overheads accessing the global memory. Therefore, this option was discarded. Also, asynchronous memory transfers were not possible to apply in this work, considering that there are no memory transfers that can be made while executing any kernel.

This chapter explains the implementation of the pipeline on the machine's host and devices.

4.1 LogN parallelization

Fig. 3.3 shows the LogN algorithm. To parallelize it, three kernels were created, i) the flip kernel; ii) the filtering kernel; and iii) the energy calculation kernel. To perform the Discrete-time Fourier transform (DTFT) and Inverse discrete-time Fourier transform (IDTFT) on the GPU, Nvidia provides the optimized CUFFT Application Programming Interface (API) [25].

4.1.1 Filtering Phase on the GPU

This phase only comprises two of the three kernels mentioned previously, where each thread is associated to one image pixel. The flip kernel has half the threads of the remaining kernels. Since the input matrix is flipped horizontally, only half of the matrix width needs to be swept. Regarding the filtering kernel, each line of the input spectrum has to be multiplied by a line of the log-Gabor coefficients matrix. This way, the output of the kernel consists of N matrices used in the posterior energy calculation phase. This is illustrated in Fig. 4.2. The quantity of data processed in this kernel depends on the number of wavelets chosen for the filter. The larger the number of wavelets, the larger the filter becomes.

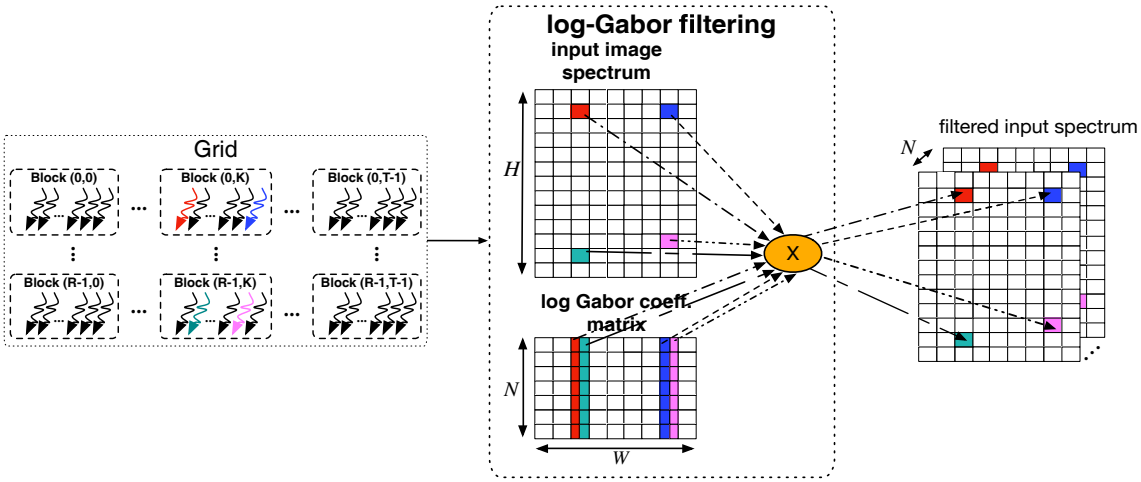


Figure 4.2: Representation of the log-Gabor kernel. (T,R) are the number of blocks in the (x,y) directions, W and H are the width and height of the input images, respectively and N is the number of wavelets.

4.1.2 Energy Calculation on the GPU

This operation represents a time consuming phase that depends on the image dimensions, the disparity range and the number of scales chosen. The inputs are the images after

being filtered by the log-Gabor wavelets, $\mathscr{F}^{-1}(\mathscr{I}.G)$ and $\mathscr{F}^{-1}(\mathscr{I}'_f.G)$, as depicted in Fig. 3.3, and the output is the Disparity Space Image (DSI). Each thread processes the energy for each pixel and the corresponding disparity, i.e. the symmetry (E^S) and anti-symmetry (E^A) energies as established in (3.6) and (3.7). Finally, the joint energy is determined. The energy calculation kernel is depicted in Fig. 4.3.

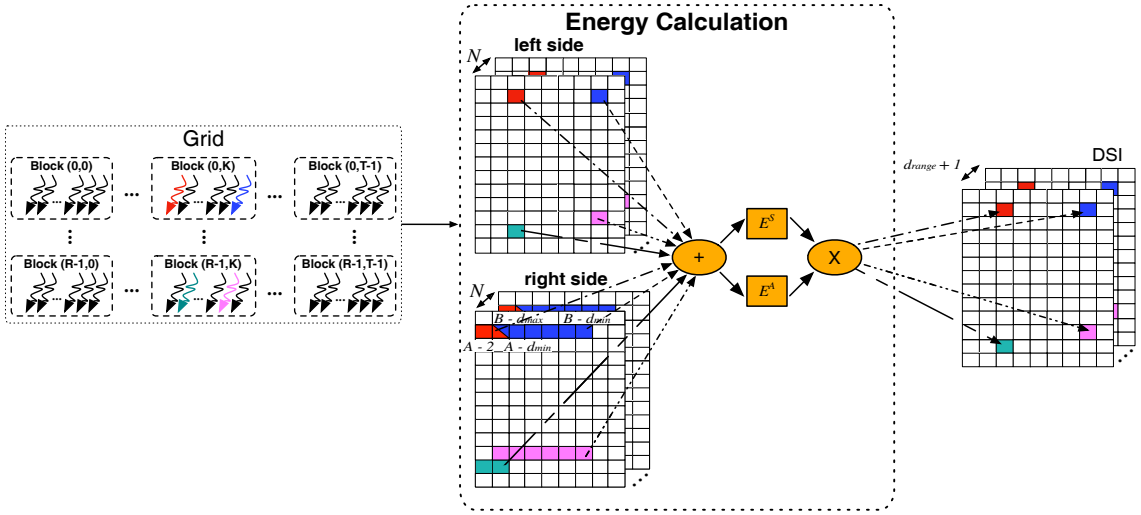


Figure 4.3: Representation of the Energy calculation kernel. (T,R) are the number of blocks in the (x,y) directions, N is the number of wavelets, *dmin* and *dmax* are the minimum and maximum disparities, respectively, and *drange* is the difference between *dmax* and *dmin*.

4.2 Aggregation and refinement parallelization

This section encompasses four algorithms. Unfortunately, the filling algorithm is not parallelizable, which means that it runs on the CPU. The remaining three kernels, namely aggregation, consistency check and 3D conversion, run on the GPU, which implies that the data needs to be transferred from the GPU to the CPU and then back to the GPU. This is shown in Fig. 4.1.

4.2.1 Aggregation on the GPU

Here, each thread calculates the sum of the matching costs over the defined square window, for each disparity, and chooses the disparity associated with the highest sum of costs (Fig. 4.4). Like in the other kernels, each thread is associated to one pixel. This kernel can be computationally costly, depending on the window size and the disparity range chosen. With a large window size, the processing time of the pipeline can increase significantly, damaging processing time performance.

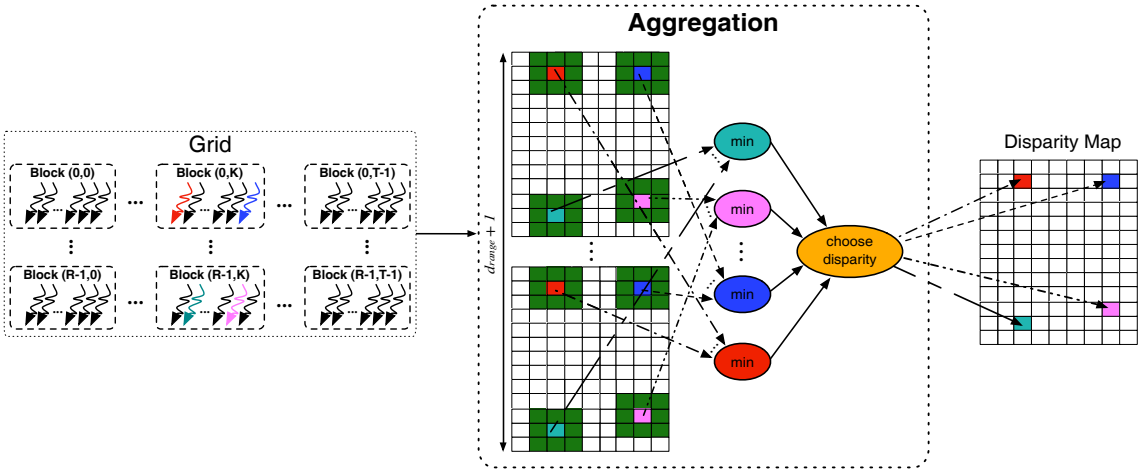


Figure 4.4: Representation of the Aggregation kernel. (T,R) are the number of blocks in the (x,y) directions, and *drange* is the difference between *dmax* and *dmin*.

4.2.2 Refinement on the GPU

The consistency check phase involves two GPUs working in parallel, as two disparity maps are necessary to perform a cross comparison. The maps are calculated concurrently on the two GPUs, saving considerable processing time. Again, each thread verifies the consistency of the disparity associated to a single pixel. This kernel is observable in Fig. 4.5.

