MASTER IN HIGH PERFORMANCE COMPUTING

Final Year Project

Design and implementation of a HPC solution for Stereo Matching

Student: Luis Omar Álvarez Mures

Advisors: Emilio J. Padrón González

Juan Ramón Rabuñal Dopico

A Coruña, September 15, 2015.

Información xeral

Título do proxecto: **"Deseño e implementación dunha solución HPC para Stereo Matching"**

Clase de proxecto: Proxecto de desenvolvemento en investigación

Nome do estudante: Luis Omar Álvarez Mures

Nome dos directores: Emilio José Padrón González
Juan Ramón Rabuñal Dopico

Membros do tribunal:

Data de lectura:

Cualificación:

Dedicated to Rebeca and my parents.

Acknowledgements[1]

My sincerest thank you for the help provided:

To Mr. Emilio José Padrón González.

To Mr. Juan Ramón Rabuñal Dopico.

[1] In alphabetical order

Resumo:

O obxectivo deste proxecto é o deseño e implementación dunha ferramenta multiplataforma para Stereo Matching. Consideramos como visión estereoscópica ou visión tridimensional (3D) a calquera técnica capaz de recoller información visual tridimensional e crear ilusión de profundidade nunha imaxe. De maneira natural a nosa visión é estereoscópica, puidendo percibir sensación de profundidade, afastamento, etc. Isto se consegue grazas á separación horizontal dos nosos ollos, procesando o noso cerebro as diferenzas entre as imaxes percibidas polo noso sistema visual.

Véñense empregando sistemas artificiais de visión estereoscópica para a obtención de información 3D en diferentes aplicacións dende hai varias décadas. O problema central que abordan estes sistemas é o da determinación da correspondencia entre os píxels que proveñen do mesmo punto dos pares de imaxes da escena tridimensional, ou correspondencia estereoscópica.

Tras unha primeira fase de estudo dos distintos algoritmos existentes, levarase a cabo unha implementación secuencial base empregando a linguaxe de programación C++. Esta linguaxe multiparadigma estende a ampliamente coñecida liguaxe de programación C con mecanismos que permiten tanto a manipulación de obxectos como a capacidade para a programación xenérica. Este compromiso entre eficiencia e versatilidade fai que esta linguaxe sexa ampliamente utilizada en entornos HPC.

Para as versións paralelas do código en CPU utilizarase OpenMP, unha API de programación multi-proceso en plataformas de memoria compartida. En canto á implementación GPU, levarase a cabo cunhas tecnoloxías GPGPU, CUDA e OpenCL, unhas APIs que permiten crear aplicacións con paralelismo a nivel de datos e tarefas que poden executarse en procesadores gráficos.

Abstract:

The objective of this project is designing and implementing a multi-platform HPC Stereo Matching solution. Any technique that is capable of collecting 3D information and creating the illusion of depth in an image, is considered stereoscopic vision or tridimensional (3D) vision. Our vision is stereoscopic by nature, being able to perceive the sensation of depth, distance, etc. This is achieved thanks to the horizontal separation between our eyes, leading to the processing of the differences between the perceived images of the visual system by our brain.

Artificial systems for stereoscopic vision for the obtaining of 3D information in multiple applications, have been employed for several decades. The main problem that these systems tackle; is the determination of the correspondence between the pixels that come from the same point in the image pairs of the tridimensional scene, or Stereo Matching.

After a preliminary phase for the study of the current algorithms for Stereo Matching, a base sequential implementation using C++ will be developed. This multi-paradigm language extends the amply known programming language C with mechanisms that allow the manipulation of objects or the ability for using generic programming. This compromise between efficiency and versatility makes this language vastly used in HPC environments.

For the parallel versions of the code in CPUs we will utilize OpenMP, a multi-process programming API for shared memory platforms. Moreover, for the GPU implementation, GPGPU technologies will be used. CUDA and OpenCL are APIs that allow the creation of applications with data-level and task-level parallelism that can be executed in graphical processors.

Lista de palabras chave:

Correspondencia estéreo, tiempo real, visión estereoscópica, Visión Artificial, GPGPU, GCVL, open source, multi-plataforma.

Keywords:

Stereo Matching, real-time, stereoscopic vision, Computer Vision, GPGPU, GCVL, open source, multi-platform.

Hardware e software empregado:

- Intel Core i7s, NVIDIA GTXs & GTs.
- C++, Visual Studio, Git, OpenCV, OpenMP, OpenCL, CUDA, Boost, GCC.

Contents

1 Introduction **1**

1.1 Motivation and context 2

1.2 Objectives 3

1.3 Structure 3

2 Technological Foundations **5**

2.1 Computer Vision 6

2.1.1 Stereo Vision 8

2.1.2 Stereo Correspondence 8

2.1.3 Block Matching 10

2.2 GPGPU 12

2.2.1 Architecture 13

2.2.2 OpenCL 14

2.2.3 CUDA 15

3 GCVL: GPU Computer Vision Library **17**

3.1 Development Methodology 17

3.1.1 Agile manifesto 17

3.2 Design 19

3.2.1 Class diagram 19

3.2.2 General Tools 20

3.2.3 CPU Module 21
3.2.4 OpenCL Module 21
3.2.5 CUDA Module 22
3.3 Technology 24
3.4 Usage 25
3.4.1 CPU Module 25
3.4.2 OpenCL Module 25
3.4.3 CUDA Module 26

4 Performance & Experimental Results **29**
4.1 Performance Results 29
4.2 Experimental Results 34

5 Conclusions and future lines of work **39**
5.1 Conclusions 39
5.2 Future lines of work 40

List of Figures

1.1 Pose estimation 1

1.2 Stereoscopic vision 2

2.1 Background segmentation 5

2.2 Optical illusion 6

2.3 Example of stereoscopic images employed in scene reconstruction. 8

2.4 Stereo vision 9

2.5 Stereo reconstruction 10

2.6 Stereo vision 12

2.7 CPU Architecture 13

2.8 GPU Architecture 13

2.9 OpenCL Architecture 14

2.10 CUDA Architecture 16

3.1 GCVL class diagram. 20

3.2 General Tools module class diagram. 21

3.3 CPU module class diagram. 22

3.4 OpenCL module class diagram. 23

3.5 CUDA module class diagram. 24

4.1 CPU performance results for the Tsukuba dataset. 30

4.2 CPU performance results for the Bowling dataset. 31

4.3 OpenCL performance results for the Tsukuba dataset. 31
4.4 OpenCL performance results for the Bowling dataset. 32
4.5 CUDA performance results for the Tsukuba dataset. 32
4.6 CUDA performance results for the Bowling dataset. 33
4.7 OpenCL vs. CUDA in the Bowling dataset. 33
4.8 Speedup obtained in the Bowling dataset. 34
4.9 Depth map obtained with a maximum disparity of 16 and an aggregation window of 5 in the Tsukuba dataset. 34
4.10 Depth map obtained with a maximum disparity of 16 and an aggregation window of 9 in the Tsukuba dataset. 35
4.11 Textureless map of the Bowling dataset, pixels marked as white are low-texture regions. 35
4.12 Depth map obtained with a maximum disparity of 100 and an aggregation window of 13 in the Bowling dataset. 36
4.13 Depth map obtained with a maximum disparity of 170 and an aggregation window of 13 in the Bowling dataset. 37

List of Tables

4.1 Datasets used in our tests. 29

Chapter 1

Introduction

Computer Vision is a field that includes techniques for acquiring, processing, analyzing, and understanding images. This scientific area is not only limited to images but also to high-dimensional data from the real world. It aims to produce numerical or symbolic information in order to perform decisions [SS01]. A common trend in this field, has been duplicating the abilities of human vision by electronically perceiving and interpreting the image.

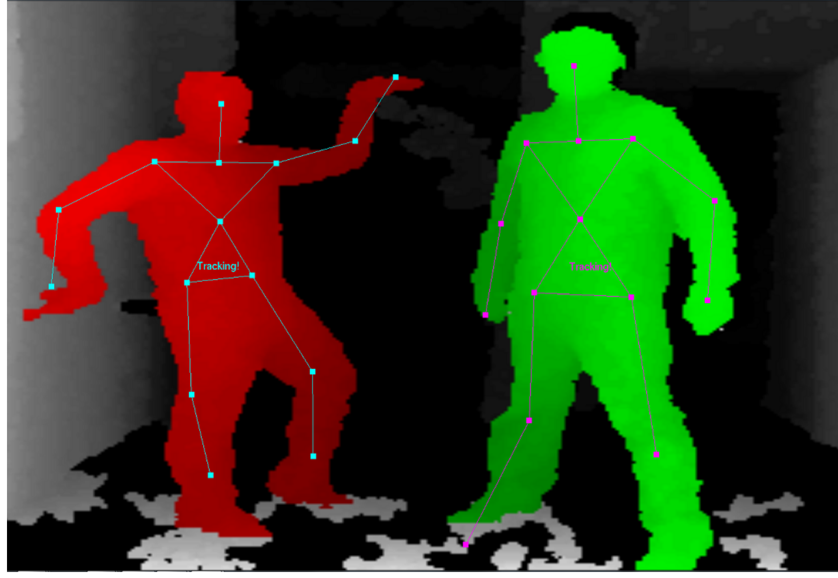


Figure 1.1: The specific task of determining the pose of an object in an image is referred to as pose estimation.

This image understanding can be seen as the extraction of symbolic information from image data using models constructed with the aid of geometry, physics, statistics, and

learning theory [FP03]. As a scientific discipline, Computer Vision considers the theory behind artificial systems that extract information from images. The image data can have many forms, such as video frames, views from multiple cameras, or multi-dimensional data from a medical scanner. As a technological discipline, Computer Vision seeks to apply its theories and models to the construction of artificial vision systems.

Sub-domains of computer vision include scene reconstruction, event detection, video tracking, object recognition, object pose estimation, learning, indexing, motion estimation, and image restoration (see Figure 1.1). Our case study pertains to the field of stereo vision. In this subject, the main objective is the extraction of 3D information from digital images, such as obtained by a CCD camera. By comparing information about a scene from two vantage points, 3D data can be extracted by examination of the relative positions of objects in the two panels (see Figure 1.2).

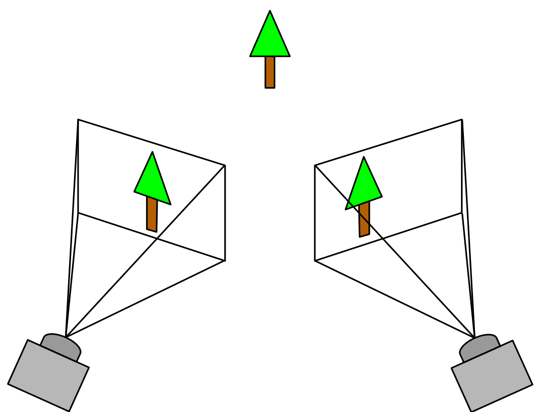


Figure 1.2: Basic diagram of a stereo vision setup.

