Programming methodologies for ADAS applications in parallel heterogeneous architectures

Djamila Dekkiche

NNT : 2017SACLS388

Thèse de doctorat de l'Université Paris-Saclay préparée à Université Paris-Sud

Laboratoire des Systems et Applications des Technologies de l'Information et de l'Energie

Ecole doctorale n°580
Sciences et Technologies de l'Information et de la Communication
Spécialité de doctorat
Traitement du signal et des images

par

Mme. Djamila DEKKICHE

Programming methodologies of ADAS applications on parallel heterogeneous architectures

Thèse présentée et soutenue à "Digiteo Labs, Gif-sur-Yvette", le 10 Novembre 2017.

Composition du Jury :

M.	François VERDIER	Professeur, Université de Nice	(Président du jury)
M.	Vincent FREMONT	Maître de conférences, UTC, Compiègne	(Rapporteur)
M.	Dominique HOUZET	Professeur, Grenoble-INP, Grenoble	(Rapporteur)
M.	Marc DURANTON	Expert international, CEA LIST	(Examinateur)
M.	Alain MERIGOT	Professeur, Université Paris-sud	(Directeur de thèse)
M.	Bastien VINCKE	Maître de conférences, Université Paris-sud	(Co-Directeur de thèse)
M.	Witold KLAUDEL	Chef de projet, Renault, IRT SystemX	(Invité)

Title: Programming methodologies of ADAS applications on parallel heterogeneous architectures

Keywords: ADAS, parallel computing, computer vision, embedded systems

Abstract: Computer Vision (CV) is crucial for understanding and analyzing the driving scene to build more intelligent Advanced Driver Assistance Systems (ADAS). However, implementing CV-based ADAS in a real automotive environment is not straightforward. Indeed, CV algorithms combine the challenges of high computing performance and algorithm accuracy. To respond to these requirements, new heterogeneous circuits are developed. They consist of several processing units with different parallel computing technologies as GPU, dedicated accelerators, etc. To better exploit the performances of such architectures, different languages are required depending on the underlying parallel execution model.
In this work, we investigate various parallel programming methodologies based on a complex case study of stereo vision. We introduce the relevant features and limitations of each approach. We evaluate the employed programming tools mainly in terms of computation performances and programming productivity. The feedback of this research is crucial for the development of future CV algorithms in adequacy with parallel architectures with a best compromise between computing performance, algorithm accuracy and programming efforts.

Titre: Méthodologies de programmation d'algorithmes de traitementd'images sur des architectures parallèles et hétérogènes

Mots Clés : ADAS, traitement parallèle, computer vision, systèmes embarqués

Résumé: La vision par ordinateur est primordiale pour la compréhension et l'analyse d'une scène routière afin de construire des systèmes d'aide à la conduite (ADAS) plus intelligents. Cependant, l'implémentation de ces systèmes dans un réel environnement automobile et loin d'être simple. En effet, ces applications nécessitent une haute performance de calcul en plus d'une précision algorithmique. Pour répondre à ces exigences, de nouvelles architectures hétérogènes sont apparues. Elles sont composées de plusieurs unités de traitement avec différentes technologies de calcul parallèle: GPU, accélérateurs dédiés, etc. Pour mieux exploiter les performances de ces architectures, différents langages sont nécessaires en fonction du modèle d'exécution parallèle.
Dans cette thèse, nous étudions diverses méthodologies de programmation parallèle. Nous utilisons une étude de cas complexe basée sur la stéréo-vision. Nous présentons les caractéristiques et les limites de chaque approche. Nous évaluons ensuite les outils employés principalement en terme de performances de calcul et de difficulté de programmation. Le retour de ce travail de recherche est crucial pour le développement de futurs algorithmes de traitement d'images en adéquation avec les architectures parallèles avec un meilleur compromis entre les performances de calcul, la précision algorithmique et la difficulté de programmation.

To my loving parents Ahmed and Ferroudja

To my husband Amirouche

To my brother and sisters

To my family

To my friends

This work would not have been possible without your love and support.

Acknowledgements

This dissertation would not have been possible without the guidance and the support of several individuals, who contributed and extended their valuable assistance in the working out and completion of this study.

First, I give thanks to God for protection and ability to do work.

Foremost, I would like to express my sincere gratitude to both my advisor Alain Mérigot and co-supervisor Bastien Vincke for the continuous support of my Ph.D study and research, for their patience, confidence and immense knowledge. Their guidance helped me in all the time of research and writing of this thesis. I thank them for giving me the opportunity to work under their direction and for believing in my capacities.

I would like to thank the rest of my thesis committee for accepting to judge this work: Mr. Marc Duranton, Pr. Dominique Houzet, Mr. Vincent Frémont, Pr. François Verdier, Mr. Witold Klaudel and Mr. Joel Falcou.

This research work has been carried out in the framework of the Technological Research Institute SystemX, and therefore granted with public funds within the scope of the French Program "Investissements d'Avenir". My sincere thanks go to IRT SystemX Staff, ELA project members for providing the environment and required resources to complete this work. I would like to thank my referee Mr. Aymen Boudguiga for his support, help and for correcting my scientific papers.

Last but not the least, I would like to thank my family: my parents, Dad and Mom, you have always encouraged me, you did all your best to help and support me. I would never be able to thank you enough. I would like to thank also my husband who supported and always believed on me.

Publications

International Journal

1. "Targeting System-Level and Kernel-Level Optimizations of Computer Vision Applications on Embedded Systems"
   D. Dekkiche, B. Vincke and A. Mérigot, Journal of Low Power Electronics (JOLPE), to appear on December 2017.

International Conferences and Symposiums

1. "Targeting System-Level and Kernel-Level Optimizations of Computer Vision Applications on Embedded Systems"
   D. Dekkiche, B. Vincke and A. Mérigot, International Symposium on Embedded Computing and System Design (ISED), Patna, India, 2016.
2. "Investigation and Performance Analysis of OpenVX Framework Optimizations on Computer Vision Applications"
   D. Dekkiche, B. Vincke and A. Mérigot, International Conference on Control, Automation, Robotics and Vision (ICARCV), Phuket, Thailande, 2O16.
3. "Vehicles Detection in Stereo Vision Based on Disparity Map Segmentation and Objects Classification"
   D. Dekkiche, B. Vincke and A. Mérigot, International Symposium on Visual Computing (ISVC), pp. 762-773, Las Vegas, USA, 2015.

Contents

List of Figures

List of Tables

-

1

Introduction

As reported by the World Health Organization (WHO) in 2015 [2], every year approximately 1.25 million people die as a result of road traffic crash. Between 20 and 50 million more people suffer from non-fatal injuries, with many incurring a disability as a result of their injury. Most of these accidents are attributed to human errors or drivers distractions. Although the big efforts spent since 1990s to develop safety technologies solutions such as airbags, anti-lock brakes, etc, human casualties in the road environment are still too high. To better manage the road traffic and reduce the risk of accidents, new technologies have been proposed, called Advanced Driver Assistance Systems (ADAS), such as the Adaptive Cruise Control (ACC) and the Lane Departure Warning (LDW) systems. ADAS are on-board intelligent systems developed to avoid the risk of accidents and improve road safety by assisting the drivers in their driving tasks. These systems help with monitoring, warning, braking, and steering tasks while driving. The demand of ADAS is expected to increase over the next decade. This is largely requested by regulatory and consumers interest for safety applications that protect drivers and reduce accidents. Indeed, both the European Union and the United States are planing to equip all vehicles with autonomous emergency-braking systems and forward-collision warning systems by 2020.