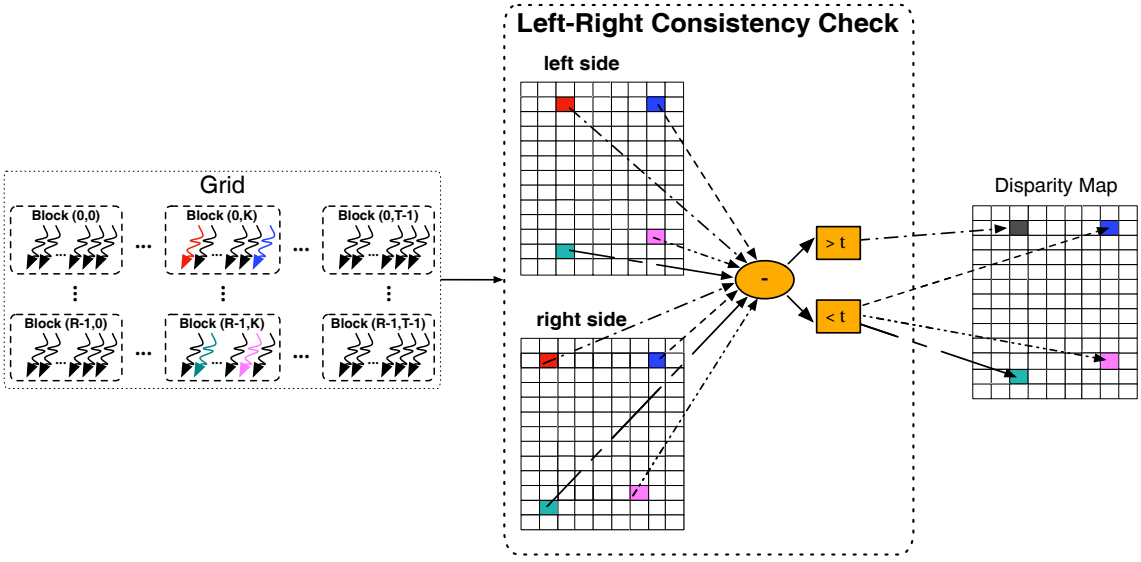


Figure 4.5: Representation of the consistency check kernel. (T,R) are the number of blocks in the (x,y) directions and t is the threshold value.

As mentioned before, the algorithm used to fill the occluded pixels cannot be ported to work on a GPU, hence the data must be transferred to the host's memory in order to be processed by the CPU. When all calculations are over, the data can be brought back to the device's memory.

4.2.3 3D Reconstruction on the GPU

At the end of the pipeline, the final disparity map is used to calculate the 3D coordinates for the 3D view/scene reconstruction. Each thread, associated to one pixel, is responsible for calculating the three coordinates necessary to generate the 3D map, according to (3.9), (3.10) and (3.11). With all pixels processed, data can be transferred from the device to the host.

4.3 Conclusions

This chapter describes the implementation of a fully functional real-time pipeline. The parallelization of LogN is challenging, considering three kernels are needed and various optimization techniques are applied to each kernel.

The use of two GPUs enabled the creation of two disparity maps at the same time, which is of great advantage for the disparity check phase since LogN and aggregation represent the most time consuming stages.

Unfortunately, not all parallelization techniques were possible to apply. Shared memory would limit the cores' occupancy and as a result it shows a negative impact in the final processing times. Asynchronous data transfers cannot be implemented in the application. Nevertheless, other optimization techniques like pinned memory allocations, avoidance of thread divergence, loop unrolling or coalesced memory accesses were used.

The next chapter addresses the studies made, visual results achieved and processing times obtained during this work.

5

Experimental Results

Contents

To present the results, this section has been split into three parts: the first one focuses on the tests performed to refine the disparity maps; the second one discusses the visual results of the 3D scenes generated; and the third one emphasizes the processing times of the program.

To determine the percentage of error in non-occluded areas, the absolute difference between the disparities of corresponding pixels of the calculated disparity map and the matching ground truth image is evaluated. If the difference is superior to one, then the computed disparity is wrong. Also, all tested images are rectified [17].

The pipeline here presented was developed using CUDA 6.5 and uses a GeForce GTX Titan dual-GPU workstation with an i7 4790k @ 4 GHz, 32 GB of RAM running Ubuntu 14.04.1 and GCC 4.8.2. The 3D maps can be visualized with MeshLab v1.3.2.

5.1 Parameter refinement

The objective was to create the best 3D reconstruction for images with slanted surfaces. With that goal in mind, several combinations of the already mentioned pipeline parameters were applied to different sets of images, namely two images from a dataset created for testing purposes (820x1142 pixels); nine images from the Kitty dataset [26] (375x1242 pixels); one image from the synthetic Tunnel dataset [27] (300x400 pixels); another image from the Tsukuba set [3] (288x324 pixels); and, finally, an image from the Oxford Corridor set (256x256 pixels). The Tsukuba image does not contain slant but is used for reference. The values used for the parameters are shown in table 5.1.

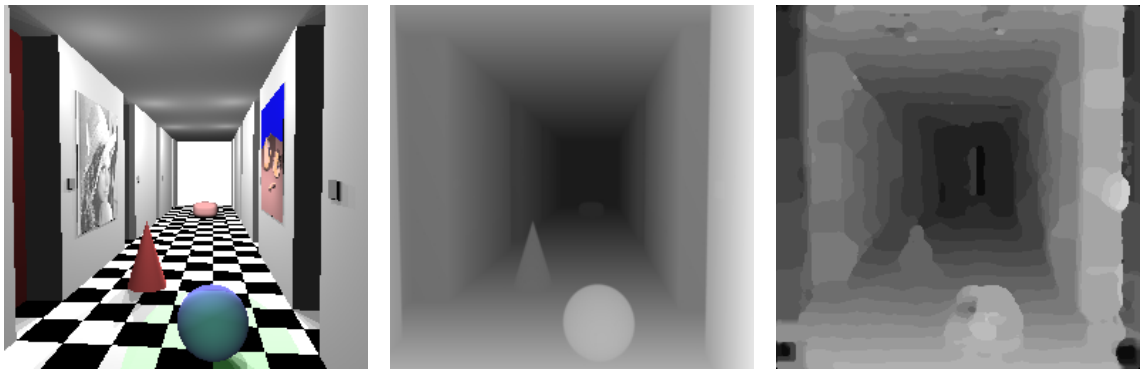
Table 5.1: Parameters values used for the tests.

Parameters				
N	Ω	s	W_0	A_w
15	0.35	1.02	0.2	3
20	0.55	1.05	0.25	9
30	0.75	1.08	0.3	15

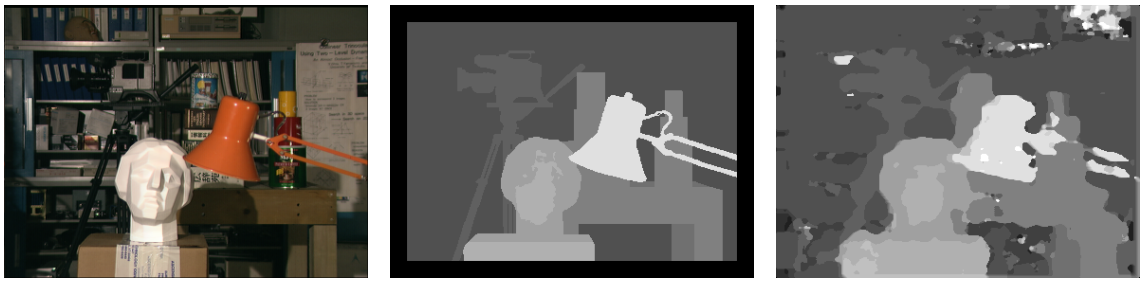
The tests performed in [4] were carried out using specific values for each parameter. That combination is used in this thesis and is called the reference combination, where N=20, Ω=0.55, s=1.05, W_0=0.25 and A_w=9. With these values, the objective is to reach an equilibrium by not compromising neither textured or textureless regions. In fact, that study has been made before [4]. In figure 5.1 the Oxford Corridor, Tsukuba and Tunnel disparity maps calculated with the reference combination can be seen, using the proposed parallel processing pipeline.

With the tests performed, the aim is to show the impact of the various parameters on

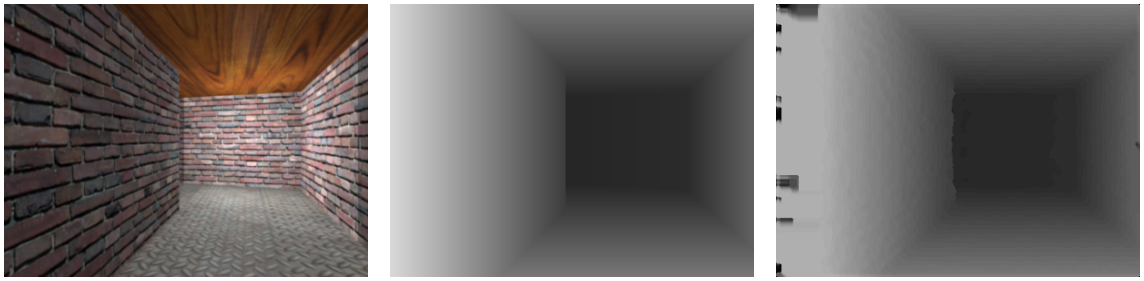
the disparity maps and to find the best combination for images with slanted surfaces in urban scenes. By observing Fig. 5.4, the changes that occur if only one of the parameters is modified are notorious. Increasing the number of wavelets will allow the filter to have a larger range, reacting better to images with textured and textureless regions. The shape factor changes the value of the center frequency of each wavelet, originating different results. The higher the scaling step, the bigger the difference between the center frequencies of each wavelet. In this case, it favours textureless regions. Choosing the right center frequency for the first wavelet is crucial, as it can change the results drastically. In Figures 5.4i) and 5.4j) is visible that with a high first frequency, the disparity map yields better results for textured regions. Finally, a large aggregation window corrects some wrong disparities but affects the definition on the discontinuities.



(a) Oxford Corridor set



(b) Tsukuba set



(c) Tunnel set

Figure 5.1: Oxford Corridor, Tsukuba and Tunnel datasets, ground truth and disparity maps using SymStereo.

By processing disparity maps (trying different combinations of parameters that can be found in table 5.2) for three images and comparing them with the ground truth provided, the accuracy of the pipeline was tested. The results are shown in Fig. 5.2, with the Tunnel image being the most accurate. Combinations two, three and four provide the best results possible for the Oxford Corridor, Tsukuba and Tunnel images, respectively.