1.1 Project motivation and context

One of the key steps in stereo vision techniques, is obtaining the stereo correspondence between two images or Stereo Matching. The correspondence problem refers to the process of ascertaining which parts of one image correspond to which parts of another image. This is not a trivial affair, since differences appear due to movement of the camera, the elapse of time, and/or movement of objects in the photos.

Current approaches to obtain this information, involve the usage of a single CPU. Our aim with this project is taking advantage of modern hardware to its fullest capacities. This is specially important nowadays, since camera systems keep on improving leading to images with very high resolutions. This complex task involves exploiting the massive parallel power of current multi-core CPUs, GPUs, etc. In addition, our desire is not only

the implementation of a Stereo Matching algorithm; but the creation of a C++ framework that will allow the user to implement parallel Computer Vision algorithms with ease.

The set of software tools developed in this project has been called **GCVL** (GPU Computer Vision Library). This library provides several means to implement GPU algorithms in an simpler manner using OpenCL or CUDA. In addition, a sample implementation of a Stereo Matching algorithm called Block Matching will also be provided to showcase the features offered in the framework. This algorithm will have a serial, OpenMP, OpenCL and CUDA implementation that employs the designed software tools.

1.2 Project objectives

The main objectives in this project are:

- Studying different Stereo Matching techniques.
- Design, implementation and documentation of tools to ease GPGPU programming.
- Design, implementation and documentation of the chosen algorithm.
- CPU parallelization of the implemented algorithm.
- GPU parallelization of the implemented algorithm.

In this project, a fast Stereo Matching algorithm is required. In addition, the algorithm needs to keep all the precision possible, but be suitable for real-time processing of interactive image sequences. Not only this, but also the algorithm needs to be highly parallel so GPUs can be efficiently used. This algorithm will be implemented with the developed toolset for GPGPU computing. Moreover, evidence will be collected of the scalability and performance of the chosen algorithm and its parallel implementations.

1.3 Document structure

The rest of the document is organized following this structure:

Chapter 2. Technological Foundations

This chapter briefs very basic concepts about Computer Vision needed to understand the rest of the document: How stereo correspondence, stereo matching and our chosen solution work. This chapter will also introduce the reader to basic GPGPU concepts.

Chapter 3. GCVL: GPGPU Computer Vision Library

This chapter describes the structure of the library we propose in this work and presents the class diagram of the software package.

Chapter 4. Performance & Experimental Results

This chapter presents the obtained performance and experimental results, so that the reader can compare our parallel implementations with the sequential one.

Chapter 5. Conclusions and future lines of work

Finally, this chapter presents the main conclusions obtained with this project and outlines possible extensions to GCVL.

Chapter 2

Technological Foundations

In this chapter, some Computer Vision concepts are introduced. Paying special attention to our case study that is stereo correspondence. We first introduce what is computer vision and delve into our case study, second we explain the chosen Stereo Matching algorithm and third describe what GPGPU is and its applications in our area of interest. Only a brief description of these concepts is presented, to get more familiarized with Computer Vision the books [SS01] and [FP03] are recommended. For stereo correspondence techniques, the book chapter [Sze11] is an essential source.

Figure 2.1: The human visual system has no issue interpreting the subtle changes in translucency and shading in this photograph and segmenting the object from its background.

2.1 Computer Vision

As humans, we perceive the tridimensional structure of the world around us with apparent ease. Think of how vivid the tridimensional perception is when one looks at a vase of flowers sitting on the table. You can tell the shape and translucency of each petal through the subtle patterns of light and shading that play across its surface. One can easily segment each flower from the background of the scene (see Figure 2.1).

Looking at a framed group portrait, you can effortlessly for example count all of the people and even guess their emotions from their facial appearance. Perceptual psychologists have spent decades trying to understand how the visual system works and, even though they can find optical illusions to tease apart some of its principles (see Figure 2.2), a solution to the complete puzzle remains elusive.

Figure 2.2: This work by Eric Johansson is composed of optical illusions that use highly creative pieces that render manipulations of perspective to make the viewer see a perplexed set of images.

In areas, such as rendering a still scene composed of everyday objects or animating extinct creatures such as dinosaurs, the illusion of reality is almost perfect. In computer vision, we are trying to do the opposite, i.e., to describe the world that we see in one or a sequence of images and to reconstruct its properties, such as shape, illumination, and color distributions. It is incredible that humans and animals do this so effortlessly, while

computer vision algorithms are so error prone.

Applications range from tasks such as industrial machine vision systems which, say, inspect bottles speeding by on a production line, to research into artificial intelligence and computers or robots that can comprehend the world around them. In many computer vision applications, the computers are preprogrammed to solve a certain task, but methods based on learning are now becoming more common. Examples of applications of computer vision include systems for:

- Process control.
- Navigation.
- Event detection.
- Information organization.
- Object or environment modeling.
- Interaction.
- Automatic inspection.
- Etc.

Each of the application areas described above employ a range of computer vision tasks; more or less well-defined measurement problems or processing workloads, which can be solved using a variety of methods. Some examples of typical computer vision tasks are presented below:

- **Recognition:** the classical problem in computer vision, image processing, and machine vision is that of determining whether or not the image data contains some specific object, feature, or activity.
- **Motion analysis:** several tasks relate to motion estimation where an image sequence is processed to produce an estimate of the velocity either at each points in the image or in the 3D scene, or even of the camera that produces the images.
- **Scene reconstruction:** Given one or (typically) more images of a scene, or a video, scene reconstruction aims at computing a 3D model of the scene.
- **Image restoration:** The aim of image restoration is the removal of noise (sensor noise, motion blur, etc.) from images.

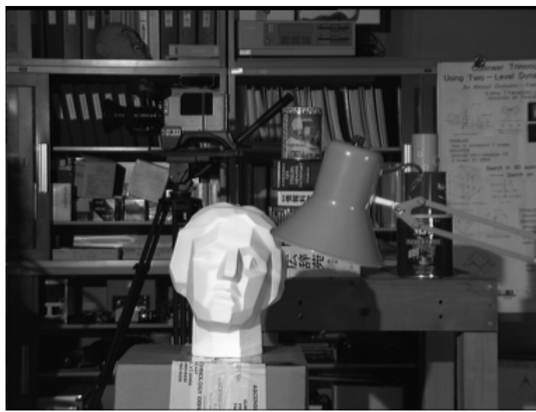
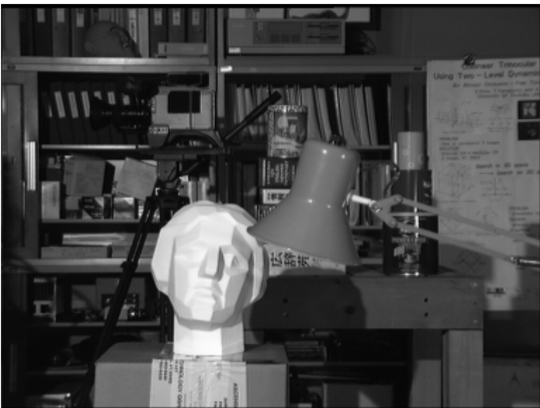



Figure 2.3: Example of stereoscopic images employed in scene reconstruction.

Our main interest in this project is *scene reconstruction* from one or more pairs of images (see Figure 2.3), specifically with stills obtained using stereo vision techniques. This images can be processed to obtain a depth estimation of the scene, once it is known, we have a 3D model of said scene.

2.1.1 Stereo Vision

Any technique that is capable of collecting 3D information and creating the illusion of depth in an image, is considered stereoscopic vision or tridimensional (3D) vision. Our vision is stereoscopic by nature, being able to perceive the sensation of depth, distance, etc. This is achieved thanks to the horizontal separation between our eyes, leading to the processing of the differences between the perceived images of the visual system by our brain.

Computer stereo vision is the extraction of 3D information from digital images, such as obtained by a CCD camera. Artificial systems for stereoscopic vision for the obtaining of 3D information in multiple applications, have been employed for several decades. The main problem that these systems tackle; is the determination of the correspondence between the pixels that come from the same point in the image pairs of the tridimensional scene. By comparing information about a scene from two vantage points, 3D data can be extracted by examination of the relative positions of objects in the two panels (see Figure 2.4).

2.1.2 Stereo Correspondence

Given two or more images of the same 3D scene, taken from different vantage points, the correspondence problem refers to the task of finding a set of points in one image which can

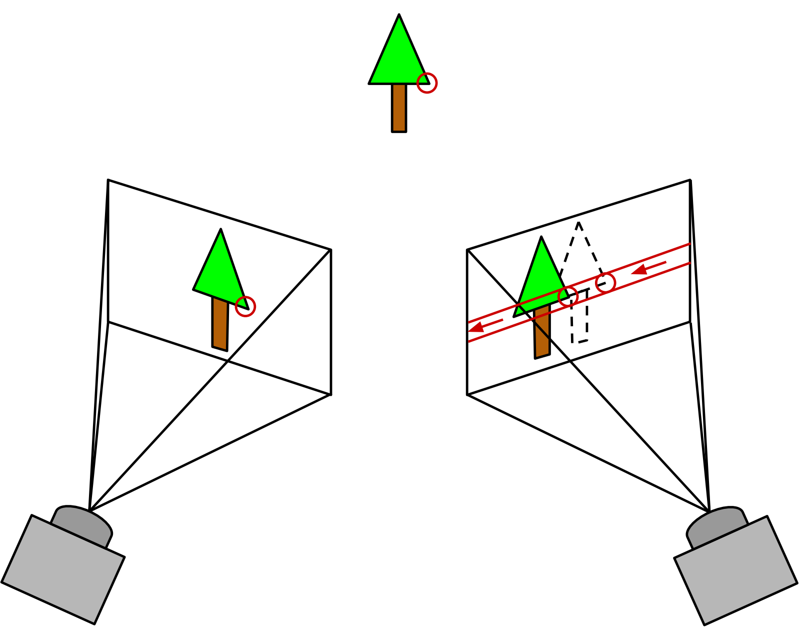


Figure 2.4: Diagram describing relationship of image displacement to depth with stereoscopic images, assuming flat co-planar images.

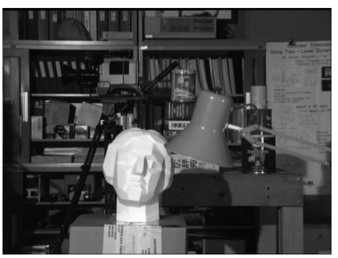
be identified as the same points in another image. To achieve this, points or features in one image are matched with the corresponding points or features in another image. The images can be taken from a different point of view, not at the same time, or with objects in general motion relative to the cameras.

From the earliest forays into visual perception, it was known that we depth is perceived based on the differences in appearance between the left and right eye. As a simple test, hold your finger vertically in front of your eyes and close each eye at a time. You will notice that the finger jumps left and right, relative to the background of the image. The same effect is visible in the image pair shown in Figure 2.5, in which the foreground objects shift left and right with respect to the background.

Stereo Matching is the process of employing two or more images and estimating a 3D model of the scene by finding matching areas of interest in the images and converting their 2D positions into 3D depths. This tackles the question of how to build a better 3D model, e.g., a sparse or dense depth map that assigns relative depths to pixels in the input images (see Figure 2.5).