This research work has been carried out in the framework of the Technological Research Institute SystemX (IRT SystemX), and therefore granted with public funds within the scope of the French Program "Investissements d'Avenir". The IRT SystemX is specialized in digital sciences, it holds collaborative projects bringing together research laboratories and industrial companies. This work is carried out in a project called "Electronique et Logiciel pour l'Automobile" with academic laboratories and automobile

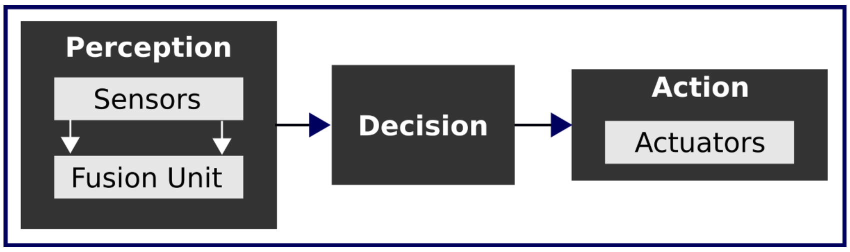


Figure 1.1: Architecture of an ADAS

industrial partners such as: Renault, PSA, Intempora, Continental, Valeo .... The project aims to study and investigate the different aspects related to autonomous cars such as security, vision-based ADAS and real time supervisors. My work is involved within the second task "image processing" where we investigate the different parallel programming tools as well the embedding methodologies of ADAS applications.

1.1 Context : Embedding Vision-Based ADAS

An ADAS is considered as complex real time embedded system which consists of three important layers [3], [4] as illustrated in Figure 1.1.

- The *perception* layer includes a set of sensors such as radars and cameras. It may also include a sensor data fusion unit which allows the computation of appropriate sensors data to estimate a consistent state of a vehicle and its environment.
- The *decision* layer uses the data fusion unit outputs to analyze the current situation and to decide the appropriate actions to be transmitted to actuators.
- The *action* layer receives the actions from the decision layer, and either it delivers visual, acoustic and/or haptic warning information to the driver, or it provides automatic actions such as braking.

As depicted in Figure 1.1, the perception layer consists of a set of sensors and a fusion unit. Usually we find passive sensors as cameras, and active sensors such as radars and lidars . Computer vision, together with radar and lidar, is at the forefront of technologies that enable the evolution of ADAS. Radars and lidars offer some advantages, such as long detection range (about 1-200 m), and capability to operate under extreme weather conditions. However, it is vulnerable to false positives, especially around road curves, since it is not able to recognize objects. Camera-based systems have also their own limitations. They are very affected by weather conditions, and they are not as reliable as radar when obtaining depth information. On the other hand, they have a wider field of view, and more importantly, they can recognize and categorize objects. From cost point of view, cameras are usually cheaper than radars. For all these reasons, modern ADAS applications use sensor fusion to combine the strengths of all these technologies. Normally, a radar or lidar sensor is used to detect potential candidates, and then, during a second stage, computer vision is applied to analyze the detected objects. However, not all applications need sensor fusion, and some applications such as Lane Departure Warning (LDW) or Driver Fatigue Warning (DFW) can rely entirely on a camera-based system.

1.1.1 ADAS Challenges and Opportunities

There are several challenges to design, implement, deploy, and operate ADAS. The system is expected to be *fast* in processing data, *accurately predict* context, and react in *real time*. In addition, it is required to be *robust*, *reliable*, and have *low error* rates. There have been significant efforts and researches to solve all these challenges and to develop the technology that will make ADAS and autonomous driving a reality.

ADAS are considered as hard real-time control systems in the automotive domain. ADAS use a lot of data reported from several sensors. These data must be updated regularly to reflect the current environment state. Thus, these systems need to be managed by real-time database systems in order to store and manipulate real-time data efficiently. However, the design of ADAS is highly complex; it is difficult to model the time constraints related to both data and transactions.

1.1.2 Real-Time ADAS

Computer vision algorithms are traditionally designed to give high accuracy with less focus on speed or execution time [5]. However, ADAS require images to be processed **as they are captured** so that the car can react quickly to changes. A fast computer vision algorithm has to complete its execution within a **predictable time bound** so that time spent on one frame does not delay the next frame.

An autonomous vehicle must be as good as or better than a human driver. The average reaction time of an human driver alert is 700 ms [6]. Hence, each ADAS task must have a runtime that is bounded above by 700 ms. In safety-critical tasks that could prevent the loss of life such as pedestrian detection, the bound should be much lower. In an automotive real-time system, many programs will execute as periodic tasks, with different periods depending on the program's purpose. For example, the vehicles detector **must run as frequently as possible**. When the vehicles detector finishes processing an image it should **immediately** start again processing the newest image from the camera since what it has been processed is already out of date. In system with lower criticality, such as lane departure warning, the period between executions of the task may be much longer.

1.2 Problem Statement

The role of computer vision in understanding and analyzing the driving scene is of great importance in order to build more intelligent driver assistance systems. However, the implementation of these Computer vision-based applications in a real automotive environment is not straightforward. The vast majority of works of the scientific literature test their driver assistance algorithms on standard PCs. When these algorithms are ported to an embedded device, they see their performance degraded and sometimes they cannot even be implemented. Since there are several requirements and constrains to be taken into account, there is a big gap between what is tested in a standard PC and what finally runs in the embedded platform. Furthermore, there is no standard hardware and software for a specific application. Hence, different architectures and programming tools have been proposed by the industry and the scientific community and it is on still non-mature markets.

1.2.1 Challenges of Embedded Image Processing Algorithms

Image processing algorithms combine the challenges of high computing performance and algorithm accuracy to cope with the rapidly rising resolution and frame rate of sensors and the increasing complexity of image processing algorithms.

To understand the challenge of efficient image processing, consider a 3×3 box filter implemented as separate horizontal and vertical passes. We might write this in `C++` as a sequence of two loop nests. An efficient implementation on a modern CPU requires `SIMD` vectorization and multi-threading. However, once we start to exploit parallelism,

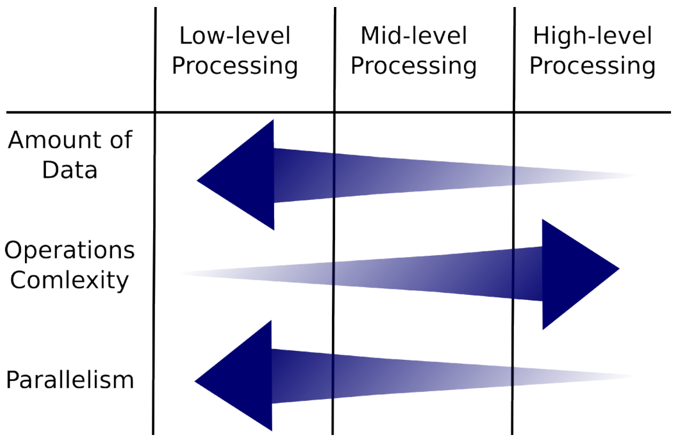


Figure 1.2: Processing Levels Specifications

the algorithm becomes bottlenecked on memory bandwidth. Computing the entire horizontal pass before the vertical pass destroys producer-consumer locality. In other words, horizontally intermediate values are computed before they are consumed by the vertical pass. This doubles the storage and memory bandwidth required. Exploiting locality requires interleaving the two stages by tiling and fusing the loops. Tiles must be carefully sized for alignment, and efficient fusion redundant computing values on the overlapping boundaries of intermediate tiles. At the end, these optimizations allow to accelerate the considered 3×3 box filter.

From this simple example, we notice that accelerating image processing algorithms is not a trivial task. In this work, we focus on the real aspect and requirements to embed vision-based ADAS. Obtaining a real time performance on embedded vision is a very challenging task, as there is no hardware architecture that meets perfectly the requirements of each processing level and kernel specifications. From the literature, we distinguish three different processing levels in computer vision applications: low-level, mid-level and high-level [7]. Figure 1.2 illustrates the specifications of each level in terms of parallelism, data amount and complexity.