Table 5.2: Combinations used in the tests with ground truth verification for the Oxford Corridor, Tsukuba, Tunnel (1, 2, 3, 4) and KITTI (5, 6, ..., 15) datasets.

	Combination														
Parameters	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
N	20	30	15	15	20	30	30	30	20	30	15	30	30	30	20
Ω	0.55	0.35	0.55	0.35	0.55	0.55	0.55	0.35	0.55	0.55	0.35	0.55	0.35	0.35	0.35
s	1.05	1.08	1.05	1.02	1.08	1.05	1.08	1.08	1.08	1.08	1.08	1.05	1.05	1.05	1.08
W_o	0.25	0.2	0.2	0.3	0.2	0.2	0.3	0.25	0.25	0.2	0.2	0.25	0.2	0.25	0.25
A_w	9	15	15	9	15	15	15	15	15	15	15	15	15	15	15

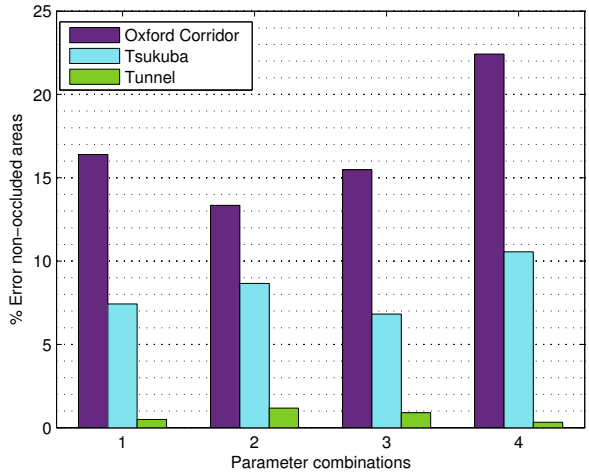


Figure 5.2: Test results for the Oxford Corridor, Tsukuba and Tunnel images with ground truth verification.

The proposed parallel processing pipeline, in particular the *logN* variant of SymStereo has shown good results, especially in images with slanted surfaces. This is verifiable in Figures 5.4 through 5.14. Every disparity map computed with the reference combination, despite having some miscalculated disparities, show a good reconstruction quality. In order to discover the combination that would produce the best 3D map, the tests taken encompassed the creation of several disparity maps of four training images from the KITTI dataset by combining all parameters. Fig. 5.3 shows the pipeline accuracy for images 'A' and 'B' (the only ones presented in this thesis with ground truth, images 'C' to 'K' don't have a ground truth image) of the KITTI dataset. The pipeline produces good results, with a low error percentage. The combinations used were the best performing combinations of the tests taken with the KITTI training dataset and can be found in table 5.2.

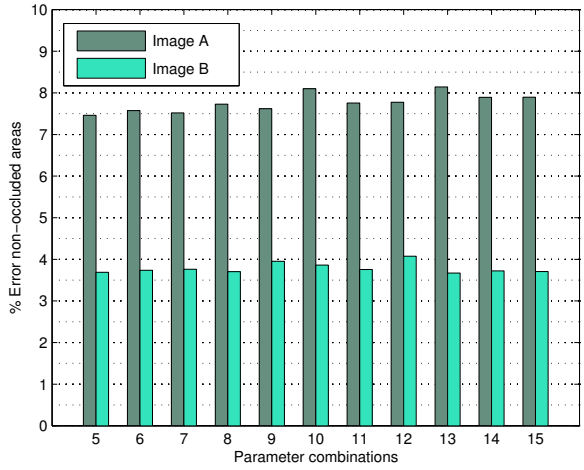


Figure 5.3: Test results for images 'A' and 'B' of the KITTI dataset with ground truth verification.

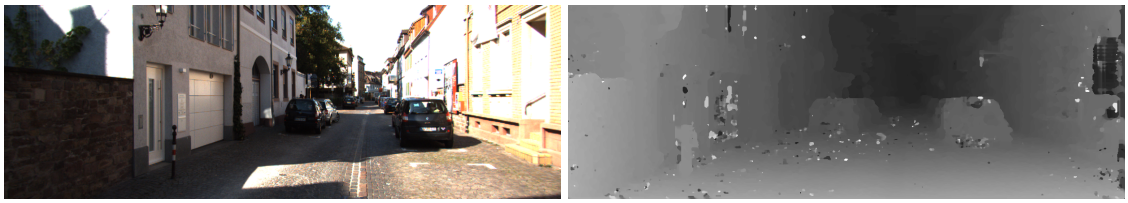
5.2 3D Reconstruction of urban scenes with slant

By observing the final disparity maps, it is verifiable that they are very smooth. Because the tested images are slant dominated, the parameters were manipulated in a way that benefits textureless zones and neglects texture dominated zones. The calculated 3D reconstructions can be observed in different views (from the front, side and top views) in Fig. 5.15 through 5.18. The close analysis of the reconstructions shows that around discontinuities there are some wrongly calculated pixels. For example, in Fig.5.15h), the trunk of the tree surrounded by wall of one of the houses in the background is an error, as the trunk and the wall were calculated as being in the same disparity zone.

When evaluating reconstructions, an important feature presented is the sky. It is one of the main noise sources existent in most test images and, since it belongs to the infinite plane, a correct disparity estimation is impossible. Sky perturbation is especially visible in Fig. 5.18b) and 5.18d).

An additional challenge for stereo algorithms is surface reflection. Despite not being too far apart, the difference between the two cameras is sufficient to take two photos that greatly differ in its pixel color intensities for the same surface. In urban scenes this is very common, as windows reflect the light and most of the cars have metallic painting that has a high reflection coefficient. Fig.5.17b) shows this problem in the car on the left, despite not being too flagrant, as the parameter refinement almost eliminated this challenge.

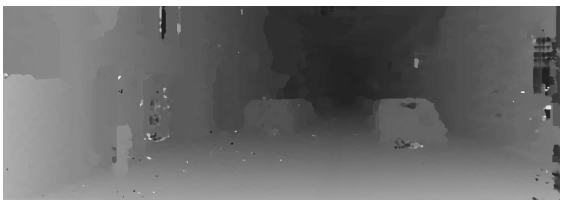
For slanted surfaces, the reconstruction is almost flawless. This is observable in every side view of the aforementioned scenes. By using the calculated parameter configuration, the wrong disparities that existed in slanted zones disappeared in most cases, being re-



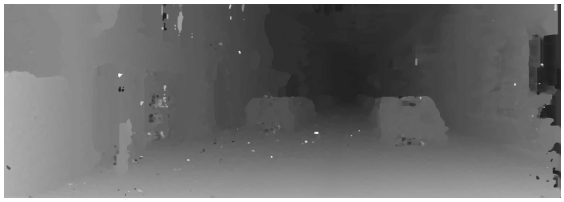
(a) Original Image 'A'

(b) N=20, Ω=0.55, s=1.05, W_0=0.25 and A_w=9

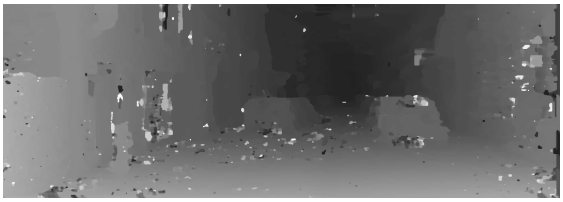
(c) N=15



(d) N=30

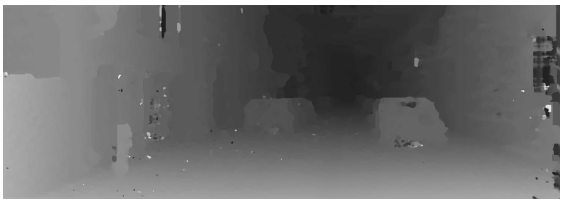


(e) Ω=0.35

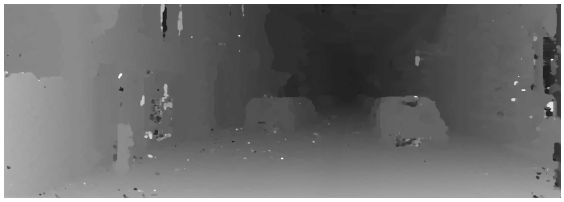


(f) Ω=0.75

(g) s=1.02

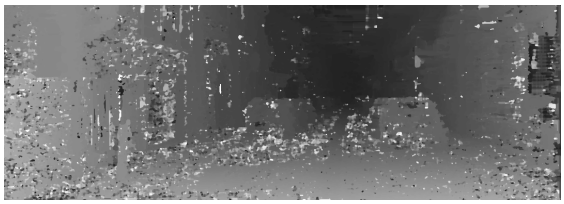


(h) s=1.08

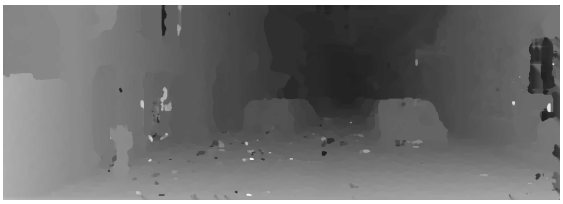


(i) W_0=0.2

(j) W_0=0.3



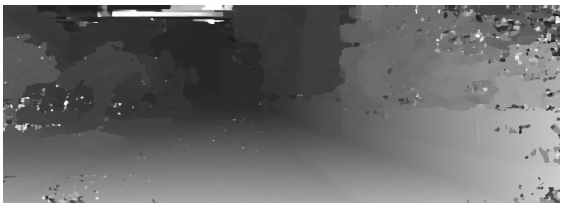
(k) A_w=3



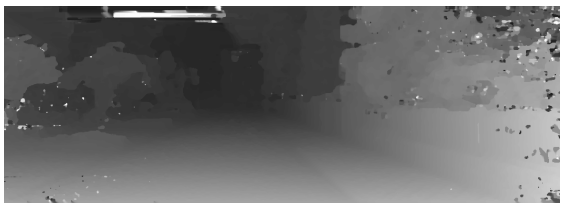
(l) A_w=15

Figure 5.4: Test results for the KITTI dataset image 'A'. In each test, only one parameter was changed, while the others maintained their reference values.