There are two main types of techniques for stereo matching:

- **Sparse** algorithms
- **Dense** algorithms



(a) Tsukuba left

(b) Tsukuba right

(c) Depth map

Figure 2.5: Stereo reconstruction techniques can convert (**a-b**) a pair of images, into (**c**) a depth map.

Early stereo matching techniques were feature-based, i.e., they first extracted a set of potentially matchable image locations, using either interest operators or edge detectors, and then searched for matching locations in other images using a patch-based metric [BBH93, HMP92]. This limitations to sparse correspondences were in part due to computational resource limitations, but also were driven by a need to limit the answers produced by algorithms to matches with high confidence.

While sparse matching algorithms are still used on occasion, most stereo matching algorithms today focus on dense correspondence. This came to be since it is required for applications, such as image-based rendering or modeling. This problem is more difficult than sparse correspondence, since inferring depth values in regions with similar textures requires making a certain amount of inferences.

After carefully studying multiple algorithms to achieve our goal, we have chosen Block Matching. This is not one of the algorithms with the highest precision, but we believe it can be highly parallel and fits our requirement of real-time processing of video images.

2.1.3 Block Matching

The chosen algorithm follows a similar structure to the following publication [SS02]. With stereo cameras, objects in the field of view of these will appear at slightly different locations within the two images due to the cameras varying perspectives of the same scene. A standard method for calculating the disparity map is to use Block Matching, essentially, we will be taking a small region of pixels in the right image, and searching for the closest matching region of pixels in the left image. The structure and pseudo code of the implemented algorithm can be seen in algorithm 1.

Correlation based matching typically produces dense depth maps by calculating the disparity at each pixel within a neighborhood. This is achieved by taking a square window

Algorithm 1: Block Matching algorithm

Input: $left$, $right$, $aggregationdim$, $maxdisparity$
Output: $depthmap$

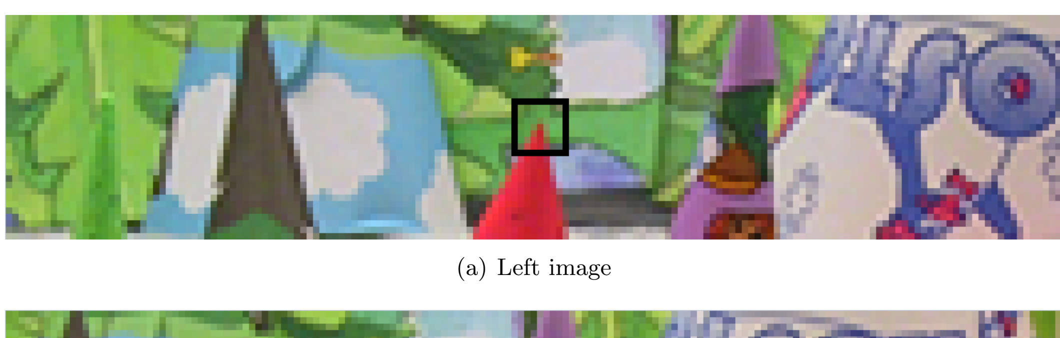
```
1  radius ← aggregationdim/2
2  for x ← 0 to width do
3      for y ← 0 to height do
4          offsetx ← x − radius
5          offsety ← y − radius
6          sum ← 0
7          bestsum ← −1
8          for d ← 0 to maxdisparity do
9              for i ← offsetx to aggregationdim + offsetx do
10                 for j ← offsety to aggregationdim + offsety do
11                     sum ← sum + |left[i ∗ width + j] − left[i ∗ width + j − d]|
12                 end
13             end
14             if sum < bestsum then
15                 bestsum ← sum
16                 bestd ← d
17             end
18             sum ← 0
19         end
20         depthmap[y ∗ width + x] ← bestd
21     end
22 end
```

of certain size around the pixel of interest in the reference image and finding the homologous pixel within the window in the target image, while moving along the corresponding scanline. The goal is to find the corresponding (correlated) pixel within a certain disparity range d that minimizes the associated error and maximizes the similarity (see Figure 2.6).

The most relevant part in this algorithm, is the comparison of the pixel windows between the left and right images. In our implementation we have implemented two approaches. The first, uses the following equation to calculate the *Sum of Absolute Differences* (SAD) between pixels:

$$\sum_{i,j \in W} |I_l(i,j) - I_r(i,j)| \tag{2.1}$$

Being W the aggregation window dimension, I_l the left image and I_r the right image. The second uses a *Sum of Squared Differences* (SSD) to achieve the same objective as we can observe in the next equation:



(a) Left image

(b) Right image search

Figure 2.6: Diagram describing relationship of image displacement to depth with stereoscopic images, assuming flat co-planar images.

$$\sum_{i,j \in W} (I_l(i,j) - I_r(i,j))^2 \tag{2.2}$$

Being W the search window dimension, I_l the left image and I_r the right image. One of these calculations is repeated for pixel windows in the right image at a distance $d \in [0, disparity_{max}]$.

In short, the correspondence process involves the computation of the similarity measure for each disparity value, followed by an aggregation and optimization step. Since these steps consume a lot of processing power but can be computed in parallel, there are significant speed-performance advantages to be had in optimizing the matching algorithm.

2.2 GPGPU

General-purpose computing on graphics processing units or GPGPU, is the use of a graphics processing unit (GPU), which typically handles computation only for computer graphics workloads, to perform calculations in applications traditionally handled by the central processing unit (CPU) [FTM02]. One can parallelize tasks even further using multiple graphics cards in one computer, or large numbers of graphics chips [FM04].

2.2.1 Architecture

Essentially, a GPGPU pipeline swaps data between one or more GPUs and CPUs and analyzes it as if it were in image or other graphic form. Because video cards can operate on 3D geometry and graphical data at speeds way faster than a traditional CPU, migrating data into graphical form and then using the GPU to process it and analyze it, can result in profound speed gains.

In a traditional **CPU architecture**, the CPU is able to access a large, general-purpose memory bank, called Random Access Memory (RAM) as is visually seen in Figure 2.7.

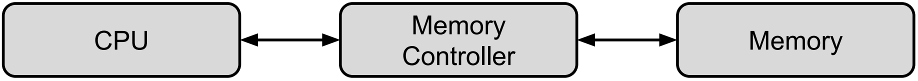


Figure 2.7: Diagram describing a simplified CPU architecture.

Nowadays, the CPU often contains more than one core, making CPUs capable of more than one task at a time. This makes modern CPUs much faster than their single core, scalar predecessors.

In contrast, a **GPGPU architecture** uses *Shared-Memory Multiprocessors* (SMP), these are a are a set of processors that all have their own local memory. These memory banks are shared within a thread group, but not between more than one of these groups. However, each compute unit also has access to a global memory bank, which is shared between all processors.

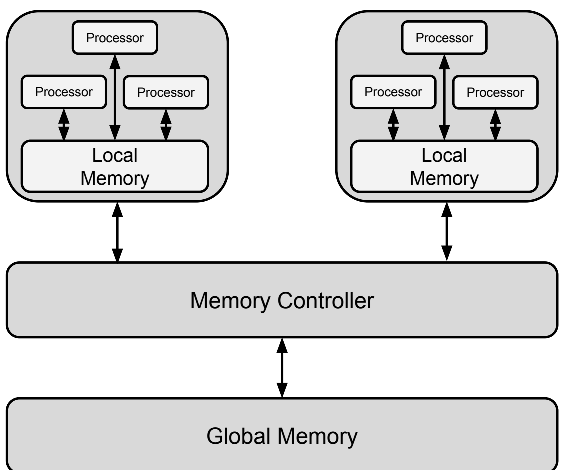


Figure 2.8: Diagram showing a simplified SMP architecture.

This is the parallel architecture that NVIDIA and AMD both use in their GPUs. Likewise, it is also the model enforced in the OpenCL and CUDA specification.

2.2.2 OpenCL

Open Computing Language (OpenCL) is a framework for writing programs that make use of heterogeneous platforms consisting of CPUs, GPUs, digital signal processors (DSPs), field-programmable gate arrays (FPGAs) and other processors. OpenCL provides parallel computing using task-based and data-based parallelism. In addition, it is also an open standard maintained by the non-profit technology consortium Khronos Group [1].

OpenCL interprets a computing system as a number of heterogeneous compute devices, which might be CPUs or accelerators such as graphics processing units, attached to a host processor (a CPU). It defines a C-like language for writing programs, called kernels, that are later executed on the compute devices. A single compute device typically consists of several compute units, which in turn comprise multiple *processing elements* (PEs).

A single kernel execution may run on all or many of the PEs in parallel. How a compute device is subdivided into compute units and PEs depends on vendor criteria; a compute unit can be thought of as a CPU core, but the notion of core is hard to define across all the types of devices supported by OpenCL and the number of compute units may not correspond to the number of cores that the vendor can advertise.

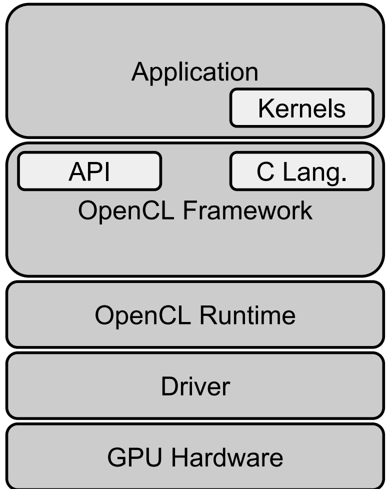


Figure 2.9: Diagram showing the simplified OpenCL architecture.

In addition to its this kernel programming language, OpenCL defines an *application pro-*

[1] https://www.khronos.org/

gramming interface (API) that allows normal programs running on the host to launch kernels on the compute devices and manage device memory, which is (at least conceptually) separate from host memory. Programs in the OpenCL language are intended to be compiled at run-time, so that applications that use OpenCL are portable between implementations for various host devices (see Figure 2.9).

OpenCL defines a four-level memory hierarchy for the compute device:

- **Global memory:** Shared by all processing elements, but has high access latency.
- **Read-only memory:** Smaller, low latency, writable by the host CPU but not the compute devices.
- **Local memory:** Shared by a group of processing elements.
- **Private memory:** Per-element private memory (registers).

2.2.3 CUDA

Compute Unified Device Architecture (CUDA) is a parallel computing platform and application programming interface (API) model created by NVIDIA. It allows software developers to use CUDA-enabled GPUs for general purpose processing. The CUDA platform is a software layer that gives direct access to the virtual instruction set and parallel computational elements of GPUs.

In contrast with OpenCL, this is a proprietary framework and is not compatible with such a varied array of devices, CUDA is only compatible with NVIDIA GPUs. It is compatible with programming languages such as C, C++ and Fortran. This makes it easier for specialists in parallel programming to utilize GPU resources, as opposed to previous API solutions like Direct3D and OpenGL, which required advanced skills in graphics programming.