In low-level processing, we find repetitive operations at pixel level which is then characterized by high level of parallelism. As an example, we may cite simple filters such as edge detectors (Sobel) and noise reduction (Gaussian). These processing are usually optimized through `SIMD` instructions. In the second processing level–mid-level–, operations are performed on certain region of interest which respond to particular criteria. This level includes operations such as features extraction, segmentation, object classification and optical flow. This level shows lower parallelism and higher complexity compared to filters. These operations can only be parallelized on high performance architectures such as many-core and GPUs. Finally, the last level includes decision-making operations where sequential processing is present the most of the time.

1.2.2 Hardware Platforms

As discussed previously, computer vision applications deal with an important amount of data requiring high computing performance. Also, a hardware product that is installed inside a vehicle must be embedded and needs to fulfill the requirements of embedded vision systems. Actually, there is no hardware architecture that meets perfectly all the requirements. This section gives an overview of available hardware.

FPGA A Field Programmable Gate Array (FPGA) is an IC designed to be configured by a customer after manufacturing. They have lower power consumption and they are better suited for low-level processing than general purpose hardware, where they clearly outperform them. However, they are not so good for the serial processing necessary in mid and high levels. As FPGAs are dynamically programmable, they can be reconfigured to different applications. For instance, a driver-assistance system might use different applications at day and night, or when driving through tunnels. Rather than have all the algorithms implemented in custom hardware (ASICs) at the same time, an FPGA-based system can select the most suitable algorithm, reconfigure, and continue processing.

GPU Graphics Processing Unit (GPU) was initially designed to accelerate the creation of images intended for output to a display and for image rendering tasks. Nowadays, GPUs are also used for general-purpose computing. GPUs have traditionally been considered as power consuming devices and they are not very frequent yet in vehicle applications. However, recent solutions such as the NVIDIA DRIVE PX platform based on the NVIDIA Tegra X1 SoC [8] are very promising.

DSP Digital Signal Processors (DSPs) have been the first choice in image processing applications. DSPs offer single cycle multiply and accumulation operations, in addition to parallel processing capabilities and integrated memory blocks. DSPs are very attractive for embedded automotive applications since they offer a good price to performance ratio. However, they require higher cost compared with other options such as FPGAs, and usually, they are not easy and fast to program.

Multi-core Systems Multi-core systems are usually appreciated by car manufactures and suppliers for many reasons. First, existing software of multiple conventional single-core processors can be executed in parallel without any software changes using an appropriate operating system. Second, multi-core systems are at the lower end of the price scale.

Heterogeneous Architectures Each of the previously discussed hardware solutions has some advantages and limitations. For instance, GPUs provide high performance computing but consume an important amount of energy. FPGAs consume less energy but need more time and knowledge to prototype, design and program an application. To cope with this issue and find a better compromise between computing performance, energy efficiency and cost, *heterogeneous* Systems on Chips (SoCs) have been introduced. Semiconductor manufacturers proposed to integrate two or more processing units with different technologies in the same chip. SoCs are usually cheaper and have higher reliability than multi-chip solutions. Recently there is a growing trend to use System on Chips in embedded vision. As examples, we can cite Texas Instruments (TI) TDAx SoCs [9] (ARM, DSP, EVE) and NVIDIA Tegra X1 [10] (ARM, GPU).

1.2.3 Software Tools

To address vision-based ADAS challenges in embedded systems, several approaches have been presented that introduce an additional layer of abstraction between the developer and the actual target hardware. These approaches aim to cope with the high computation needs and energy efficiency demand of image processing algorithms particularly in terms of real time requirements.

As first approaches, we find extensions to `C` programming language and libraries or `APIs`. These approaches allow to keep the original `C` code. Parallelization is performed by adding for instance directives to the `for` loops. `OpenMP` [11] and `OpenACC` [12] optimize image processing algorithms via the compiler at the loop level through directives. Both `OpenMP` and `OpenACC` can be used on CPUs and GPUs. `OpenACC` has been initially developed to target different hardwares. However, `OpenMP` was only supported on CPU-based systems such as multi-core architectures for shared memory parallelization. GPU support is available from `OpenMP.4.0` release.

Relevant parallel languages have been proposed in graphics such as `CUDA` [13] and `OpenCL` [14]. These languages employ single program multiple data programming model. `CUDA` can only be used on NVIDIA GPUs. `OpenCL` is a well known programming environment for both many-core accelerators and GPUs. It is supported by some heterogeneous architectures like Altera FPGA architecture [15]. Like `C`, they allow the specification of very high performance implementations for many algorithms. But because parallel work distribution, synchronization, kernel fusion, and memory are all explicitly managed by the programmer, complex algorithms are often not adapted to these languages, and the optimizations required are often specific to an architecture, so code must be rewritten for different platforms.

The previous discussed programming languages and libraries are general purpose techniques. However, some domain-specific languages (DSL) for image processing applications exist such as `Halide` [16]. `Halide` is a programming language for image processing algorithms targeting different architectures including x86/SSE, ARM v7/NEON, GPU(CUDA, OpenCL) and Native Client. `Halide` model is based on stencil pipelines for better locality and parallelism. We find also Numerical Template Toolbox `NT2`. It is a `C++` library-based DSL. It uses generative programming idioms so that architecture features become mere parameters of the code generation process.

There are some compilers which allow *automatic parallelization*. These compilers examine the `for` loops to automatically decide sections that can be run in parallel, as well as how many threads to use. As an example, Intel `C/C++` compiler icc does this when the option `-parallel` is enabled. At a glance, the automatic parallelization compiler seems to be the best solution since it does not require the user to do anything, but in reality, as the code becomes more complex, the compiler has difficulty finding what can be parallelized, making the performance suffer. To our best knowledge nowadays (July 2017), no existing compiler (at least no commercial) can auto-generate parallel code for **heterogeneous** systems such as the accelerator.

1.3 Motivations

Embedding vision-based ADAS applications raised important scientific challenges which were behind the birth of this work. As discussed previously, vision-based ADAS require high performance computing in real time. This led to the design and development of new architectures and new programming languages. Both proposed hardwares and softwares have different features and no one is perfect of a specific application. For instance, GPUs respond to the performance computing requirement of image processing algorithms, but they are considered as power consuming devices. In the other hand, FPGAs consume less power but require more time to design and develop an application based on hardware description languages such as `VHDL`. From software point of view, high-level techniques such as compiler directives (`OpenMP, OpenACC`) are easy to use and no code rewriting is required anymore. However, there are some functionalities which are not managed such as shared and texture memory in GPUs. `CUDA` which is a low-level language support more features and allows the programmer to better optimize the code for instance at different memory levels. However, with `CUDA`, code rewriting is required.

As a general idea, it is not evident to find the best architecture and parallel language which will respect our constraints particularly in terms of real time requirements. Indeed, things get more complicated in heterogeneous architectures where different computing technologies are employed which may then need different programming techniques.

Figure 1.3 illustrates the development process of embedded ADAS as well as the corresponding challenges. Globally, embedding an ADAS application requires **three** major steps. First, after the algorithm specifications are set, the algorithm is developed on a standard fixed platform (PC). The algorithm is then validated at functionality level. Second, before moving to embedded platforms, some analysis of the code is required to determine the bottlenecks–most time consuming functions– and the algorithm is decoupled to small blocks referred as **functions** or **kernels**. Third, porting process can be started by focusing on the previously identified bottlenecks.