(a) Original Image 'B'



(b) N=20, Ω=0.55, s=1.05, W_0=0.25 and A_w=9



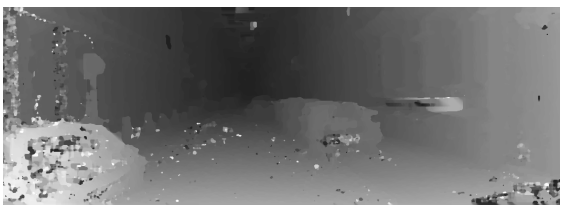
(c) N=20, Ω=0.55, s=1.05, W_0=0.2 and A_w=9

(d) N=20, Ω=0.55, s=1.08, W_0=0.2 and A_w=9

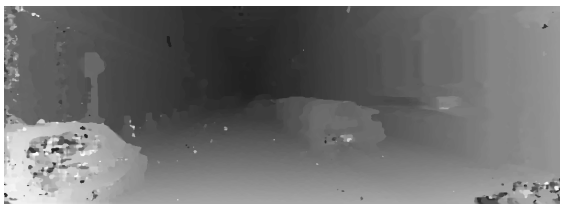
(e) N=20, Ω=0.55, s=1.08, W_0=0.2 and A_w=15

Figure 5.5: Test results for the KITTI dataset image 'B'.

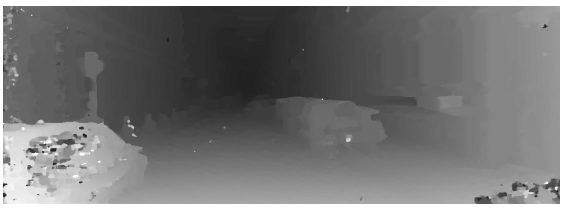
(a) Original Image 'C'



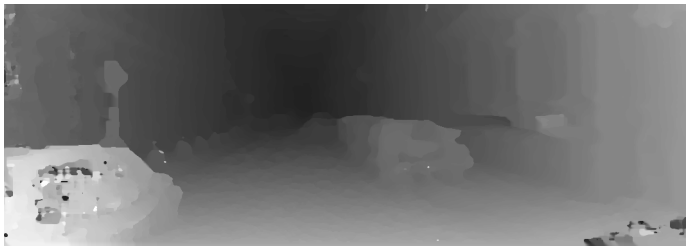
(b) N=20, Ω=0.55, s=1.05, W_0=0.25 and A_w=9



(c) N=20, Ω=0.55, s=1.08, W_0=0.25 and A_w=9



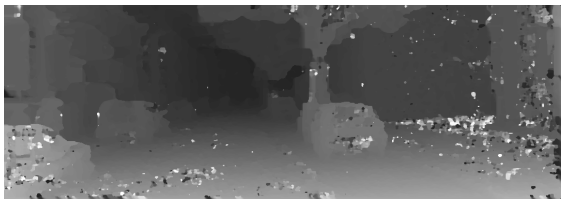
(d) N=20, Ω=0.55, s=1.08, W_0=0.2 and A_w=9



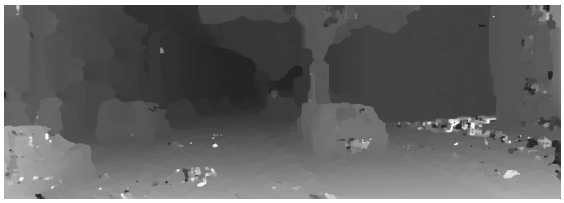
(e) N=20, Ω=0.55, s=1.08, W_0=0.2 and A_w=15

Figure 5.6: Test results for the KITTI dataset image 'C'.

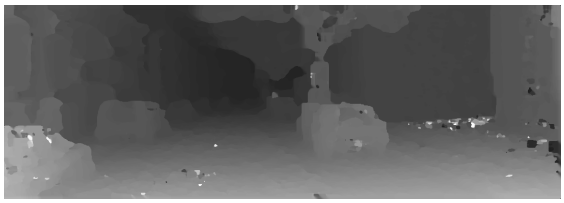
(a) Original Image 'D'



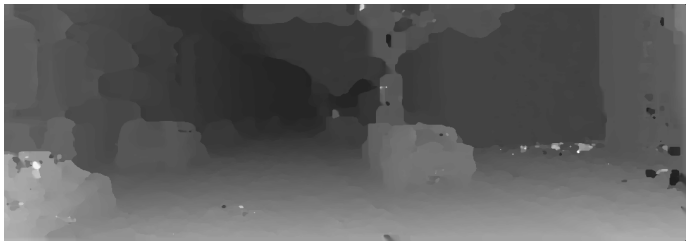
(b) N=20, Ω=0.55, s=1.05, W_0=0.25 and A_w=9



(c) N=20, Ω=0.55, s=1.05, W_0=0.25 and A_w=15



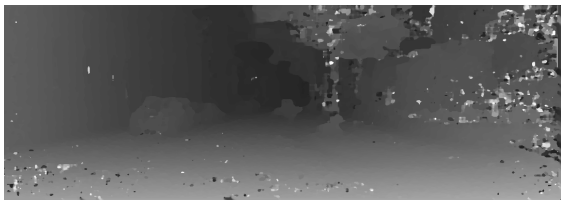
(d) N=20, Ω=0.55, s=1.08, W_0=0.25 and A_w=15



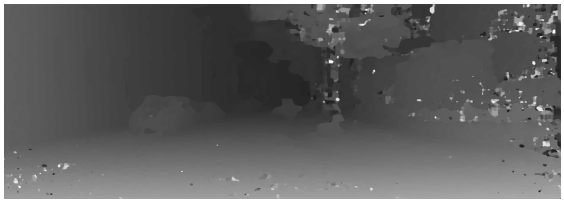
(e) N=20, Ω=0.55, s=1.08, W_0=0.2 and A_w=15

Figure 5.7: Test results for the KITTI dataset image 'D'.

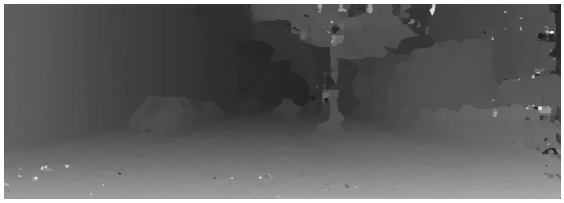
(a) Original Image 'E'



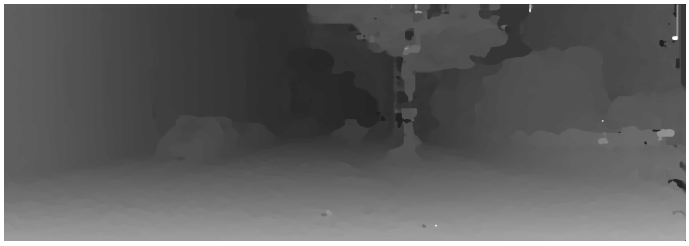
(b) N=20, Ω=0.55, s=1.05, W_0=0.25 and A_w=9



(c) N=20, Ω=0.55, s=1.05, W_0=0.2 and A_w=9



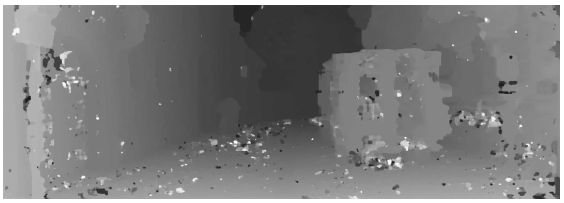
(d) N=20, Ω=0.55, s=1.05, W_0=0.2 and A_w=15



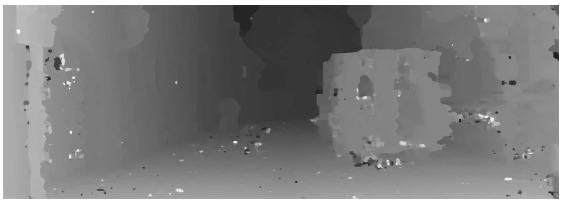
(e) N=20, Ω=0.55, s=1.08, W_0=0.2 and A_w=15

Figure 5.8: Test results for the KITTI dataset image 'E'.

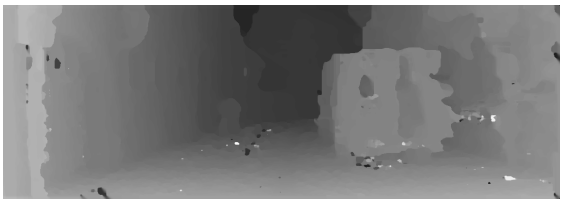
(a) Original Image 'F'



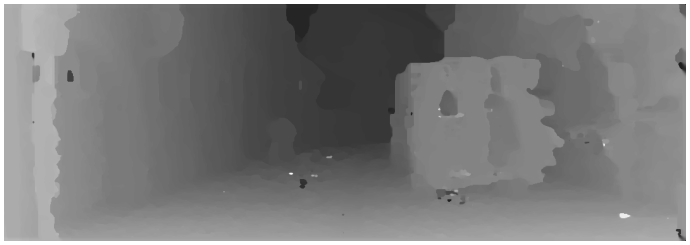
(b) N=20, Ω=0.55, s=1.05, W_0=0.25 and A_w=9



(c) N=20, Ω=0.55, s=1.08, W_0=0.25 and A_w=9



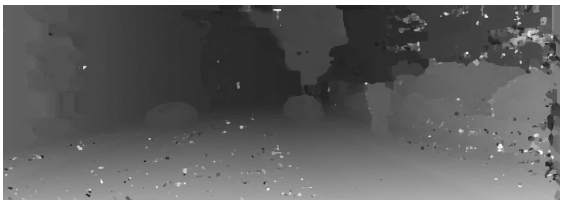
(d) N=20, Ω=0.55, s=1.08, W_0=0.25 and A_w=15



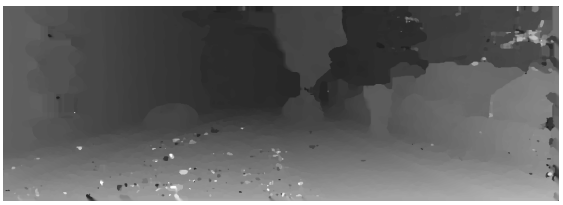
(e) N=20, Ω=0.55, s=1.08, W_0=0.2 and A_w=15

Figure 5.9: Test results for the KITTI dataset image 'F'.