Another difference between OpenCL and CUDA is how applications are compiled. In OpenCL kernels are compiled ad-hoc with the OpenCL framework, while in CUDA we will use a custom copiler called NVCC to compile the complete application or the parts that contain CUDA code. The CUDA architecture is built around a scalable array of multiprocessors, each one of them having several scalar processors, one multi-threading unit, and a shared memory chip. The multiprocessors are able to create, manage, and execute parallel threads, with a small overhead. The threads are grouped in blocks, which are executed in a single multiprocessor, and the blocks are grouped into grids. When a CUDA program calls a grid to be executed in the GPU, each one of the blocks in the

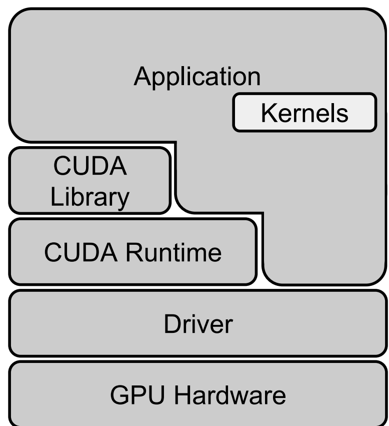


Figure 2.10: Diagram showing the simplified CUDA architecture.

grid is numbered and distributed to an available multiprocessor. When a multiprocessor receives a block to execute, it splits the threads in warps, a set of 32 consecutive threads. Each warp executes a single instruction at a time, so the best efficiency is achieved when the 32 threads in the warp executes the same instruction. Each time that a block finishes its execution, a new block is assigned to the available multiprocessor.

CUDA defines a similar memory hierarchy for compute devices:

- **Global memory:** Shared by all processing elements, but has high access latency.
- **Read-only memory:** Smaller, low latency, writable by the host CPU but not the compute devices.
- **Shared memory:** Shared by a block of threads.
- **Local memory:** Per-thread local memory.
- **Register memory:** Per-thread registers.

Chapter 3

GCVL: GPU Computer Vision Library

GCVL is a set of software tools and libraries that allow the user to run and implement common Computer Vision algorithms on modern GPUs. It comprises a set of OpenCL and CUDA tools, a Block Matching example algorithm implementation and base classes to implement custom algorithms.

3.1 Development Methodology

Agile software development [Coc01] is a combination of development methods that use iterative and incremental development, where requirements and solutions mature through collaboration between self-organizing, cross-functional teams. The motto of this method is "embrace change"; that is why it encourages adaptive planning, evolutionary development and delivery, a time-boxed iterative approach, and promotes quick and flexible response to change.

3.1.1 Agile manifesto

In February of 2001, several developers met at Snowbird, Utah resort, to debate different lightweight development methods. They published the *Manifesto for Agile Software Development* [Bec01] to define the approach that is now called agile software development.

The conclusions that we can reach from the manifesto's items are described below:

- **Individuals and Interactions**: In agile development self-organization and motiva-

tion are really important. Other values promoted by the manifesto are co-location[1] and pair programming[2].

- **Working software**: Working software will be utilized for more purposes than presenting documents to the client.
- **Customer collaboration**: The software requirements cannot be fully realized from the beginning of the software development cycle, so being in touch with the customer is really important.
- **Responding to change**: Agile development is keen on fast responses to change and continuous development.

More principles are mentioned in the manifesto, some of them are:

- Customer satisfaction by rapid delivery of useful software.
- Welcome changes even late in the development.
- Working software is the principal measure of progress.
- Maintaining a constant pace.
- Cooperation between business people and developers.
- Build projects around motivated individuals.
- Attention to technical excellence.
- Simplicity.

Agile methods break down tasks into small increments with minimal planning and normally long-term planning is not directly involved. *Iterations* are short timeframes that typically last from one to four weeks. A team works in each iteration through a full software development cycle; including planning, requirements analysis, design, coding, etc. This minimizes risk and facilitates adaptation to change. An iteration may not add enough new functionalities to warrant a market release, but the objective is to have an available release at the end of each iteration.

Team composition does not depend on corporate hierarchies or corporate roles of team members. They normally have the responsability of completing tasks that deliver the

[1]The act of placing multiple individuals within a single location.
[2]Two programmers work together at one workstation.

required functionalities that an iteration requires. How to meet an iteration's objectives is decided individually.

The "weight" of the method depends on the type of project, the planning and order of tasks in a generalist project should not be the same as in a research project.

Agile methods encourage face-to-face communication instead of written documents if possible. Most teams work in an open office (the *bullpen*), which makes this type of communication easier.

Each agile team contains a customer representative, that ensures that customer needs and company goals are aligned.

Most agile methods encourage a routine that includes daily face-to-face communication among team members. In a brief session team members tell each other what they achieved the previous day, what they are going to do today and the problems that have appeared.

As agile development emphasizes on working software as the primary measure of progress and has a clear preference in face-to-face communication this results in less written documentation than other methods. This does not mean that documentation should be disregarded, but that less emphasis is made on documentation because is not needed as much.

3.2 Design

GCVL is developed following an object oriented approach and using the C++ programming language. For GPGPU programming we have chosen OpenCL and CUDA, because of their inter-operation capabilities with OpenGL. This will allow us to process directly data that already resides in the GPU, without having to do expensive memory transfers. Because of the open nature of OpenCL, the software will be able to run on any graphics card vendor (AMD, NVIDIA or Intel) and the SO that the user prefers (Windows, OSX or Unix). For NVIDIA GPUs we have also developed a CUDA implementation, this way users will be able to squish the maximum amount of performance out of the green team's devices. Design patterns have been used as much as possible. This section briefly depicts the high-level design of GCVL in the form of class diagrams.

3.2.1 Class diagram

The high-level design of the **GCVL** library is depicted in the class diagram of Figure 3.1. In this design, the **General Tools** module contains utility classes that contain commonly

used functions and configuration parameters. In addition, the **CPU** module contains tools to implement multi-core CPU algorithms using GCVL, examples, template classes, etc.

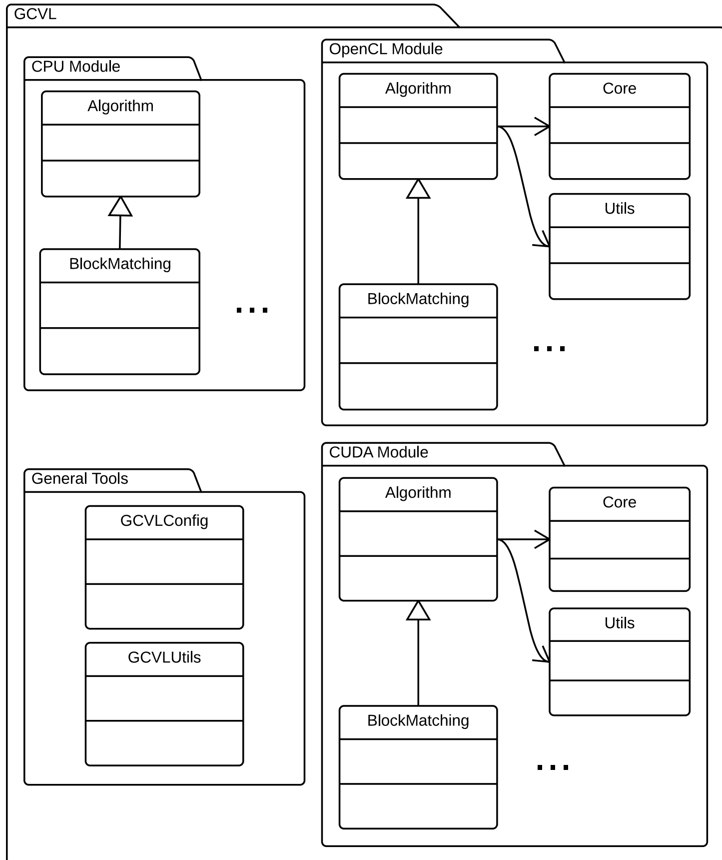


Figure 3.1: GCVL class diagram.

Furthermore, the **OpenCL** module contains classes related to the development of OpenCL algorithms (tools, example algorithms, kernels, etc.).

Finally, the **CUDA** module resembles the aforementioned module, containing tools, algorithm examples, kernels, etc. It possesses all the classes that are needed to implement CUDA programs in a simple manner.

All of the GPU modules present in GCVL can be compiled on demand using CMake options, so if the CUDA framework or the OpenCL libraries are not needed, they can be deactivated and the rest of the library can be used without any kind of issue.

3.2.2 General Tools

In the diagram of Figure 3.2, the following classes are the most relevant:

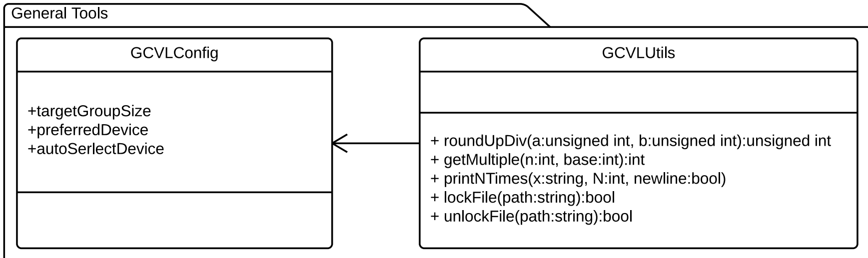


Figure 3.2: General Tools module class diagram.

- **GCVLConfig:** This class contains the configuration parameters present in GCVL. For example, the suppression of warnings, constant definitions, etc.
- **GCVLTools:** Class that contains utility functions used in all the modules present in GCVL, be it CPU or GPU modules.

3.2.3 CPU Module

This is a GPGPU oriented computation library, but it also provides code for multi-core CPUs. The relevant classes are shown in the class diagram Figure 3.3:

- **Algorithm:** This base class contains the template for the implementation of multi-core CPU algorithms in GCVL.
- **BlockMatching:** This class serves as an example of the implementation of a multi-core CPU algorithm in GCVL.

3.2.4 OpenCL Module

Since this is a GPU computing library, the first related module is the OpenCL module. In it, classes, algorithms, kernels that belong to the OpenCL implementation are packed together for convenience in the **opencl** name space. The most significant classes related to the management of the OpenCL algorithms are shown in the class diagram of Figure 3.4:

- **Algorithm:** This base class contains the template for the implementation of OpenCL algorithms in GCVL.
- **BlockMatching:** This class serves as an example of the implementation of a OpenCL algorithm in GCVL.

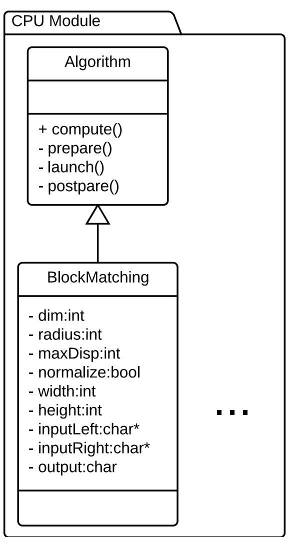


Figure 3.3: CPU module class diagram.