During the whole process, the developer meets a set of challenges as illustrated in Figure 1.3 (blue bubbles) :

- **Efficient migration PC $\rightarrow$ embedded systems** While this seems to be straightforward, it is not that evident. With small codes and less algorithm complexity,

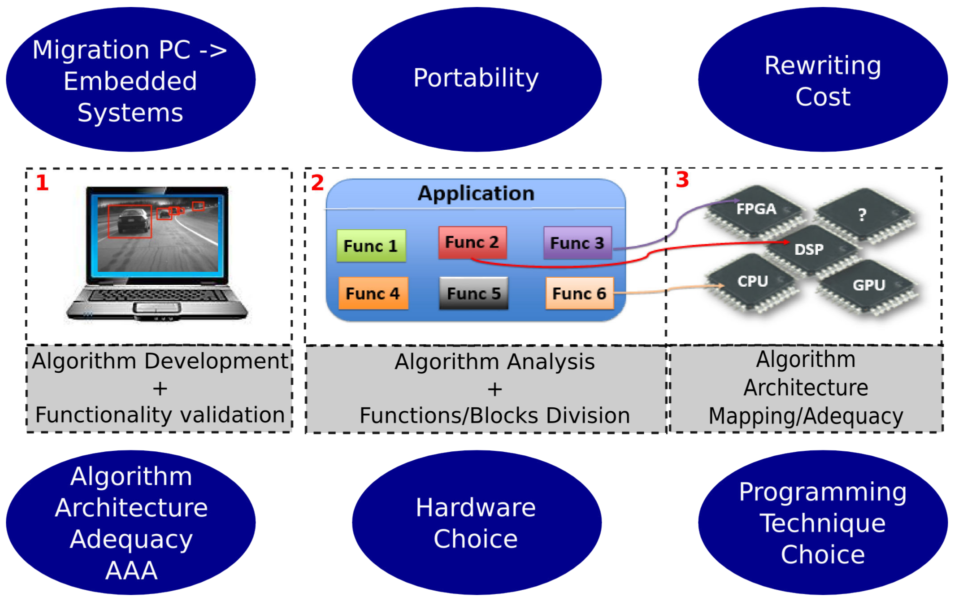

Migration PC -> Embedded Systems
Portability
Rewriting Cost
1
2
Application
Func 1
Func 2
Func 3
Func 4
Func 5
Func 6
3
FPGA
?
DSP
CPU
GPU
Algorithm Development + Functionality validation
Algorithm Analysis + Functions/Blocks Division
Algorithm Architecture Mapping/Adequacy
Algorithm Architecture Adequacy AAA
Hardware Choice
Programming Technique Choice

Figure 1.3: Embedded ADAS Development Process and Challenges.

developers may port the whole algorithm at once. However, with more complicated algorithm, sometimes it is better to make *progressive* migration. In other words, we port one block (function, kernel) at once. This is a good approach for fast debugging.

- **Portability** If different architectures are targeted, then the portability becomes a crucial point. Since most of the high-level languages such as `CUDA` can only be applied to NVIDIA GPUs.

- **Rewriting cost** This is related to portability. If we use different architectures, we may need different programming languages. Hence, rewriting is required which is a time consuming process which has also a cost at economy level for companies.

- **Parallel programming technique(s)' selection** As we have seen previously, different parallel programming techniques exist. Each one has its advantages and limitations. The developer needs sometimes to write the algorithm with different techniques to select the best approach at the end.

- **Hardware selection** At hardware level, there are also a wide choice. Some architectures are designed for energy efficiency purpose as FPGA. Others provides high performance computing as FPGAs.

- **Algorithm Architecture Adequacy (AAA)** Embedding image processing algorithms may require some adaptations less or more important at algorithmic level to achieve the requirements. A compromise between algorithm quality and speedup is necessary on embedded vision-based applications.

1.4 Research Goals and Contributions

In this thesis, the previously presented challenges are targeted. We focus mainly on the portability issue of vision-based ADAS applications. Portability is related to parallel programming techniques in one hand and to the choice of the hardware in another hand.

In this work, we first propose an algorithm for vehicles detection based on stereo vision [17]. The algorithm is based on disparity map segmentation and objects classifications. It is worth noting that this algorithm has not been developed to compete or to give better performance compared to the existing works in the literature. It has been rather developed to be used as a use-case for our experiments on embedding vision-based algorithms. This algorithm has two important features:

1. *High performance computing* To target the previously discussed challenges, we need to have a time consuming algorithm which needs to be processed in real-time in embedded systems. For this reason, we selected the stereo vision as perception system. Stereo matching is then performed to generate a disparity map. Stereo matching is one of the most studied problems in computer vision. For decades, many stereo matching algorithms have been proposed with new improvements in both matching accuracy and and algorithm efficiency. Most of the proposed works tend to be contradictory in reported results: *accurate stereo matching methods are usually time consuming.* For this reason, we selected to include stereo matching in our ADAS application.

2. *Various kernels specifications* When parallelizing image processing algorithms, optimizations may differ from one kernel to another. This difference comes from kernels' specifications. By specifications we mean for instance data access pattern, data flow, data dependency, etc. Each feature requires a specific optimization. As an example, let's take Sobel filter. From data access pattern point of view, it is a local neighboring operator. To compute each output pixel, we need the

neighboring pixels of the corresponding input pixel. This feature implies to focus on memory accesses and cache locality for better performance. We developed the algorithm in such a way to have a variety of kernels with different features. This allows us to perform different experiments and test various optimizations.

Once the algorithm is developed and validated, we move to the core of this thesis. We start the process of embedding the algorithm. We target different architectures available in the market an we use various parallel programming methodologies. We selected multi-core systems (CPU-based) and NVIDIA platforms; Jetson k1 and Tegra X1 (GPU-based). The choice is based on two important criteria. First, we selected the most available and employed platforms in automobile industry nowadays such NVIDIA platforms. Second, we focus on the stability and the performance of the hardware in one hand and the maturity of the corresponding software in the other hand. For each employed architecture, we introduce its relevant features and then describes their effects on the applied algorithm and the parallelization approaches.

For image processing, the global organization of execution and storage is critical. Image processing pipelines are both wide and deep: they consist of many data-parallel stages that benefit hugely from parallel execution across pixels. However, stages are often memory bandwidth limited; they do little work per load and store. Gains in speed therefore come not just from optimizing the inner loops, but also from global program transformations such as tiling and fusion that exploit producer-consumer locality down the pipeline. In other words, optimizations of vision-based applications fall in two major axes: *kernel-level* and *system-level*. By kernel-level optimizations we mean all optimizations that we apply on kernels or small functions. In this case, we usually use traditional parallel languages such as `OpenMP, OpenCL, CUDA`, etc. With these tools, we only optimize parts of the algorithm. In the second axis–system-level–, we aim to optimize the algorithm as a whole system. This is performed by focusing for instance on data transfers and allocations, memory bandwidth and inter-processor communications. Usually these optimizations are managed through frameworks such as `OpenVX`.

In this work we target both axes. In the first part, we employ and investigate kernel-level optimizations. We use different parallel techniques such as vectorization and shared memory parallelization. We use also various programming approaches starting from high-level techniques such compiler directives (`OpenMP, OpenACC`) to low-level approaches (`CUDA`). We test different optimizations to accelerate the algorithm on the selected architectures. The best choice of optimization is architecture-specific; implementations optimized for an x86 multicore and for a modern GPU often bear little resemblance to each other. Each optimization is tested with the different programming techniques for evaluating the obtained results at the end. We present the advantages and limitations of each technique. We evaluate the employed approaches in terms of obtained performances and programming productivity.

In the second part, we target the second axis; system-level optimizations. We use `OpenVX` framework which allows us to apply some system-level optimizations such as *kernels fusion* and *data tiling*. We investigate the different proposed optimizations [18].

In the third part, we propose an approach to target both kernel-level and system-level optimizations [19]. To our best knowledge, this approach has not yet proposed in the literature. We propose to embed customized computer vision kernels on `OpenVX` framework. We use different techniques to accelerate kernels and then embed them in `OpenVX` to benefit from its system-level optimizations simultaneously.

The feedback of this research is crucial for the development of future image processing applications in adequacy with parallel architectures with a best compromise between computing performance and algorithm accuracy.

1.5 Thesis Organization

Chapter 2 refers to related works regarding ADAS vision-based acceleration on embedded systems. We enumerate the different programming techniques proposed and employed in the literature.

In chapter 3, we present the use-case we developed and employed in this thesis to perform our experiments. It is an ADAS application which performs vehicles detection based on stereo vision. We present the algorithm in details. A comparison with the state of the art is also provided at the end.