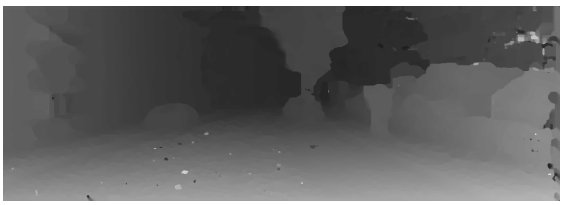
(a) Original Image 'G'



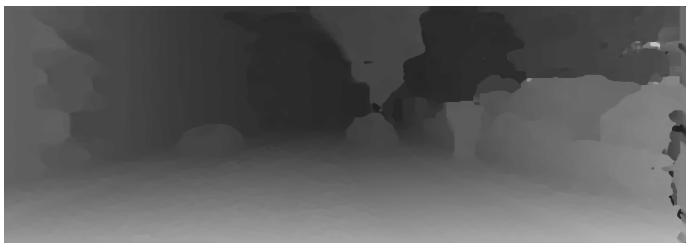
(b) N=20, Ω=0.55, s=1.05, W_0=0.25 and A_w=9



(c) N=20, Ω=0.55, s=1.05, W_0=0.25 and A_w=15



(d) N=20, Ω=0.55, s=1.05, W_0=0.2 and A_w=15



(e) N=20, Ω=0.55, s=1.08, W_0=0.2 and A_w=15

Figure 5.10: Test results for the KITTI dataset image 'G'.

(a) Original Image 'H'

(b) N=20, Ω=0.55, s=1.05, W_0=0.25 and A_w=9

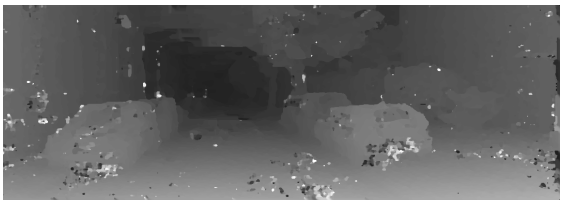
(c) N=20, Ω=0.55, s=1.08, W_0=0.25 and A_w=9

(d) N=20, Ω=0.55, s=1.08, W_0=0.2 and A_w=9

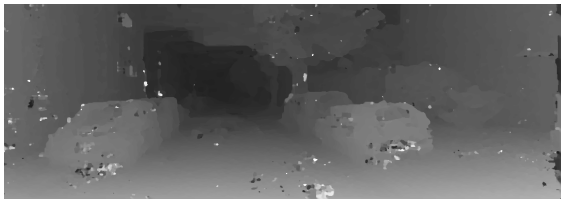
(e) N=20, Ω=0.55, s=1.08, W_0=0.2 and A_w=15

Figure 5.11: Test results for the KITTI dataset image 'H'.

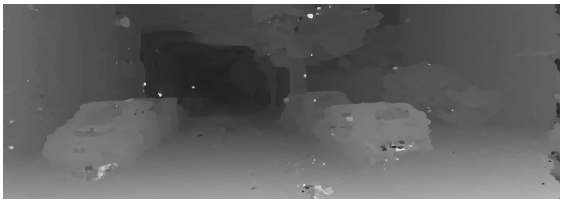
(a) Original Image 'I'



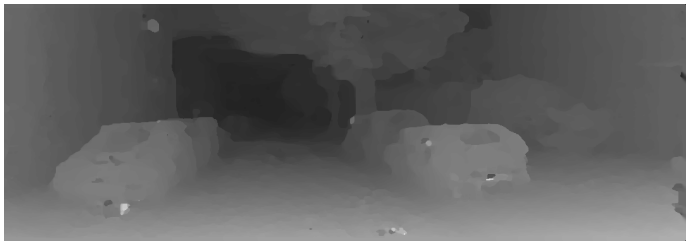
(b) N=20, Ω=0.55, s=1.05, W_0=0.25 and A_w=9



(c) N=20, Ω=0.55, s=1.05, W_0=0.2 and A_w=9



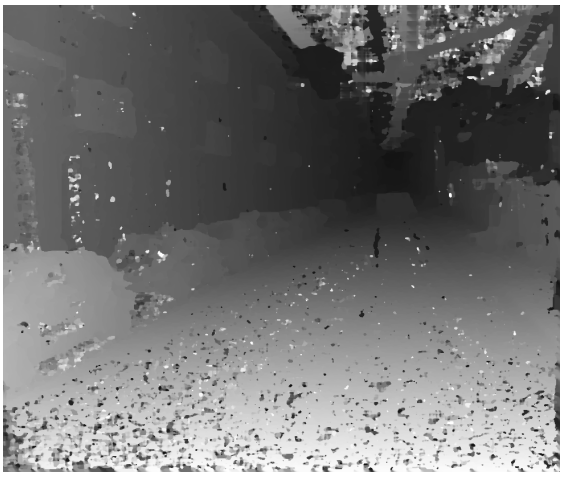
(d) N=20, Ω=0.55, s=1.08, W_0=0.2 and A_w=9



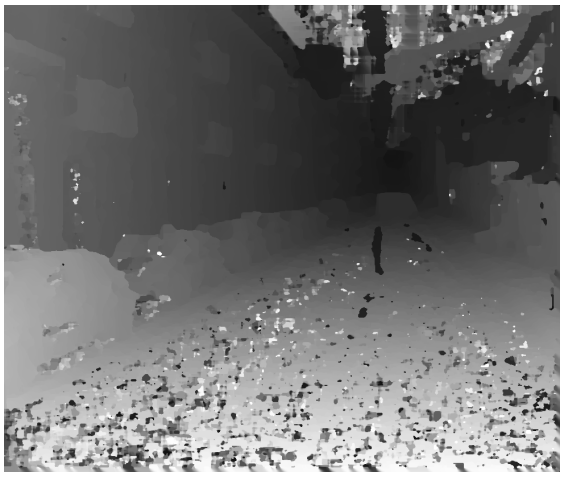
(e) N=20, Ω=0.55, s=1.08, W_0=0.2 and A_w=15

Figure 5.12: Test results for the KITTI dataset image 'I.

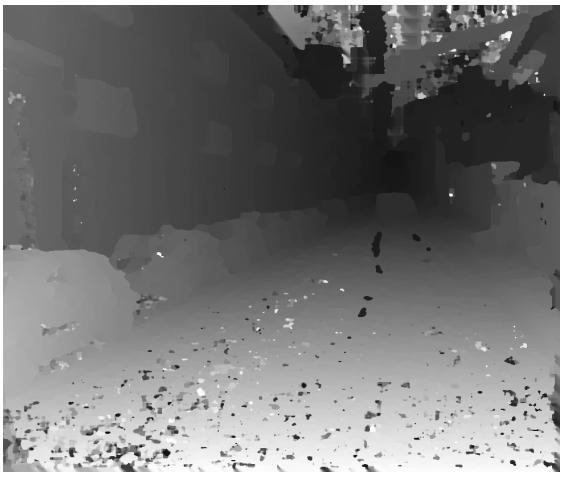
(a) Original Image 'J'



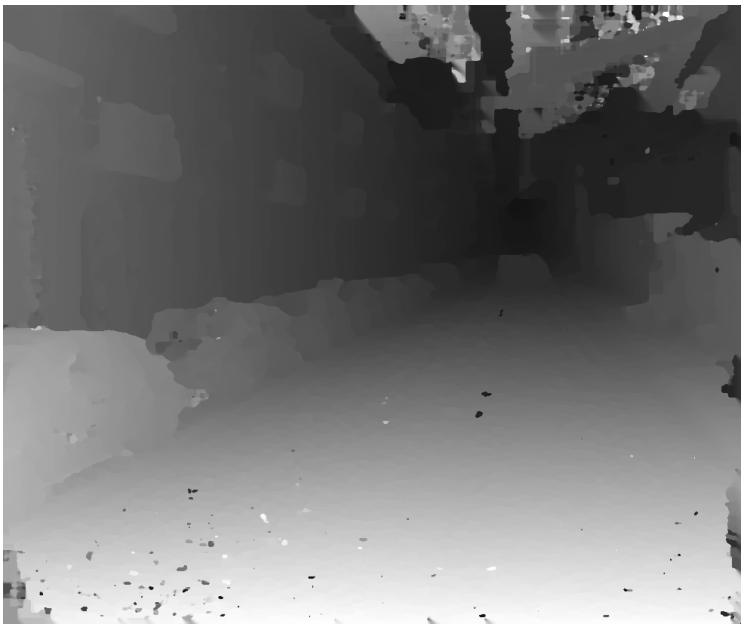
(b) N=20, Ω=0.55, s=1.05, W_0=0.25 and A_w=9



(c) N=20, Ω=0.55, s=1.05, W_0=0.25 and A_w=15



(d) N=20, Ω=0.55, s=1.05, W_0=0.2 and A_w=15



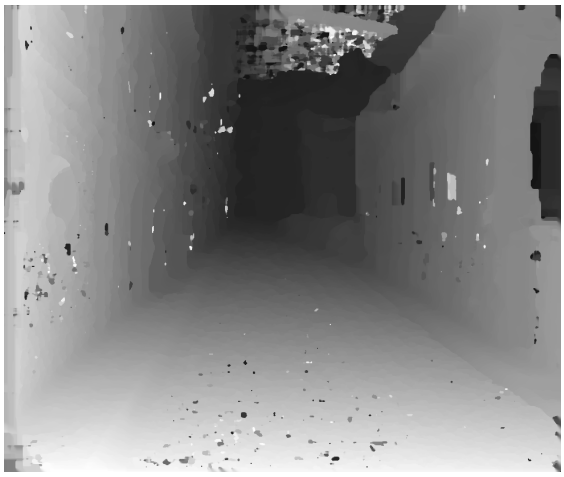
(e) N=20, Ω=0.55, s=1.08, W_0=0.2 and A_w=15

Figure 5.13: Test results for the created dataset image 'J'.