- **Core:** Class in charge of the creation of OpenCL platforms, selection of the best GPU, creation of queues, etc.
- **Platform:** Helper class that aids in the creation of a certain computing platform. AMD, NVIDIA, Intel, etc.
- **PlatformsList:** Class that holds a list with all the available platforms in the system.
- **Device:** Utility class that aids in the creation of compute devices.
- **DevicesList:** Class that holds a list of all the compute devices available in a platform and tries to guess the best one.
- **Kernel:** Wrapper class for the creation of kernels in OpenCL.
- **Array:** Helper class for OpenCL device array creation.
- **Data:** Utility class for the creation of OpenCL device data.

3.2.5 CUDA Module

The second GPU computing module gives the user the choice of using CUDA to implement GPU related algorithms using the **cuda** name space. The most relevant classes explained here are present in the class diagram in Figure 3.5:

- **Algorithm:** This base class contains the template for the implementation of CUDA algorithms in GCVL.

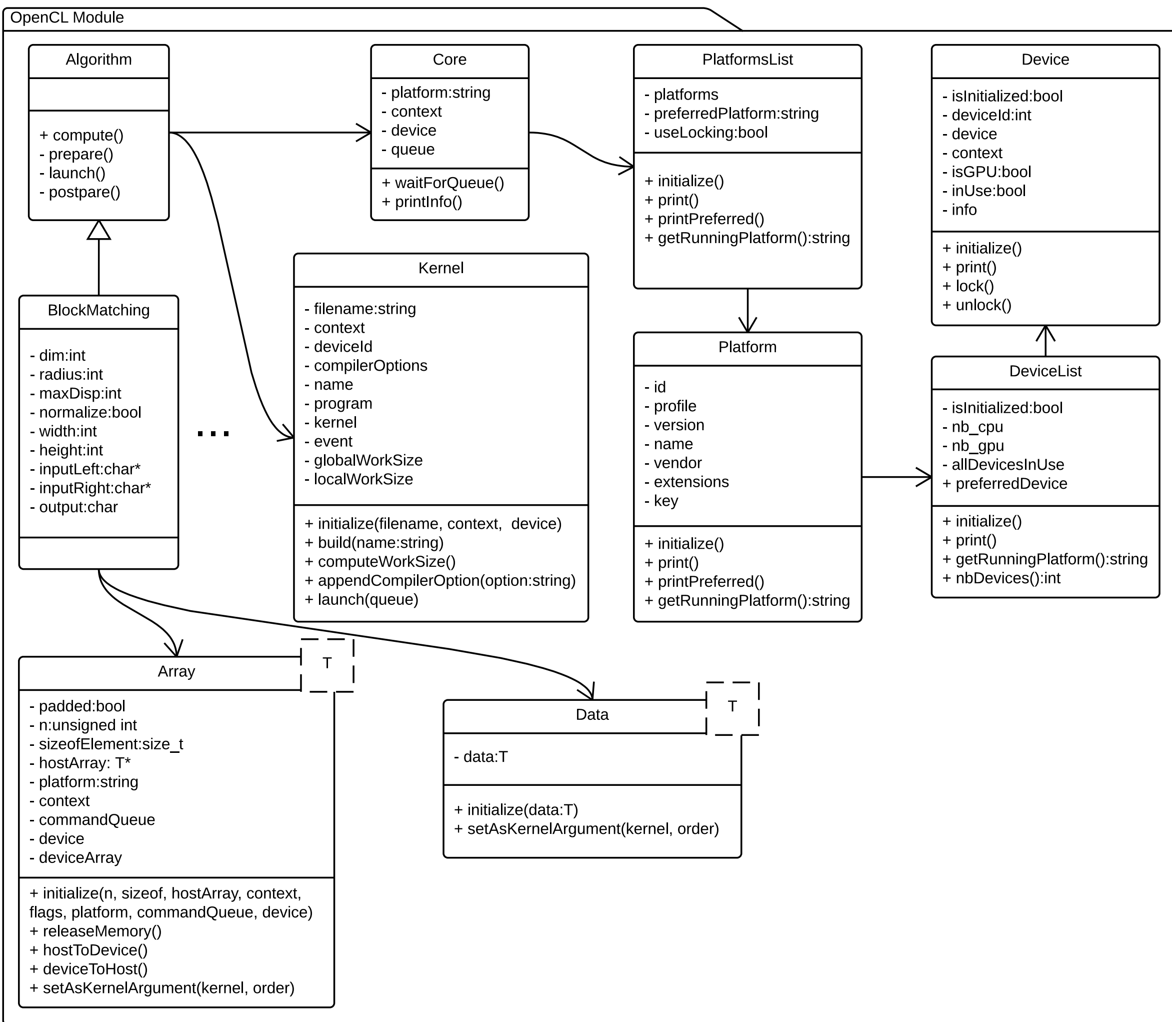


Figure 3.4: OpenCL module class diagram.

- **BlockMatching:** This class serves as an example of the implementation of a CUDA algorithm in GCVL.

- **Core:** Class in charge of the creation of the CUDA environment, selection of the best GPU, creation device arrays, etc.

- **Device:** Utility class that aids in the instantiation of compute devices.

- **DevicesList:** Class that holds a list of all the compute devices available in the system. In addition, it tries to guess the best one depending on architecture, compute units and clock speed.

- **Array:** Helper class for CUDA device array creation.

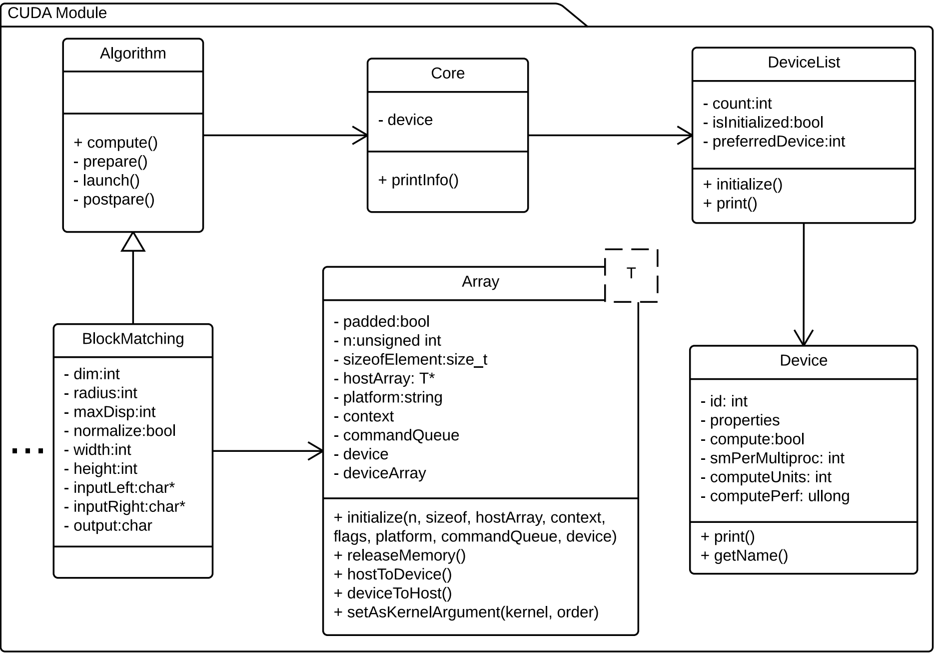


Figure 3.5: CUDA module class diagram.

3.3 Technology

As mentioned before, GCVL uses the C++ programming language. In order to create a true multi-platform framework, we have also employed the CMake [CHK+00] build system. The compilers in which the code has been tested are the following:

- **GCC 4.9:** The *GNU Compiler Collection* (GCC) is a compiler system created by the GNU Project supporting various programming languages.
- **Clang 6.1:** This is a compiler front end for C, C++ and other programming languages. It uses LLVM as its back end.
- **Visual C++ 12.0:** This compiler features tools for developing and debugging C++ code on Microsoft Windows platforms.

For GPGPU programming we have chosen **OpenCL** and **CUDA**, because of their interoperation capabilities with OpenGL. In addition these are the two most used GPGPU frameworks, being OpenCL the leading open source GPGPU framework and CUDA the leading proprietary framework.

In order to keep code versions organized and backed up, **Git** [Tor05] has been used as a version control system. This is a distributed revision control system with an emphasis on

speed, data integrity, and support for distributed, non-linear work-flows.

Since this is an open source project, we have also used **GitHub** [PWWH08], a web-based Git repository hosting service. It offers all of the distributed revision control and source code management functionality of Git as well as adding its own features. It provides access control and several collaboration features such as bug tracking, feature requests, task management, and wikis for every project. It is the ideal tool for open source collaboration.

In order to quickly pinpoint issues in the multiple target platforms, we will use the continuous integration tool **Travis CI**, an open-source hosted and distributed continuous integration service used to build and test projects hosted in GitHub.

In addition, we have used **Doxygen** to document the code for maintainability, which is a tool for writing software reference documentation. It is written within the code, and is thus relatively easy to keep up to date and understand.

3.4 Usage

Once the design concepts behind GCVL and its technology have been explained, the only task left is to know how the modules are used. We will first see how to run the CPU module algorithms, next we will see how the OpenCL tools and algorithms work and last but not least we will see how the CUDA module can be utilized to run the CUDA algorithms.

3.4.1 CPU Module

The CPU module usage is pretty straightforward, one only needs to include the corresponding algorithm class; in our example it will be the BlockMatching algorithm (see Listing 3.1). In this example, the two paths of the input images and the output image pointer are passed to the BlockMatching class, next the BlockMatching settings are set and the computation is started. The result will be available to the user in the output pointer.

3.4.2 OpenCL Module

The OpenCL module inner workings are a little more complex, but its usage is still really simple. The first step is to include the corresponding algorithm class and Core class; in our example it will be the BlockMatching algorithm (see Listing 3.2). In this case, the core, the two paths of the input images and the output image pointer are passed to the BlockMatching class. Next, the BlockMatching settings are set and the computation is

```
#include <gcvl/blockmatching.h>

int main(int argc, char *argv[]) {

        int dim = 5, maxDisp = 16;

        bool norm = true;

        std::unique_ptr<unsigned char[]> output;

        gcvl::BlockMatching bm(argv[1], argv[2], output);

        bm.setAggDim(dim);

        bm.setMaxDisp(maxDisp);

        bm.setNormalize(norm);

        bm.compute();

}
```

Listing 3.1: How to use the BlockMatching class.

started. The result will be available to the user in the output pointer.