The first part of chapter 4 concentrates on analyzing the algorithm from the execution time point view. We identify the bottlenecks; those functions or part of the code which are the most time consuming in the whole algorithm. After that, we propose to adapt the application at algorithmic level to respond to some features required in parallel computing such regular access pattern. In the second part, we give the first results of accelerating the algorithm on CPU-based approaches at kernel-level. Different programming techniques are employed which perform either vectorization or shared memory parallelization. At the end of the chapter, we evaluate the obtained results and the contribution as well as the limitation of each programming technique.

In chapter 5, we propose to accelerate the algorithm on GPU-based systems at kernel-level. We propose different optimizations from high priority to low priority according to the complexity of GPUs architectures and memory hierarchy. We use two different approach, a low-level and a high-level technique. We discuss the obtained results, the issues encountered as well as the performance obtained. At the end, we compare the two approaches from different aspects: performance, productivity, limitations, etc.

In chapters 6, we start by investigating the `OpenVX` framework. We test some system-level techniques such as *kernels fusion* and *data tiling*. We show the relevant features and limitations of each technique. In the second part, we propose a novel approach to accelerate vision kernels at both kernel- and system-level. We propose to integrate accelerated kernels on `OpenVX` framework to benefit at the same time from the system-level optimizations of this framework. We test the approach on different architectures.

The thesis concludes with a summary of the different contributions and an outlook on potential perspectives.

2

State of the Art

Recently Computer Vision (CV) applications have rapidly emerged in the field of autonomous driving. Both algorithm accuracy and execution time are important metrics in designing real-time CV algorithms. Existing CV algorithms usually perform well in one aspect but not good enough in the other. Indeed, researchers and developers focus either on improving accuracy or on code optimization of an existing algorithm. There are two main ways to accelerate the algorithm. The first approach consist in optimizing at algorithmic level by using specific algorithm. In the second approach, we rely on *hardware capabilities* of some dedicated high performance architectures such as FPGAs and GPUs. In this case, we exploit the performance of these architectures through software optimizations by using specific programming languages or parallel techniques to accelerate the algorithm.

Developing CV algorithms for embedded systems is often severely constrained by the computation requirements and hardware resources of the corresponding systems, as well as the real-time operating conditions. Developers must be able to optimize the performance of their applications within the constraints imposed to the systems. Performance metrics in terms of data processing throughput and accuracy have to be balanced with other optimization objectives, such as code/data size, memory bandwidth, latency, and power consumption. It is a challenging task for embedded developers to be able to map a CV algorithm derived from theoretical research to performance-optimized software that is running in real time on an embedded platform. The author in [20] gives an overview of the different challenges we are faced to when using high performances embedded systems.

2.1 Parallelism Fundamentals

Parallel processing is applied to accelerate computation by sharing the workload among multiple processors. Scientific applications are typically subdivided into two major classes; *regular* and *irregular* applications. In regular applications, the data structures used are dense arrays and the accesses to these data structures can be characterized well at compile time. In irregular applications, some of the data structures used may be sparse arrays whose structure can only be determined at the time of program execution.

In terms of algorithm design, parallel computing strategies profits from the natural parallelism present in the algorithm which provides two main resources of parallelism:

- *Data parallelism* is the simultaneous execution on multiple cores of the same function across the elements of a dataset.
- *Task parallelism* is the simultaneous execution on multiple cores of many different functions across the same or different datasets.

In image processing and CV algorithms, task parallelism may exist, however, it is data parallelism that can be most efficiently exploited for the following reasons. Firstly, comparing to task parallelism, data parallelism is present more often in image processing algorithms. Secondly, even if task parallelism exists, it only offers limited opportunities for parallelization and speed improvement. Also, the significant increase of image resolution increases the computational requirements of most image processing algorithms rapidly. These high performance requirements can only be compensated by exploiting data parallelism. Consequently, in this thesis, we will focus on exploiting data parallelisms in CV algorithms.

2.2 Embedded Platforms for Vision-based ADAS

Achieving optimal results in terms of speed and accuracy depends both on the algorithm and the hardware platform. Usually, real-time image processing algorithms with low errors compared to the ground truth are characterized by heavy computations. One promising direction to achieve real-time processing in high performance computing applications such as image processing would be to exploit the computing performance and parallelism in some dedicated architectures.

Nowadays, there are different types of parallel computers which have been built or proposed. These architectures can be classified differently based on several aspects. One important feature is whether or not the parallel machine has a *single shared memory.* In a shared memory machine, all of the processors read from and write to a single common memory. This memory is connected to the processors through an interconnection network. In a non-shared memory computer, we rather find processors which have private local memories, and an interconnection network is used for communication. Usually, the shared memory parallel machines are easier to program since the user does not care about or perform any communication operation. However, the non-shared memory parallel computers are generally more efficient because local data accesses do not go through an interconnection network. Also, the programmer performs communication operations in the most efficient manner possible.

Another important feature to classify parallel machines is whether it operates in a Single Instruction Multiple Data Stream (SIMD) or Multiple Instruction Multiple Data Stream (MIMD) mode. In MIMD mode, the machine has the ability to process different instructions on different data simultaneously. In other words, the cores can process multiple tasks at the same time, and the system is designed to exploit task parallelism. Each core is considered as a fully functional single core processor. Hence, they are completely compatible with sequential programs for single core processors. In the other hand, the SIMD architecture aims to exploit only the data parallelism by executing the same instruction on the multiple data streams simultaneously. Since all the cores or the Processing Elements (PEs) share the same instruction at the same time, SIMD processors have significantly lower hardware complexity than MIMD processors. Hence, it is easily possible to find more PEs in SIMD processors that in MIMD processors. This results in a significant speed improvement when exploiting data parallelism with SIMD. However, the greater data-parallelism the SIMD processors are designed to exploit, the more hardware limitations the processors will have. As a result, processors designed to utilize the data parallelism usually are incompatible with programs or algorithms designed for sequential processors.

2.2.1 Multi-cores

One of the most important innovation in computer engineering has been the development of *multi-core* processors. They are composed of two or more cores in a single physical package. Today, many processors, including Digital Signal Processors (DSPs) and Graphical Processing Units (GPUs) ([21]) have a multi-core design, driven by the demand of higher performance. CPU vendors have changed strategy away from increasing the raw clock rate to adding on-chip support for multi-threading by increasing the number of cores; dual-, quad- and many-core are now commonplace. Signal and image processing developers can benefit dramatically from these advances in hardware, by modifying single-threaded code to exploit parallelism to run on multiple cores. Also, one important reason of using multi-core machines is the presence of a shared memory. This decreases significantly the cost of data transfers and inter-cores communication. However, a coherence protocol is required to ensure that the data read by a processor is consistent. It also provides a set of rules to keep the data in the cache of a processor consistent. Another reason to use multi-core processors is the programming facility at software-level based on some extensions to traditional programming languages such as `C`. This allows the programmer to benefit from the computing performances of multi-core processors with less programming efforts. These extensions will be discussed in the next section.

Different works have been proposed in the literature to implement ADAS applications on CPU-based multi-core systems. In [22], the authors presented a vision-based vehicle detection and tracking pipeline. An approach based on smartphone cameras is employed which supposes a versatile solution and an alternative to other expensive and complex sensors on the vehicle, such as LiDAR and radars. The algorithm runs at 7.6 fps (frames per second) on iPhone 5, 13.2 fps on iPhone 6 and 62.5 fps on a simulator with Intel i7 processor. The authors in [23] presented an implementation of a traffic sign detection system on a multi-core processor. The whole system can run at 25 fps on the multi-core processor, where 10 worker threads are used. In [24], a pedestrian detection algorithm was implemented on multi-core work station with Intel processor. The experimental results showed a speedup ranging from 2 to 4.