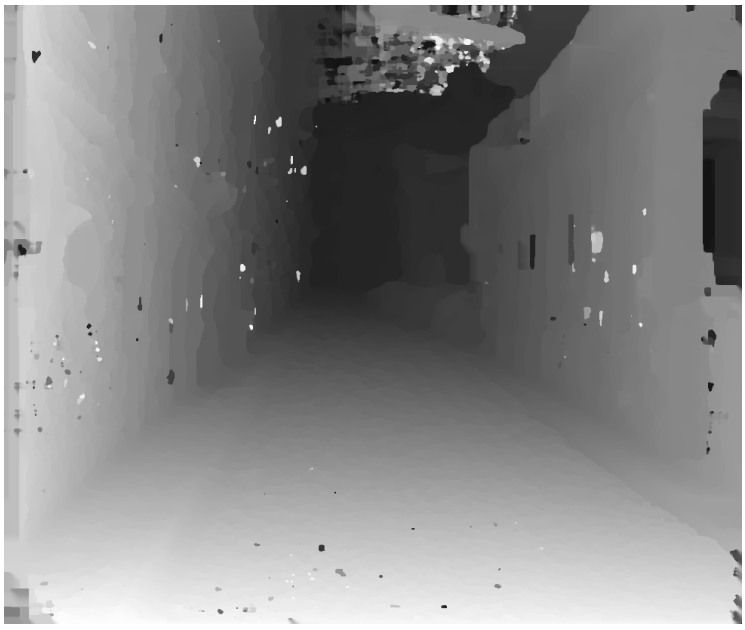
(a) Original Image 'K'

(b) N=20, Ω=0.55, s=1.05, W_0=0.25 and A_w=9

(c) N=20, Ω=0.55, s=1.05, W_0=0.25 and A_w=15



(d) N=20, Ω=0.55, s=1.08, W_0=0.25 and A_w=15



(e) N=20, Ω=0.55, s=1.08, W_0=0.2 and A_w=15

Figure 5.14: Test results for the created dataset image 'K'.

placed by the correct ones. The strength of the SymStereo matching cost is visible here, as it can provide great versatility for different situations.

Videos of the 3D reconstructions can be seen at `http://montecristo.co.it.pt/3DVideos/`.

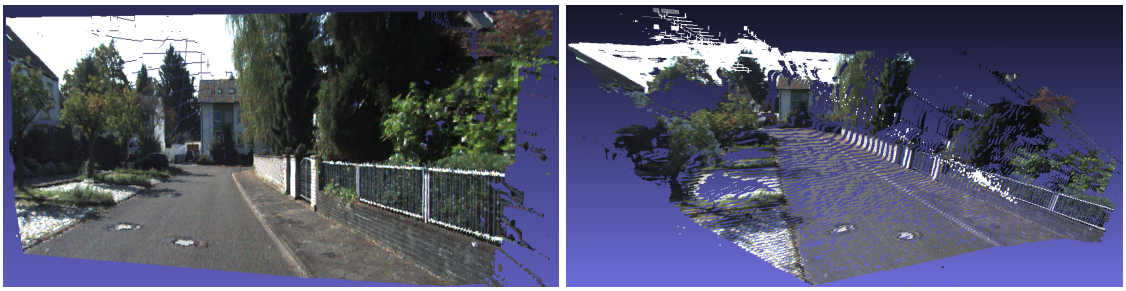
5.3 Numerical results

Results' evaluation may seem to indicate that the parameters are free to change without consequence regarding computational effort. Unfortunately, this is not true as some parameters have a high impact on the processing time of the pipeline. Of the five parameters manipulated, N and A_w have visual and processing impact while Ω, s and W_0 only have visual impact. Another value has a high impact, especially in processing time: the disparity range. Each one of the three parameters has a different impact in processing speed. By increasing the value of N, a larger log-Gabor coefficients matrix is created. This influences the transfer times of the matrix coefficients to the GPU, the log-Gabor filtering and the energy processing kernels that run on the GPU. A_w only influences the aggregation stage but the disparity range impacts not only this stage as the posterior energy calculation phase.

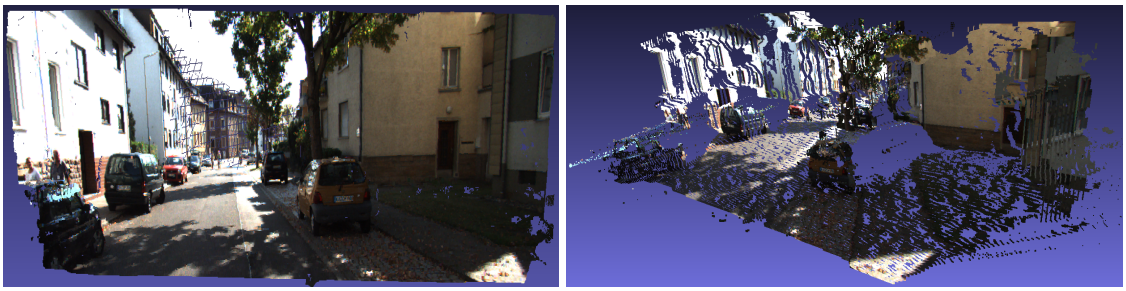
Table 5.3 shows the processing times depending on A_w, N and the disparity range, for five image resolutions. By observing the results, it can be seen that A_w greatly increases the final processing times, especially in larger images, and N is the parameter that least influences the throughput performance. The disparity range is chosen depending on the depth the tester desires to depict. A stereo pair with objects much closer and small depth needs a low disparity range, compared with a pair that has a higher depth. For every image, a range of disparities that fits the scene represented is manually chosen. This way, it is very important to reach a compromise between the parameters, as their variation has a high impact on the number of maps generated per second.

All the tests performed use disparity ranges of 22, 16, 65, 110 and 185 for the Oxford Corridor, Tsukuba, Tunnel, KITTI and the created dataset images, respectively. By analyzing table 5.3, it can be concluded that the pipeline achieves, for each image, 132.3, 120.8, 37.6, 6.1 and 2 processed volumes per second, respectively, for the reference combination. For the best quality results in each image, the effects of changing the parameters are notorious. The rates of generated volumes diminished (excluding the Tunnel image) being 80.4, 93.3, 43.5, 4 and 1.25 for each of the aforementioned image resolutions.

By comparing the pipeline with its serial counterpart, presented in table 5.4, a massive speedup is observed in the achieved processing times, as each image resolution registers a boost of 173x, 217x, 252x, 291x and 307x, respectively.



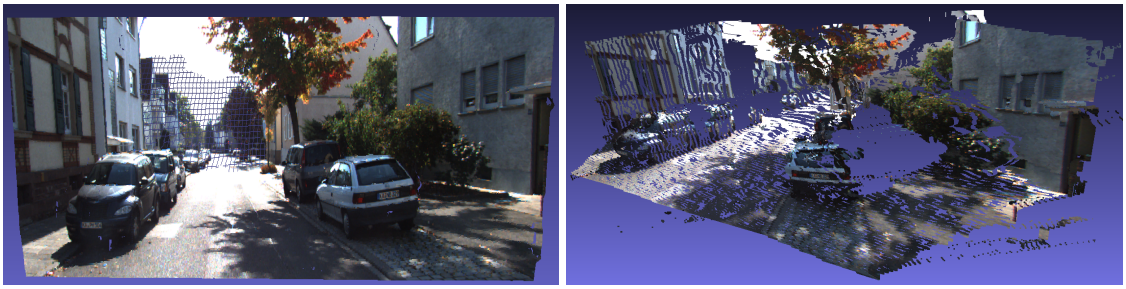
(a) Front view 3D Reconstruction (b) Side view 3D Reconstruction



(c) Front view 3D Reconstruction (d) Side view 3D Reconstruction

(e) Front view 3D Reconstruction (f) Side view 3D Reconstruction

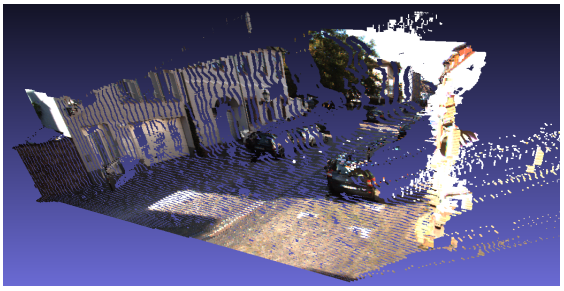
(g) Front view 3D Reconstruction (h) Side view 3D Reconstruction



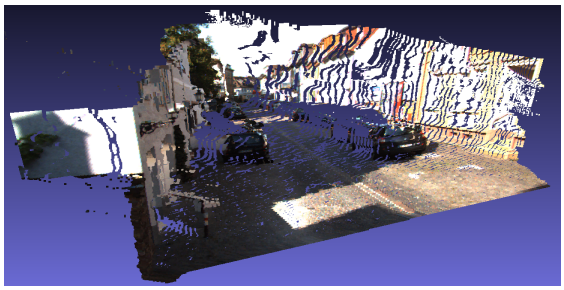
(i) Front view 3D Reconstruction (j) Side view 3D Reconstruction

Figure 5.15: 3D reconstruction of KITTI dataset images 'B', 'D', 'F', 'H' and 'I' with N=20, Ω=0.55, s=1.08, W_0=0.2 and A_w=15.