3.4.3 CUDA Module

The CUDA module is used in a similar manner to the OpenCL module, but using a different namespace. The first step is to include the corresponding algorithm class and Core class; in our example it will be the BlockMatching algorithm (see Listing 3.3). In this case, the core, the two paths of the input images and the output image pointer are passed to the BlockMatching class. Next, the BlockMatching settings are set and the computation is started. The result will be available to the user in the output pointer.

```
#include <gcvl/opencl/oclcore.h>
#include <gcvl/opencl/oclblockmatching.h>
int main(int argc, char *argv[]) {
        int dim = 5, maxDisp = 16;
        bool norm = true;
        std::unique_ptr<unsigned char[]> output;
        gcvl::opencl::Core core;
        gcvl::opencl::BlockMatching bm(core, argv[1], argv[2], output);
        bm.setAggDim(dim);
        bm.setMaxDisp(maxDisp);
        bm.setNormalize(norm);
        bm.compute();
}
```

Listing 3.2: How to use the OpenCL BlockMatching class.

```
#include <gcvl/cuda/cudacore.h>
#include <gcvl/cuda/cudablockmatching.h>
int main(int argc, char *argv[]) {
    int dim = 5, maxDisp = 16;
    bool norm = true;
    std::unique_ptr<unsigned char[]> output;
    gcvl::cuda::Core core;
    gcvl::cuda::BlockMatching bm(core, argv[1], argv[2], output);
    bm.setAggDim(dim);
    bm.setMaxDisp(maxDisp);
    bm.setNormalize(norm);
    bm.compute();
}
```

Listing 3.3: How to use the CUDA BlockMatching class.

Chapter 4

Performance & Experimental Results

In this chapter we will delve in the obtained performance results in the two test machines, as well as taking a look at the experimental results obtained for a couple of test datasets. The images that will be used for testing are described in Table 4.1. They were obtained from the Middlebury Stereo Vision Page, that offers several test datasets from which we have chosen a couple of them [SS02, SP07, HS07].

	Dataset	Width	Height
	Tsukuba	384	288
	Bowling	1252	1110

Table 4.1: Datasets used in our tests.

4.1 Performance Results

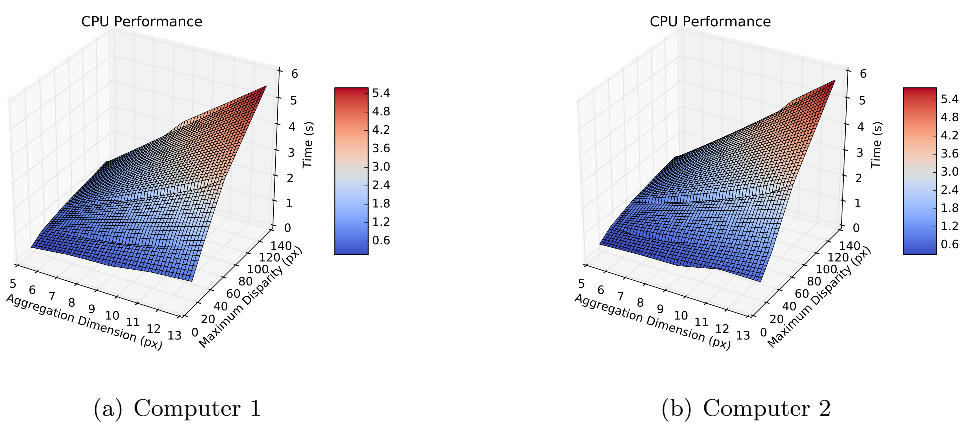
Two computers with different hardware were used for testing. The first, **Computer 1** has the following specs:

- Intel Core i7-3770 CPU (4 cores, 8 threads)
- NVIDIA GT 640 graphics card
- 16 GB of 1600 MHz DDR3 RAM
- WDC WD10EZRX HDD

The second, **Computer 2** has the following specifications:

- Intel Core i7-4930K CPU (6 cores, 12 threads)
- NVIDIA GTX 780Ti graphics card
- 16 GB of 2133 MHz DDR3 RAM
- WD Caviar Black HDD

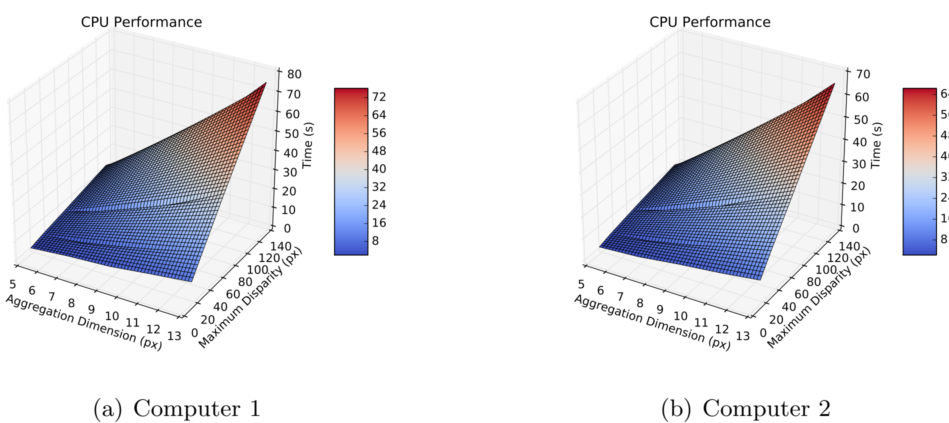
The operating system used to run the tests was Windows 7 x64, with the VC++ v110 compiler.



(a) Computer 1 (b) Computer 2

Figure 4.1: CPU performance results for the Tsukuba dataset.

The first test performed can be seen in Figure 4.1. In it we have used the Tsukuba dataset to test CPU performance with multiple window sizes and maximum disparities. As we can see, the diferences in small images when using a CPU with more cores are negligible; in fact, probably because of cache issues the CPU with more cores fares worse. As we can see, the performance achieved for the more demanding settings are not suitable for real-time applications even with low resolutions like these.

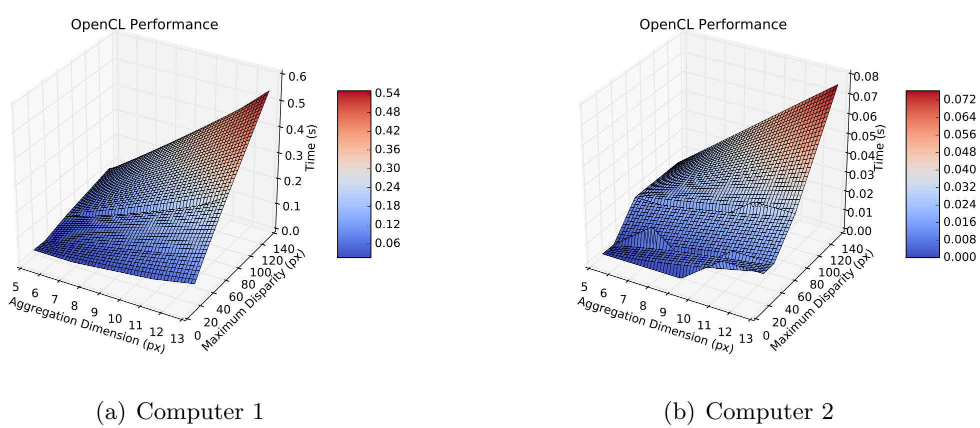


(a) Computer 1

(b) Computer 2

Figure 4.2: CPU performance results for the Bowling dataset.

Next, in Figure 4.2, we have tested the Bowling dataset for the same range of window sizes and maximum disparities. These dataset possesses a higher resolution and we start to see how the faster machine obtains better times than the slower one. But, since we have increased resolution substantially, the timings obtained with this dataset are even less suitable for real-time use cases.



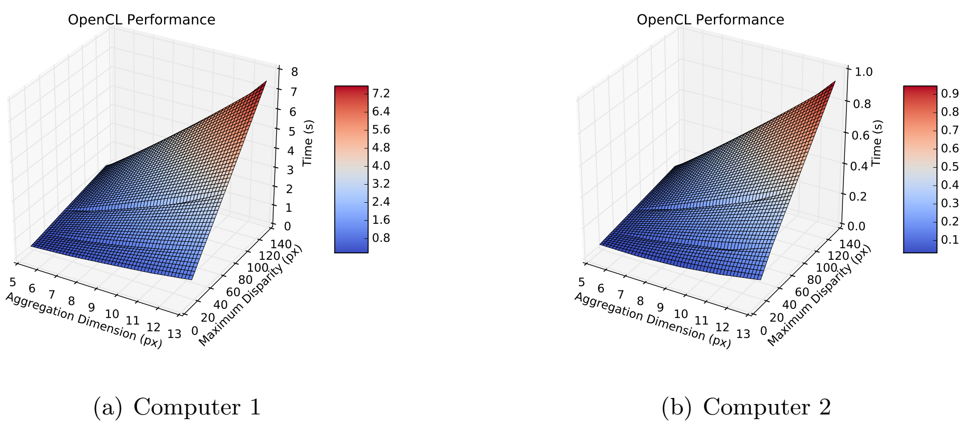
(a) Computer 1

(b) Computer 2

Figure 4.3: OpenCL performance results for the Tsukuba dataset.

Since CPU approaches do not seem like they are going to allow real-time processing of the chosen resolutions, we proceed in Figure 4.3 to test our OpenCL implementation with the Tsukuba dataset employing the same settings. We instantly see that this implementation improves computation times substantially. With this dataset, even the low end graphics card is able to compute the disparity map in times suitable for real-time applications.

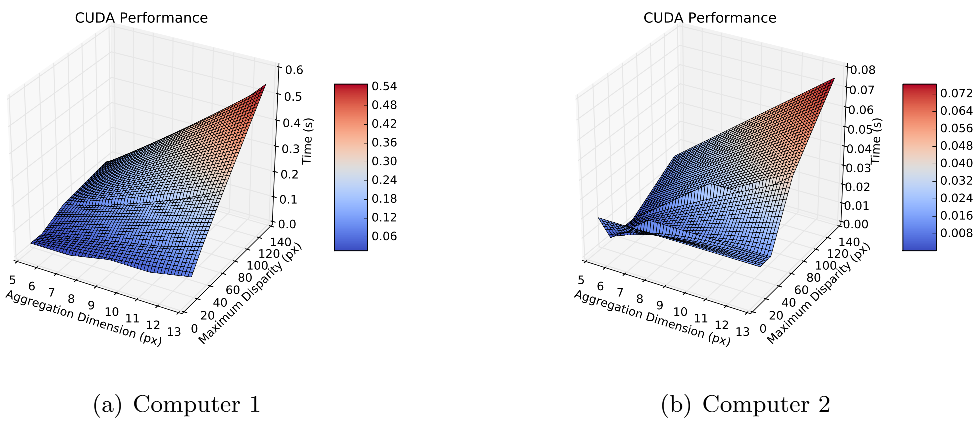
Computer 2 that has a more powerful graphics processor obtains even better results, achieving further performance gains.



(a) Computer 1 (b) Computer 2

Figure 4.4: OpenCL performance results for the Bowling dataset.

Next, in Figure 4.4, we have performed the same test on the Bowling dataset. This dataset requires a lot more processing power, that is why although the execution times have improved substantially, **Computer 1** is not obtaining the performance numbers necessary for real-time processing. In contrast, **Computer 2** does achieve results capable of processing several frames of video per second.