2.2.2 GPUs

GPUs were initially developed to perform graphical calculations. In 2006, most GPUs were designed based on a graphic pipeline model. The inputs were geometric primitives which are processed through several stages such as vertex operations and composition to give a final image [25]. Each element of this pipeline is processed by a separate hardware computing unit and multiple copies of the pipeline were printed on a GPU. This allows the execution of thousands of operations simultaneously to produce images as outputs in the screen.

The success of GPUs in graphics led to extend their application to more general purpose applications. General Purpose GPUs (GPGPUs) are considered as a more generalized hardware which makes them suitable for a wide range of algorithms. They are increasingly used in scientific supercomputers and other high performance applications [26]. As applications in general require more and more computation capabilities, GPUs are increasingly implemented in mobile phones, tablets and other low cost devices.

Since 2010/2011, GPUs are used in vehicles produced by BMW and AUDI [27]. At the beginning, these applications mostly serve in entertainment systems, but recently they are used for ADAS such as in the lane departure warning system. The manufacturer NVIDIA provide GPU-based embedded platforms with CPU ARM as host processor for applications in automotive applications [10], [28]. Today there are more than 4.5 million cars on the road powered by NVIDIA processors, including the newest models from Audi, BMW, Tesla Motors and Volkswagen [28], [29].

The literature is rich in terms of vision-based ADAS applications on GPUs. In [30], the authors proposed a real-time pedestrian detection system implemented on NVIDIA Tegra X1 GPU-CPU hybrid platform. The experiments show that the algorithm can execute 20 images of 1242×375 pixels per second, and the GPU-acceleration provides between 8x and 60x performance speedup with respect to a baseline CPU implementation. The authors in [31] proposed a vehicles detection and trajectory estimation in multiple lanes. Algorithm computation are distributed between CPU and GPU with multi-thread capabilities. An implementation of a lane detection algorithm on NVIDIA Jetson K1 (GPU+CPU) is proposed in [32]. Different approaches of lane detections have been tested on different configurations; CPU(ARM), GPU and CPU+GPU.

2.2.3 FPGAs

Field Programmable Gate Arrays (FPGAs) are configurable devices with a set of hardware resources such as logic blocks, that can be configured in different manners to perform various functionalities. Unlike CPUs or GPUs, where functionality and routing of hardware blocks is irreversible after the manufacturing stage, FPGAs can change their hardware layout at run time. Desired layouts are stored on a non-volatile cache memory and can be loaded whenever it is necessary. Programs written for a CPU are compiled to fit the hardware layout. In FPGAs, however, the hardware is routed to fit a program's requirements. This allows a very efficient and dynamic combination of both parallel and serial computation parts.

The study of the state of the art shows various works which have been conducted in implementing ADAS applications on FPGAs. In [33], the authors reviewed several pedestrian detection algorithms and discussed issues related to implementing these algorithms on an FPGA platform. An FPGA-based implementation of an automatic traffic surveillance sensor network is proposed in [34]. The algorithm extracts moving vehicles from real-time camera images for the evaluation of traffic parameters, such as the number of vehicles and their direction of movement. The obtained results show a processing frame speed of 117 fps. In [35], a vision-based lane departure warning system and its implementation on an FPGA device are presented. The algorithm runs at 40 fps on 748×260 images.

2.2.4 ASICs

Application Specific Integrated Circuits (ASIC) are Integrated Circuits (IC) designed for a particular applications, rather than for general purpose use. Designers of ASICs usually use a hardware description language such as `Verilog` or `VHDL` to describe the functionality of ASICs. ASICs usually have high performance and low power consumption. However, ASICs are not reconfigurable. Hence, once they are manufactured, they cannot be reprogrammed. This lack of flexibility has led to the use of other reprogrammed hardwares such as Field Programmable Gate Arrays (FPGAs). Despite this problem of flexibility, we can find some examples of ADAS implementations on ASICs in the literature [36], [37]. Also, ASICs have been used by Mobileye to build its first ADAS product EyeQ [38] and then improve it with more powerful hardware and processing units through the EyeQ series [39].

2.2.5 Heterogeneous Architectures

The previously presented hardwares mainly GPUs, FPGAs and multi-cores can be classified as geleral purpose multi-core systems. These systems tend to consist of multiple conventional processors on a single chip, thus providing a homogeneous processing platform. However, modern image processing algorithms contain several individual tasks with different granularities and different complexities. This implies the need to use different computing technologies within the same application to achieve the requirements mainly in terms of real-time. Actually, each hardware technology has its advantages and limitations.

Let's compare FPGA to GPU. From development complexity point of view, FPGA boards usually rely on synthesizable VHDL models. An important number of concurrent VHDL statements and small processes connected through signals are used to implement the desired functionality. Reading, understanding data flow VHDL code is not an easy task. Actually, the concurrent statements and processes do not execute in the order they are written, but when any of their input signals change value. GPUs have issues related to compatibility and portability, for instance, NVIDIA GPUs support only CUDA programming language. Actually, performance improvement could be highly variable depending on the type of graphic and employed APIs, e.g. CUDA or OpenCL. Moreover, not all tasks are suited for running on a GPU such as I/O and memory bound operations. Therefore, different architectures specialized for specific image processing tasks should be combined in order to achieve a high performance. These led to the development of *heterogeneous* architectures. These systems consist of two or more computing units with different computing technologies. Recently there is a growing trend to use heterogeneous systems in embedded vision. As examples, we find Texas Instruments (TI) TDAx SoCs [9] (ARM, DSP, EVE), NVIDIA TegraX1 [10] (ARM, GPU), NVIDIA Drive PX [8] and the R-car SoCs of Renesas [40].

In the literature, several works have been presented in implementing vision-based ADAS on heterogeneous architectures. In [41], the authors proposed a pedestrian and vehicle detection implementation on both GPU and CPU. The application runs on 640×480 images with a processing speed of 23.8 fps (42 ms) and detects both pedestrians and vehicles. A lane departure warning implementation on a hybrid CPU-FPGA system is proposed in [42]. The algorithm runs at 25 fps on 256×256 images.

2.3 Parallel Software Tools for Embedded Platforms

Computing hardware continues to move toward more parallelism and more heterogeneity to provide higher computing performance. Nowadays, we find several levels of parallelism expressed by the interconnections of multi-core and many-core accelerators. To

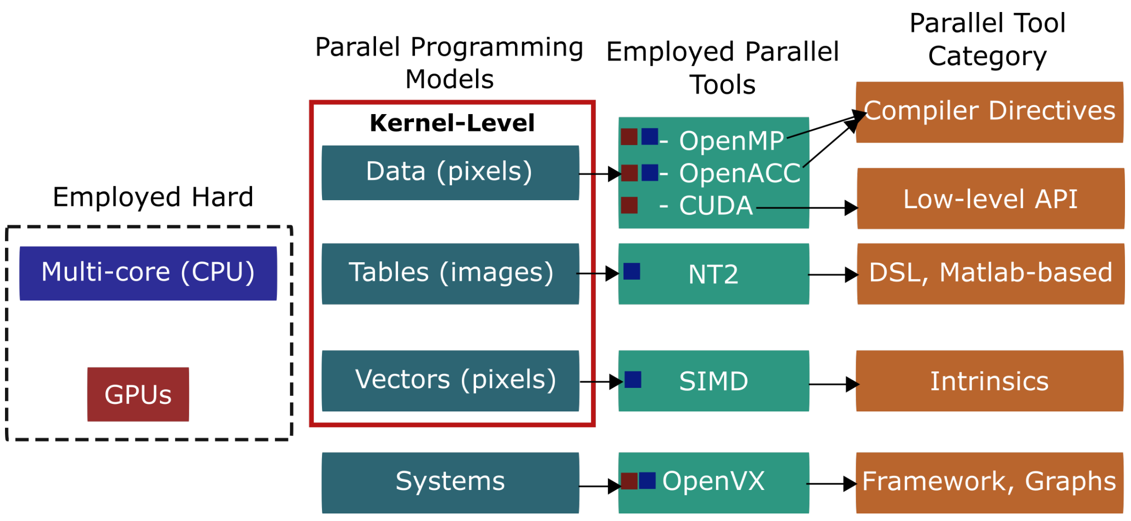


Figure 2.1: Parallel Programming Models

follow this trend in hardwares, different programming tools have been developed that introduce an additional layer of abstraction between the developer and the actual target hardware. These approaches aim to cope with the high computation requirements of some applications such as image processing algorithms to reach real-time conditions.