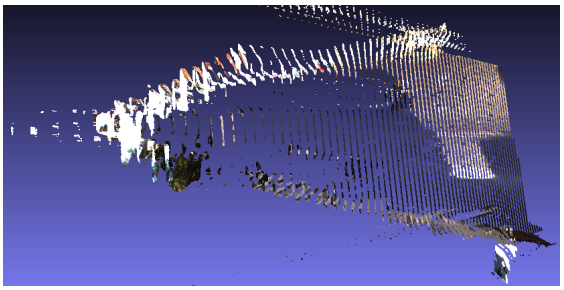
(a) Front view 3D Reconstruction



(b) Left side view 3D Reconstruction

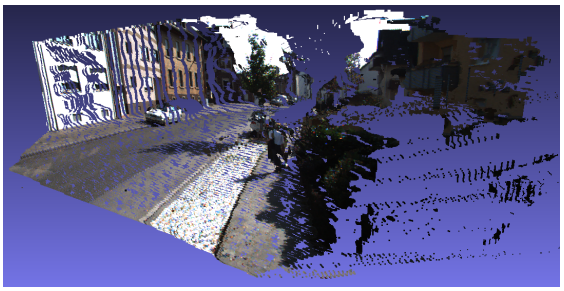


(c) Right side view 3D Reconstruction

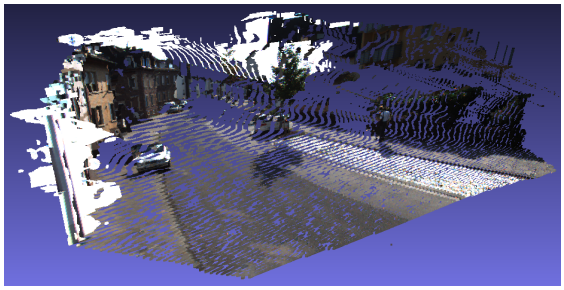


(d) Top view 3D Reconstruction

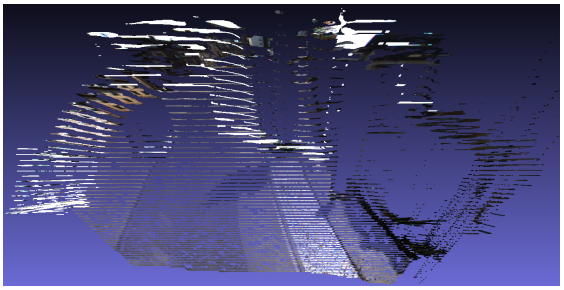
(e) Front view 3D Reconstruction



(f) Left side view 3D Reconstruction



(g) Right side view 3D Reconstruction

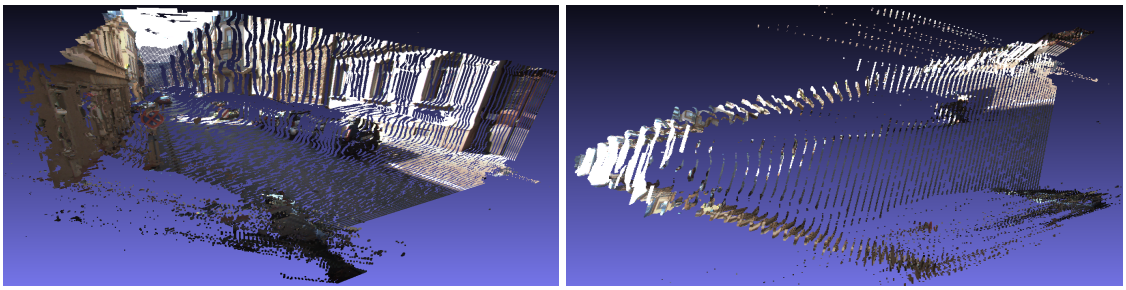


(h) Top view 3D Reconstruction

Figure 5.16: 3D reconstruction of KITTI dataset images 'A' and 'G' with N=20, Ω=0.55, s=1.08, W_0=0.2 and A_w=15.

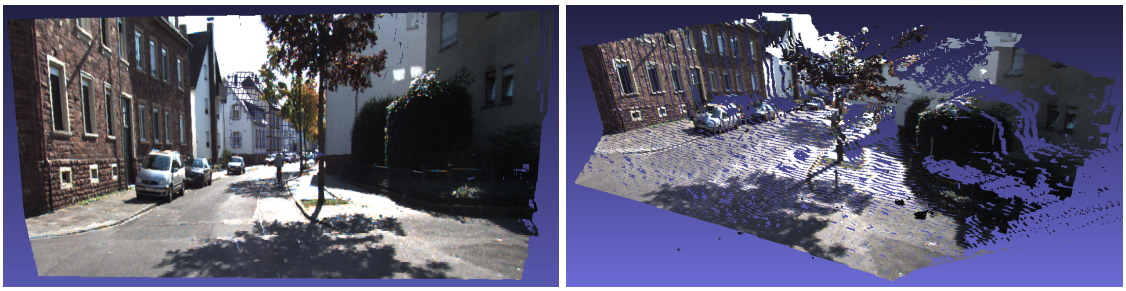
(a) Front view 3D Reconstruction

(b) Left side view 3D Reconstruction



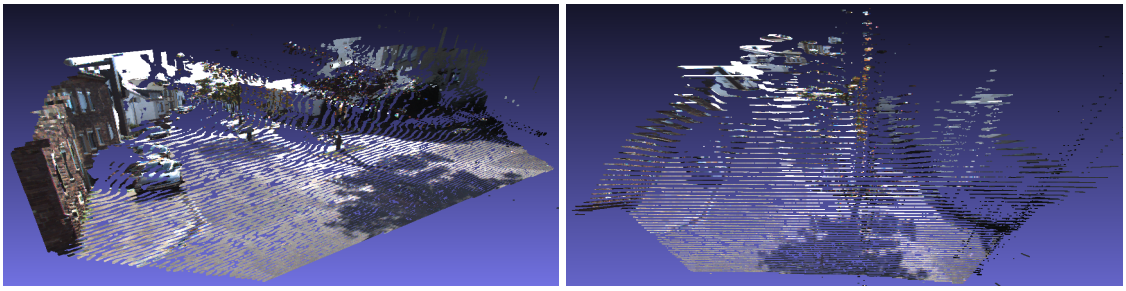
(c) Right side view 3D Reconstruction

(d) Top view 3D Reconstruction



(e) Front view 3D Reconstruction

(f) Left side view 3D Reconstruction

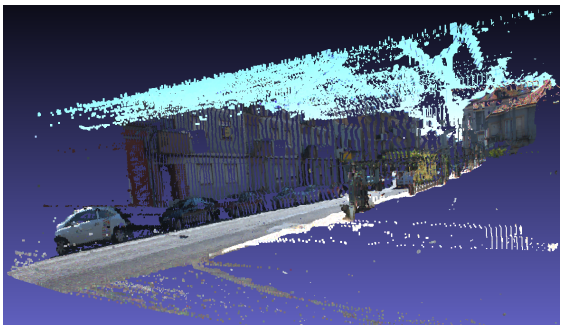


(g) Right side view 3D Reconstruction

(h) Top view 3D Reconstruction

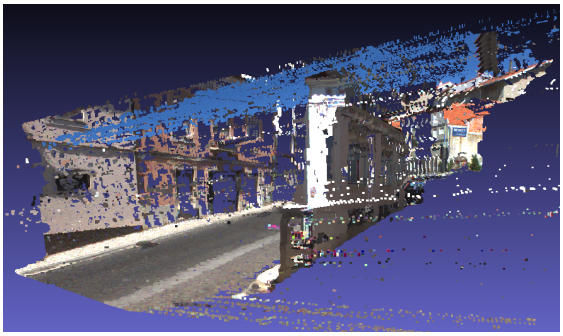
Figure 5.17: 3D reconstruction of KITTI dataset images 'C' and 'E' with N=20, Ω=0.55, s=1.08, W_0=0.2 and A_w=15.

(a) Front view 3D Reconstruction



(b) Side view 3D Reconstruction

(c) Front view 3D Reconstruction



(d) Side view 3D Reconstruction

Figure 5.18: 3D reconstruction of the created dataset images 'J' and 'K' with N=20, Ω=0.55, s=1.08, W_0=0.2 and A_w=15.

Table 5.3: Volumes per second and pipeline times (ms) with the variation of the number of scales, the disparity range and the aggregation window for the five resolutions of the studied images. The values highlighted are the processing times with the reference combination and with the combination for the best visual results obtained. The corresponding volumes per second are represented in rows "Volumes p/sec" 1 and "Volumes p/sec" 2, respectively.