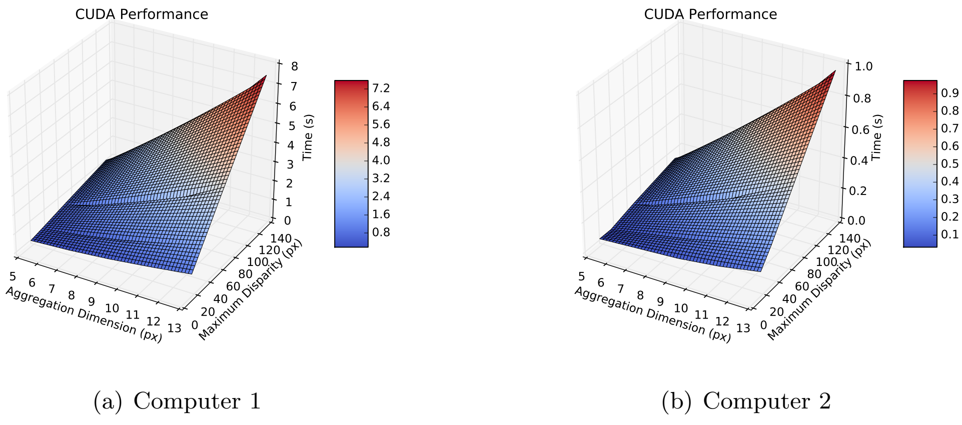


(a) Computer 1 (b) Computer 2

Figure 4.5: CUDA performance results for the Tsukuba dataset.

Once we have seen how the OpenCL implementation performs, the only task left is to test the CUDA module. The first test performed can be seen in Figure 4.5, in it we have tested the CUDA implementation with the Tsukuba dataset. The results obtained

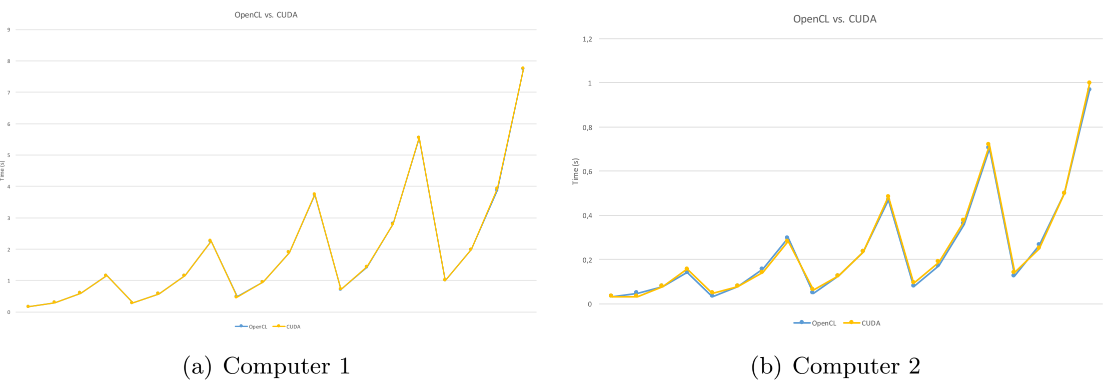
resemble closely the OpenCL results, in some cases winning and losing by small margins.



(a) Computer 1 (b) Computer 2

Figure 4.6: CUDA performance results for the Bowling dataset.

The next test performed analyzes the CUDA code with the Bowling dataset (see Figure 4.6). Again, the results obtained are quite close to the OpenCL implementation obtaining better or worse results depending on the settings used but with small differences.

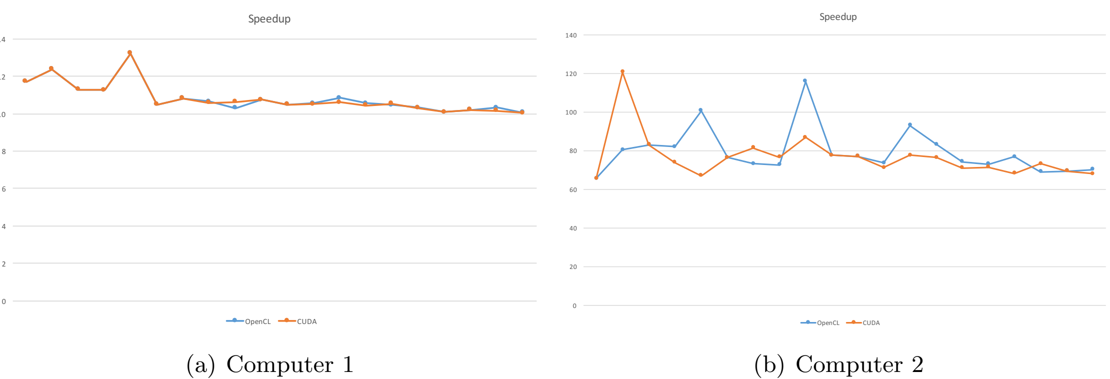


(a) Computer 1 (b) Computer 2

Figure 4.7: OpenCL vs. CUDA in the Bowling dataset.

Since execution times for the CUDA and OpenCL implementations are so close, in Figure 4.7 we have compared the execution times obtained in both test machines for the most demanding dataset. As mentioned before both implementations are really close in terms of performance.

The last analysis performed using GCVL compares the speedups obtained with the GPU against the CPU implementation of the algorithm. The CPU execution times used to calculate the speedup are from the high end CPU in **Computer 2**. The speedups obtained

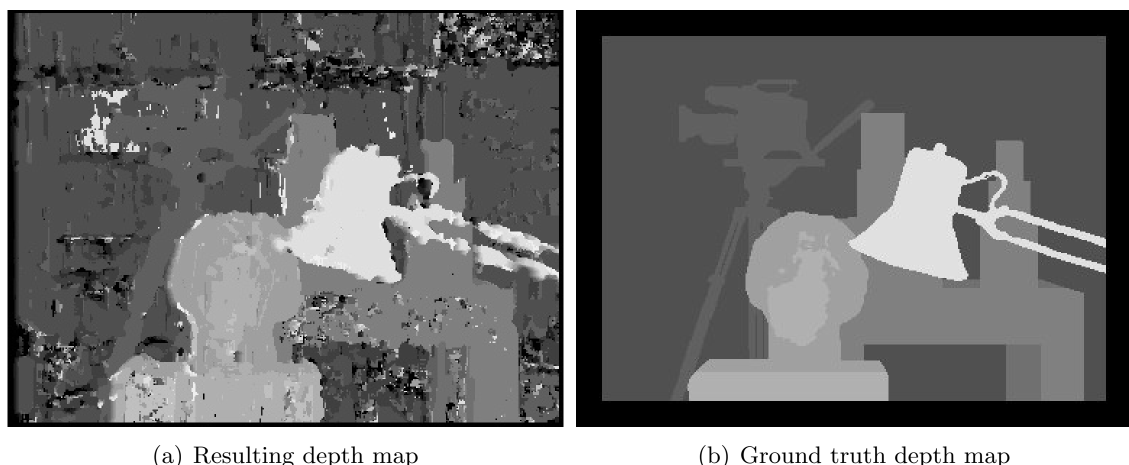


(a) Computer 1 (b) Computer 2

Figure 4.8: Speedup obtained in the Bowling dataset.

with both graphics cards are respectable, but the ones obtained with the **Computer 2** are specially remarkable, with an average speedup of 77x.

4.2 Experimental Results

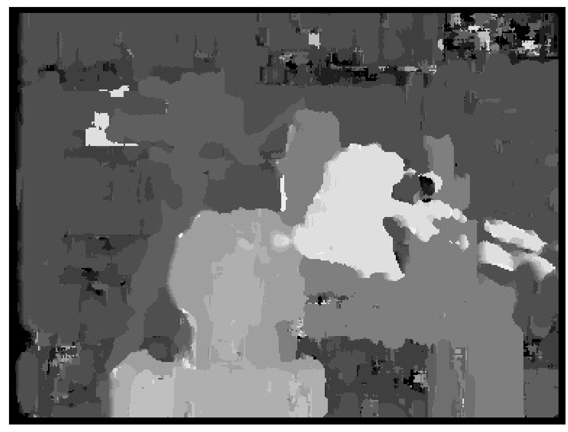


(a) Resulting depth map (b) Ground truth depth map

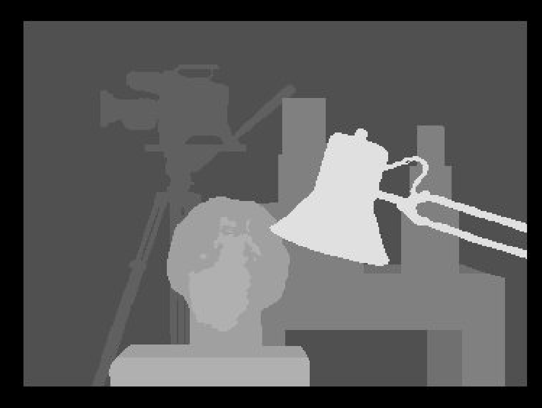
Figure 4.9: Depth map obtained with a maximum disparity of 16 and an aggregation window of 5 in the Tsukuba dataset.

In order to test the implemented algorithm, we have processed the test images with GCVL. The first results obtained were the depth maps of the Tsukuba dataset, they can be seen in Figure 4.9 and Figure 4.10.

As we can see in the tests, as we increase the aggregation window size, the depth map becomes smoother. The main drawback when increasing this parameter, is that a loss of detail occurs, since the pixel window compared is bigger and the estimation is more



(a) Resulting depth map



(b) Ground truth depth map

Figure 4.10: Depth map obtained with a maximum disparity of 16 and an aggregation window of 9 in the Tsukuba dataset.

statistically robust but less coarse.

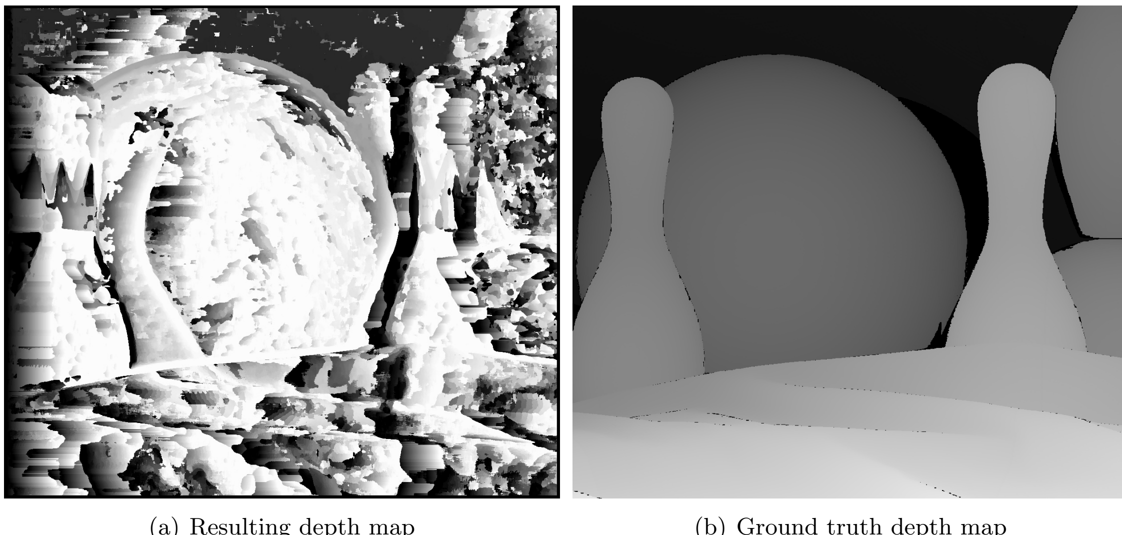
(a) Bowling dataset

(b) Textureless map

Figure 4.11: Textureless map of the Bowling dataset, pixels marked as white are low-texture regions.