The available computing softwares rely on different parallel programming models which can be classified into several categories from different aspects. Figure 2.1 depicts some parallel programming tools and their corresponding classification. It is worth noting that this figure shows mainly the employed tools and architectures on this work.

First, there are some tools which provide parallelization at **kernel-level** while others at **system-level**. **Kernel-level** optimizations aim to optimize parts of the algorithm such as functions. In other words, these optimizations are applied only at low level referred as *kernel-level*, not on the complete algorithm. **System-level** optimizations in the other hand aim to optimize the algorithm as a whole system such as optimizing the inter-processors communications, data transfers and memory bandwidth.

Second, Each parallel tool work on special level of data. Some tools aim to describe what happens at pixels level. In other words, how each pixel is treated to express parallelism. It is also possible to optimize at table levels. For instance, we describe how a whole image is parallelized. Finally, some tools provide an approach to work on a set of data as vectors for instance to apply vectorization.

Finally, the parallel tools can be classified according to the approach employed to express the parallelism. The **first** category includes *directive*-based *high-level* tools. The programmer keeps the original `C/C++` code and annotates explicitly this sequential code to specify the parallel regions. The compiler than applies the appropriate parallelization at compile time. In the **second** category, we find *low-level libraries* which are usually based on *data-parallelism*. In this case, the programmer relies on some libraries to write the parallel code and specify explicitly the regions to be parallelized. The sequential code is first analyzed at low-level to identify the parallel regions where repetitive operations are present. In the **third** category, special instruction sets are employed to apply vectorization referred as *intrinsics* such as Intel `SSE` and ARM `NEON`. In the **last** category, we find those tools which aim to apply more global optimizations. They are usually implemented through *frameworks* and *DSLs* such as `OpenVX` and `NT2`.

In the following sections, we give more details and examples of the different tools available in each category. We focus more on the tools applied in this research work.

2.3.1 Directive-based Tools

On the top of the list, we find extensions to `C` programming language and libraries or `APIs`. These approaches allow to keep the original `C` code. Parallelization is performed by adding directives to the `for` loops. `OpenMP` [11] and `OpenACC` [12] are the most employed tools in this categories. They optimize image processing algorithms via the compiler at the loop level through directives. While `OpenMP` has been first developed for multi-core systems, `OpenACC` has been rather proposed to target accelerators such as GPUs. Starting from release 4.0 [43], `OpenMP` supports parallel programming on accelerators such GPUs.

2.3.1.1 OpenMP

OpenMP Programming Model `OpenMP` API provides of a set of compiler directives and runtime library routines for parallel applications programmers. The main goal of `OpenMP` model to to greatly simplifies writing multi-threaded programs in `Fortran` and `C/C++`, it is a higher-level approach compared to `Pthreads`. In `C/C++`, directives implemented as `#pragma omp` are employed. Hence, the same sequential source code can be maintained and parallelized by adding few directives. Listing 2.1 shows a `C++` code source of a matrix-vector multiplication. This example exhibits a simple shared memory parallelism since the same instruction is applied to all elements. Listing 2.2

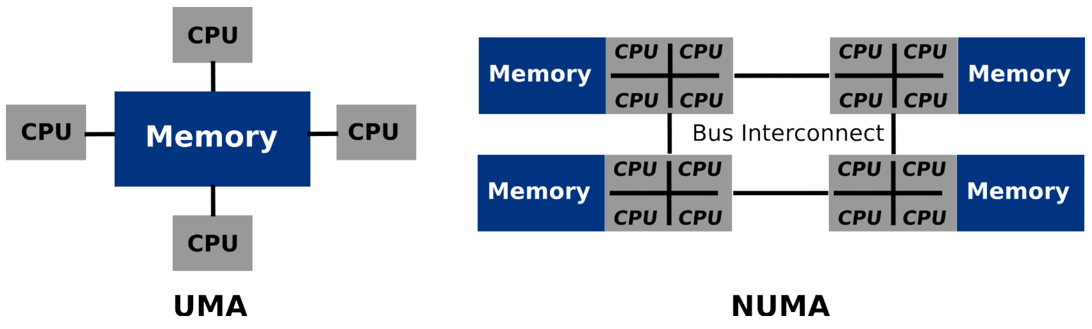


Figure 2.2: UMA and NUMA Memory Models.

shows the `OpenMP` version. The only difference between the two codes is the `OpenMP` directives added before the `for` loop (lines 3 and 6, Listing 2.2).

```
void MatVecMult(int n, int m, double *restrict a, double *restrict b, double *restrict c)
{
  for (int i=0; i<n; i++) {
    c[i] = 0.0;
    for (int j=0; j<m; j++)
      c[i] += b[i*m+j] * a[j];
  }
}
```

Listing 2.1: C++ Matrix-Vector Multiplication Source Code.

```
 1 void matVecMul_OpenMP(int n, int m, double *restrict a, double *restrict b, double *restrict c)
 2 {
 3   #pragma omp parallel for
 4   for (int i=0; i<n; i++) {
 5     c[i] = 0.0;
 6     #pragma omp parallel for
 7     for (int j=0; j<m; j++)
 8       c[i] += b[i*m+j] * a[j];
 9   }
10 }
```

Listing 2.2: OpenMP Matrix-Vector Multiplication Source Code.

OpenMP Memory Model OpenMP supports multi-core shared memory machines. The architecture of these machines can be shared memory Unified Memory Access (UMA) or Non UNiform Memory Access (NUMA) as shown in Figure 2.2. In our experiments, the UMA model is used in many-core machines. OpenMP assumes that there is a one memory space for storing and retrieving data that is available to all threads. This space is then *shared* among all threads. However, it is possible for a thread to have a temporary view of some memory space or data which is cannot be seen by other threads. This is the case with *private* variables for example. Each variable used within a parallel region can be either *shared* or *private*. Shared variable references inside the parallel construct refer to the original variables of the same name. For each private variable, a reference to the variable name inside the parallel construct refers to a variable of the same type and size as the original variable, but private to the thread, i.e., it is not accessible by other threads.

2.3.1.2 OpenACC

Presented in 2011, `OpenACC` has a directive-based approach to programming accelerators. This standard has been developed by a group of hardware and compiler vendors such as NVIDIA, PGI, Cray and CAPS. `OpenACC` provides a simple interface to programmers, offering a trade-off between performance and development effort. Generally, the compiler is responsible for generating an efficient code to take advantage of the hardware. Static compiler passes can figure out particular data or arrays with an opportunity for parallelization. As static passes and compiler optimizations are limited, `OpenACC API` also provides a set of directives that can be used to write parallel program on `Fortran`, `C/C++` to run on accelerators such as GPUs.

The main purpose behind developing `OpenACC` is to propose a single model while targeting different platforms as illustrated in Figure 2.3. `OpenACC` was initially developed by Portland Groop (PG) and supports multi-core systems and NVIDIA/CAPS GPUs. In multi-core systems, no data copy or transfer is required since we work on the same processor; the host. PGI compilers on Linux, Windows, and Mac OS X support OpenACC for multicore. It works with any supported PGI target. This feature will work with any valid PGI license.