		Image Resolution														
		256x256			**288x384**			**300x400**			**375x1242**			**820x1142**		
Aw	**N / Range**	d=12	d=22	d=32	d=6	d=16	d=26	d=45	d=65	d=85	d=70	d=110	d=150	d=155	d=185	d=215
3	15	3.67	4.82	6.02	4.28	5.68	7.40	11.21	14.51	17.82	69.10	95.11	119.52	249.28	283.98	329.95
	20	4.20	5.60	7.20	4.85	6.64	8.82	13.78	18.10	22.35	85.55	118.57	150.64	310.45	355.93	397.66
	30	5.21	7.33	9.66	6.05	8.81	12.14	20.13	26.68	33.17	124.31	175.31	220.10	452.10	525.12	590.81
9	15	4.55	6.44	8.23	5.04	7.22	9.87	17.01	**22.98**	28.50	97.96	139.80	181.49	382.45	415.25	482.29
	20	4.98	**7.56**	9.48	5.78	**8.28**	11.35	19.69	**26.59**	33.04	113.92	**163.03**	209.83	431.61	**500.21**	565.89
	30	6.11	9.04	11.99	6.86	10.68	14.82	26.11	35.10	43.37	153.88	218.03	279.48	575.29	668.68	755.79
15	15	6.46	9.82	13.21	6.53	**10.72**	15.34	28.68	39.4	50.05	162.36	234.27	301.25	617.92	719.40	832.10
	20	7.11	10.75	14.51	7.19	11.70	16.93	31.35	42.99	54.63	176.84	**254.00**	335.76	680.37	**794.88**	909.39
	30	8.07	**12.44**	17.24	8.36	14.14	20.27	37.77	51.72	64.74	214.28	313.61	399.12	822.86	972.74	1099.37
Volumes p/sec 1			132.3			120.8			37.6			6.1			2	
Volumes p/sec 2			80.4			93.3			43.5			4			1.25	

Table 5.4: CPU vs GPU and the speedup for each image resolution, with the reference combination.

Resolution	CPU (s)	GPU (ms)	Speedup
256x256	1.31	7.56	**173**×
288x384	1.80	8.28	**217**×
300x400	6.70	26.59	**252**×
375x1242	47.44	163.03	**291**×
820x1142	153.56	500.21	**307**×

Fig. 5.19 represents the duration of each phase of the pipeline for an 820x1142 image resolution, with the reference combination. As predicted, the SymStereo and aggregation stages are time consuming, corresponding to more than 95% of the processing time. Despite not being so significant, the occlusion filling algorithm running on the CPU also impacts the final time, representing 3.7% of the total time. Regarding memory transfers and the remaining kernels, they have a small impact, corresponding to the remaining 1.3%.

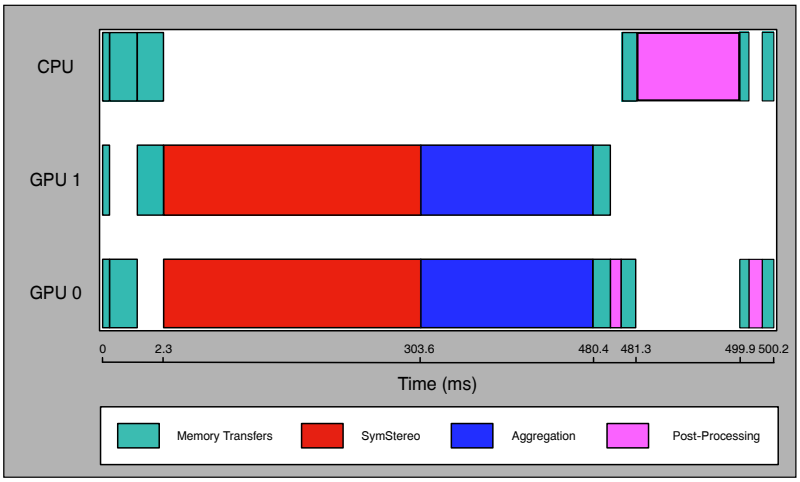


Figure 5.19: Pipeline phase times for an 820x1142 resolution image, with the reference combination.

5.4 Conclusions

With the conducted tests, it was possible to observe the impact of the multiple pipeline parameters on the processing times and visual results. The values chosen can greatly influence both outputs. Therefore, it is important to perform a careful selection in order to reach a balance between the final processing times and the precision of the 3D views achieved.

6

Conclusions

Contents

The real-time SymStereo pipeline created for this thesis is a robust dense stereo estimation application capable of computing two 3D maps per second, for high-resolution images, and 132 volumes per second, for low resolution images. With the tests presented, it was shown how the pipeline reacts to the alteration of its parameters and how that affects processing times and the final 3D maps.

As stated previously in the literature [4], the logN algorithm performs very well for most kind of challenges, especially when dealing with slanted surfaces. This thesis explores the mentioned characteristics a bit further and concludes that for a specific combination of parameters, slanted surfaces can be fully reconstructed with a high estimation precision. Unfortunately, this causes textured zones to be less precise, especially around the discontinuities.

Besides impacting visual results, changing parameters also impacts processing times. Three parameters are highly influential: the number of wavelets, the disparity range and the aggregation window size. It is highly important to reach a compromise between these three values in order to obtain good visual results and real-time processing power.

6.1 Future Work

Despite the reported good performance, the proposed pipeline still has room to evolve, especially in the aggregation and post processing stages. Several window-based algorithms can be applied in order to analyze more deeply the output results in the aggregation section of the pipeline. Also, a new occlusion filling algorithm is recommended, one that is parallelizable since memory transfers between the device and the host's memory must be avoided whenever possible. Regarding the noise originated by the sky, there are methods involving cross correlation between the left and right images that detect pixels belonging to the sky. This is of interest for this work as it allows the exclusion of the mentioned pixels from the final 3D reconstruction.

Bibliography

[1] Z. Zhang and Y. Shan, "A Progressive Scheme for Stereo Matching," in *3D Structure from Images — SMILE 2000*, ser. "Lecture Notes in Computer Science". Springer Berlin Heidelberg, 2001, vol. 2018, pp. 68–85.

[2] S. Seitz, B. Curless, J. Diebel, D. Scharstein, and R. Szeliski, "A Comparison and Evaluation of Multi-View Stereo Reconstruction Algorithms," in *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*, vol. 1, June 2006, pp. 519–528.

[3] D. Scharstein and R. Szeliski, "A Taxonomy and Evaluation of Dense Two-Frame Stereo Correspondence Algorithms," in *International Journal of Computer Vision*, vol. 47, no. 1-3, pp. 7–42, 2002.

[4] M. Antunes and J. P. Barreto, "SymStereo: Stereo Matching using Induced Symmetry," in *International Journal of Computer Vision*, pp. 1–22, 2014.

[5] S. Birchfield and C. Tomasi, "A Pixel Dissimilarity Measure That Is Insensitive to Image Sampling," in *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, pp. 401–406, 1998.

[6] R. Zabih and J. Woodfill, "Non-Parametric Local Transforms for Computing Visual Correspondence," in *Computer Vision — ECCV '94*, ser. Lecture Notes in Computer Science, J.-O. Eklundh, Ed. Springer Berlin Heidelberg, 1994, vol. 801, pp. 151–158.

[7] (2014) CUDA Zone. [Online]. Available: https://developer.nvidia.com/cuda-zone

[8] (2014) OpenCL Zone. [Online]. Available: https://developer.amd.com/tools-and-sdks/opencl-zone/

[9] G. Falcao, V. Silva, L. Sousa, and J. Andrade, "Portable LDPC Decoding on Multicores using OpenCL [Applications Corner]," *Signal Processing Magazine, IEEE*, vol. 29, no. 4, pp. 81–109, July 2012.

[10] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[11] G. E. Moore, "Readings In Computer Architecture," M. D. Hill, N. P. Jouppi, and G. S. Sohi, Eds. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000, ch. "Cramming more components onto integrated circuits", pp. 56–59.

[12] [Online]. Available: http://techreport.com/r.x/core-i7-4770k/haswell-die.jpg

[13] NVIDIA, "Whitepaper - NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110," 2012.

[14] R. Szeliski, *Computer Vision: Algorithms and Applications*, ser. "Texts in Computer Science". Springer, 2010.

[15] R. Szeliski and D. Scharstein, "Sampling the Disparity Space Image," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 26, no. 3, pp. 419–425, March 2004.

[16] R. T. Collins, "A Space-Sweep Approach to True Multi-Image Matching," in *Proceedings of the 1996 Conference on Computer Vision and Pattern Recognition (CVPR '96)*, ser. CVPR '96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 358–.

[17] C. T. Loop and Z. Zhang, "Computing rectifying homographies for stereo vision." in *Conference on Computer Vision and Pattern Recognition (CVPR)*, vol. 1. IEEE Computer Society, June 1999, pp. 125–131.

[18] P. Kovesi, "Symmetry and Asymmetry from Local Phase," in *Tenth Australian Joint Conference on Artificial Intelligence*, 1997.

[19] P. Kovesi, "Image Features from Phase Congruency," in *Videre: Journal of Computer Vision Research*, Tech. Rep., 1995.

[20] T. Kanade and M. Okutomi, "A stereo matching algorithm with an adaptive window: theory and experiment," in *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 16, no. 9, pp. 920–932, Sep 1994.

[21] K.-J. Yoon and I. S. Kweon, "Adaptive Support-Weight Approach for Correspondence Search," in *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 28, no. 4, pp. 650–656, April 2006.

[22] J. Sun, N.-N. Zheng, and H.-Y. Shum, "Stereo Matching using Belief Propagation," in *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 25, no. 7, pp. 787–800, July 2003.

[23] Y. Boykov, O. Veksler, and R. Zabih, "Fast Approximate Energy Minimization Via Graph Cuts," in *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 23, no. 11, pp. 1222–1239, Nov 2001.

[24] S. Birchfield and C. Tomasi, "Depth Discontinuities by pixel-to-pixel Stereo," in *Computer Vision, 1998. Sixth International Conference on*, Jan 1998, pp. 1073–1080.

[25] (2014) cuFFT. [Online]. Available: https://developer.nvidia.com/cuFFT

[26] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012.

[27] C. Richardt, D. A. H. Orr, I. P. Davies, A. Criminisi, and N. A. Dodgson, "Real-time Spatiotemporal Stereo Matching using the Dual-Cross-Bilateral Grid," in *European Conference on Computer Vision (ECCV)*. Springer Verlag, 2010.