Moreover, we can see that there are some artifacts in the obtained depth maps. The first reason why these artifacts occur is because of half-occluded (objects in the scene in one image, and not in the other) pixels in the final disparity map. There can also be occluded regions in the left and right images. In addition, there can be regions where there is little or no texture in the scene (a good example of this can be seen in the bowling ball of the Bowling dataset). These can be defined in [SS02] as regions where the squared horizontal

intensity gradient averaged over a square window of a given size is below a given threshold (see Figure 4.11).



(a) Resulting depth map (b) Ground truth depth map

Figure 4.12: Depth map obtained with a maximum disparity of 100 and an aggregation window of 13 in the Bowling dataset.

We can see some of the aforementioned effects in the Second results obtained using the Bowling dataset. The resulting depth maps can be viewed in Figure 4.12 and Figure 4.13.

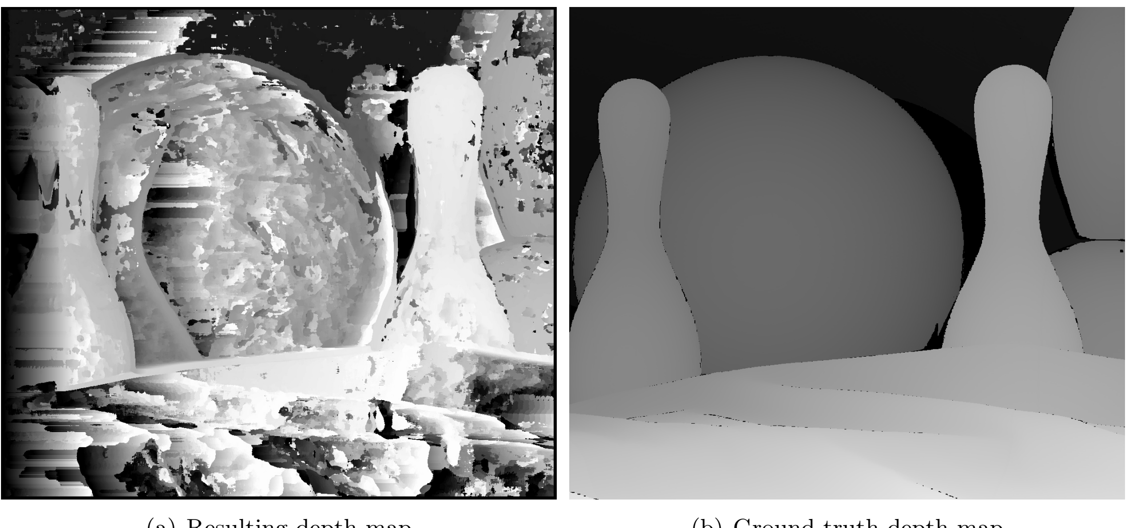
In the first test we have seen how the aggregation window size influences the smoothness of the disparity map, now we have used a different maximum disparity in each case. If we choose a lower maximum disparity, the algorithm will be faster, but one runs the risk of not finding the corresponding window in the right image.

In our datasets from the Middlebury Stereo's website, then the answer is simple. The maximum disparity value is the maximum value of the pixel in the ground truth map, divided by the scale factor. If the user has a stereo vision set-up taking real-world imagery, then we will have to do a bit of math to calculate the values. The first equation that we will use is the following:

$$r = \frac{bf}{Nx} \tag{4.1}$$

Being r the range of the object that we are trying to compute, b the distance between the centers of the two cameras, f the focal length of the image sensor, x the pixel size of the sensor and N the maximum disparity value.

Another aspect to account for is the uncertainty in detecting objects at a certain range



(a) Resulting depth map (b) Ground truth depth map

Figure 4.13: Depth map obtained with a maximum disparity of 170 and an aggregation window of 13 in the Bowling dataset.

and the actual range itself. The equation that relates these two variables is the next:

$$\Delta r = \left(\frac{r^2}{bf}\right) x \Delta N \tag{4.2}$$

Being Δr the uncertainty in detecting the object at a certain range and ΔN the change in disparity value. This means, that for a certain range, we will have some uncertainty in the obtained measurements. For more information the user can check [KAW08a, KAW08b].

Chapter 5

Conclusions and future lines of work

In this chapter we will briefly take a look at the conclusions reached after finishing this project, and the possible future lines of work that the project can follow.

5.1 Conclusions

The first conclusion reached, has been that all of the objectives of the project were met:

- Studying different Stereo Matching techniques.
- Design, implementation and documentation of tools to ease GPGPU programming.
- Design, implementation and documentation of the chosen algorithm.
- CPU parallelization of the implemented algorithm.
- GPU parallelization of the implemented algorithm.

After finishing and achieving all the aforementioned objectives, the other conclusions that have been reached are:

- **GPGPU is not always the answer:** As seen in the results, if the workload is not complex enough to compensate for kernel setup time, GPU computation time will be higher than the time it takes the CPU to process the data. One has to carefully consider if the workload and the chosen algorithms are optimal for GPU parallelization or a lot of time can be wasted.

- **GPGPU tools significantly speed up the development process:** Creating multi-platform OpenCL and CUDA algorithms is not a trivial task. Without the help of this library the implementation of the algorithms would be error prone and slower.

- **GPGPU device selection is complex:** The automatic selection of the best GPU computing device, is not an easy task. Improvements in architecture, clock speed, etc. can render devices with more compute units obsolete; making it difficult to automatically guess the best device. This is specially complex in OpenCL, because of the variety of devices present in the ecosystem.

- **GPGPU improves performance substantially:** In Computer Vision tasks GPU computing fits really well. In a wide variety of algorithms, the workloads are highly parallel; allowing us to squeeze the maximum amount of processing power out of GPUs.

5.2 Future lines of work

After finalizing the work on this project, several ideas for the expansion of the library come to mind:

- **GPU kernels optimization:** As of know the implemented kernels of the Block Matching algorithm are semi-naive implementations. It would be interesting to further optimize these kernels so they would use local memory and would access the data with more optimal patterns or other types of improvements.

- **Multi-GPU:** As of now, the GPU algorithms and tools are designed to work on a single GPU device. The next step could be the adaptation of the algorithms to work on multiple GPUs. This would increment the performance gap even further against CPUs.

- **MPI tools:** Since this is a GPU computing library, we have not delved into the usage of MPI to parallelize the algorithms. But in HPC, the usage of supercomputers sometimes requires MPI implementations. The creation of helper tools and their integration in multi-platform systems could be an interesting step forward.

- **Boost Compute:** The usage of this library for the OpenCL module, could provide STL-like common algorithms, common containers and iterators (for example, vectors, etc.).

- **Thrust:** As in the proposed OpenCL module aforementioned improvement, in CUDA we could use Thrust to achieve a similar result.
- **Optimization of the algorithm:** The improvement of the algorithm in terms of performance using message passing techniques, could also yield improvements for the CPU and GPU implementations.
- **Block Matching:** To improve the block matching algorithm, new correlation based similarity measures should be implemented. Several algorithm improvements could also be made to improve CPU and GPU computing times.

Bibliography

[BBH93] Robert C Bolles, Harlyn H Baker, and Marsha Jo Hannah. The jisct stereo evaluation. In *DARPA Image Understanding Workshop*, pages 263–274, 1993. 10

[Bec01] Kent Beck. Manifesto for agile software development. *Agile Alliance*, 2001. 17

[CHK+00] Andy Cedilnik, Bill Hoffman, Brad King, Ken Martin, and Alexander Neundorf. CMake. `http://www.cmake.org`, 2000. 24

[Coc01] Alistair Cockburn. *Agile Software Development*. ISBN 978-0-20-169969-2. Addison-Wesley Professional, first edition, 2001. 17

[FM04] James Fung and Steve Mann. Using multiple graphics cards as a general purpose parallel computer: Applications to computer vision. In *Pattern Recognition, 2004. ICPR 2004. Proceedings of the 17th International Conference on*, volume 1, pages 805–808. IEEE, 2004. 12

[FP03] David A. Forsyth and Jean Ponce. *Computer Vision, A Modern Approach*. ISBN 0-13-085198-1. Prentice Hall, second edition, 2003. 2, 5

[FTM02] James Fung, Felix Tang, and Steve Mann. Mediated reality using computer graphics hardware for computer vision. In *Wearable Computers, 2002.(ISWC 2002). Proceedings. Sixth International Symposium on*, pages 83–89. IEEE, 2002. 12

[HMP92] Yuan C Hsieh, David M McKeown, and Frederic P Perlant. Performance evaluation of scene registration and stereo matching for artographic feature extraction. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, (2):214–238, 1992. 10

[HS07] Heiko Hirschmüller and Daniel Scharstein. Evaluation of cost functions for stereo matching. In *Computer Vision and Pattern Recognition, 2007. CVPR'07. IEEE Conference on*, pages 1–8. IEEE, 2007. 29

[KAW08a] Bahador Khaleghi, Siddhant Ahuja, and QM Jonathan Wu. An improved real-time miniaturized embedded stereo vision system (mesvs-ii). In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08. IEEE Computer Society Conference on*, pages 1–8. IEEE, 2008. 37

[KAW08b] Bahador Khaleghi, Siddhant Ahuja, and QM Jonathan Wu. A new miniaturized embedded stereo-vision system (mesvs-i). In *Computer and Robot Vision, 2008. CRV'08. Canadian Conference on*, pages 26–33. IEEE, 2008. 37

[Pra12] Stephen Prata. *C++ Primer Plus*. ISBN 978-0-321-77640-2. Addison-Wesley, sixth edition, 2012.

[PWWH08] Tom Preston-Werner, Chris Wanstrath, and PJ Hyett. Github. `https://github.com/`, 2008. 25

[SP07] Daniel Scharstein and Chris Pal. Learning conditional random fields for stereo. In *Computer Vision and Pattern Recognition, 2007. CVPR'07. IEEE Conference on*, pages 1–8. IEEE, 2007. 29

[SS01] Linda G. Shapiro and George C. Stockman. *Computer Vision*. ISBN 0-13-030796-3. Prentice Hall, first edition, 2001. 1, 5

[SS02] Daniel Scharstein and Richard Szeliski. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *International journal of computer vision*, 47(1-3):7–42, 2002. 10, 29, 35

[Sze11] Richard Szeliski. Stereo correspondence. In *Computer Vision*, Texts in Computer Science, pages 467–503. Springer London, 2011. 5

[Tor05] Linus Torvalds. Git. `http://git-scm.com`, 2005. 24