OpenACC Programming Model on CPU `OpenACC` programming model in multicore systems is similar to `OpenMP` one. It relies on shared memory model. Directives are employed to give hints to the compiler; with `OpenMP` we use `#pragma omp` directives , while with `OpenACC`, we use `#pragma acc` directives. The multicore CPU is treated like an accelerator device that shares memory with the initial host thread. With a shared-memory device, most of the OpenACC data clauses (copy, copyin, copyout, create) are ignored, and the accelerator device (the parallel multicore) uses the same data as the

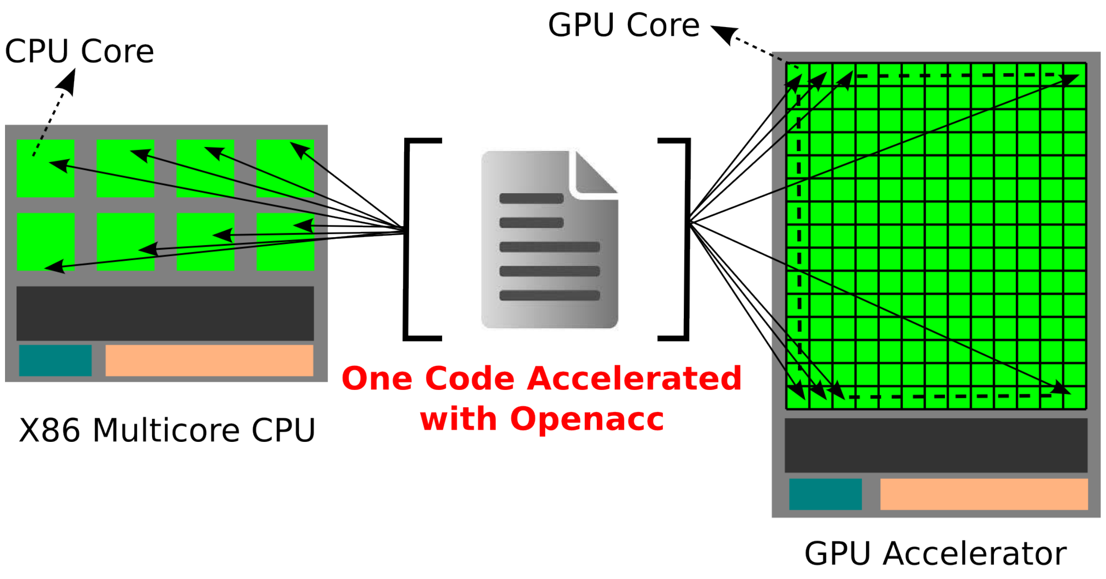


Figure 2.3: CPU and GPU Model of OpenACC.

initial host thread. Similarly, update directives and most OpenACC data API routines will not generate data allocation or movement.

OpenACC Programming Model on GPU `OpenACC` and `CUDA` have the same execution model on GPU, since in both cases, the host (CPU) manages the execution of kernels in the device (GPU). Usually, kernels affected by parallelization consists of one or several nested loops. Current `OpenACC` compilers can handle loops which are nested up to *three* levels deep referred to *gang, worker* and *vector* (Figure 2.4). In NVIDIA `GPUs`, *gang* represents a *block*, *worker* refers to a *warp* of threads and *vector* is equivalent to `CUDA thread`. The compiler decides on how to map these constructs based mainly on the hardware and the device capabilities.

OpenACC Directives `OpenACC` directives are identified from the string `#pragma acc` just like an `OpenMP` directive which can be identified with `#pragma omp` string. `OpenACC` provides two directives to create a parallel region, `parallel` construct and `kernel` directive. While the formal one is used for *loop-sharing* loops, the later one is rather used to generated different kernels when there are several loops. `OpenACC parallel` directive works as `OpenMP` directive and it allows more explicit user-defined parallelism.

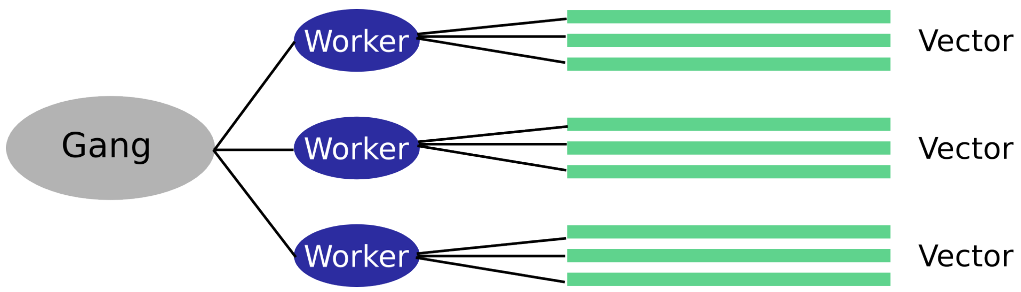


Figure 2.4: OpenACC Programming Granularity

```
void matVecMul_OpenACC(int n, int m, double *restrict a, double *restrict b, double *restrict c)
{
//100 thread blocks, each with 128 threads, each thread executes one iteration of the loop
#pragma acc parallel num_gangs(100) vector_length(128)
  #pragma acc loop gang vector
  for (int i=0; i<n; i++) {
    c[i] = 0.0;
    for (int j=0; j<m; j++)
      c[i] += b[i*m+j] * a[j];
  }
}
```

Listing 2.3: OpenACC Matrix-Vector Multiplication Source Code with Parallel Directive.

In `parallel` directive, the compiler generates only one kernel which will be executed at the launch time with a fixed number of gangs and vectors. Listing 2.3 gives and example of a matrix-vector multiplication with `OpenACC`. It also illustrates how to use `parallel` clause. We show how to set the `CUDA` grid in `OpenACC` through `num_gangs` and `vector_length`. In contrast to `parallel` construct, `kernel` directive may generates several kernels with different number of gangs and works.

It is worthy to note that `OpenACC` does not allow synchronization or data sharing between gangs just like `CUDA` and `OpenCL`. Data is only shared among workers within the same gang. This assumption is to maintain *scalability* on a large number of processing elements (PE) that run in parallel and can efficiently perform vector-like vectorization.

OpenACC Memory Model All details concerning data transfer between the host and the device memory and temporary data storage are handled by the compiler and the runtime environment. Hence, all these initialization tasks are completely *transparent* to the programmer. Data transfers between host and device memory spaces can be managed implicitly by the compiler based on some OpenACC directives inserted by the programmer in the code. Device data environment is used through the #pragma acc data directive followed by specific clause as copyin for data copy from host to device and copyout for data movement from device to host.

2.3.2 Low-level APIs

On the top of the most employed tools for parallel computing we find CUDA and OpenCL. While CUDA can only be-used on NVIDIA GPUs, OpenCL can be used on multi-core and FPGAs in addition to GPUs. Both tools are C-based low-level approaches which require more development efforts compared to directive-based techniques.

2.3.2.1 CUDA

In 2007, NVIDIA introduced a new parallel computing platform called CUDA (Compute Unified Device Architecture) which enables GPU usage for general-purpose computations. CUDA provides top level APIs to facilitate program development of GPGPUs while using massive parallelization powers of GPUs. CUDA is available only on NVIDIA cards which restricts developers to NVIDIA hardware. However, programmers can download it into a CPU-based system to compile code in GPU emulation mode. CUDA allows modified C/C++ code to specify part of the code within source files to be executed on GPU.

CUDA Programming Model In CUDA programming model, the system consists of a traditional CPU called *host*, and one or more massively data-parallel co-processing compute unites referred as *devices*. CUDA programs are considered as *hybrid* since they consist of multiple phases that are executed on either the host or a compute device such GPU. When a program or part of it exhibits low or no data parallelism, it is compiled with the host's standard C compiler such as gcc and runs as an ordinary sequential process, hence, it is executed in the host (CPU). A program with high data parallelism is implemented in the device (GPU) code. It is written using C programming language extended with keywords to label parallel regions called *kernels*. Host code uses a CUDA-specific function-call syntax to invoke kernel code.

CUDA model is based on a set of *threads* running in parallel. A *warp* is a group of threads that can run simultaneous on a *streaming multiprocessor (SM)*. The warp size

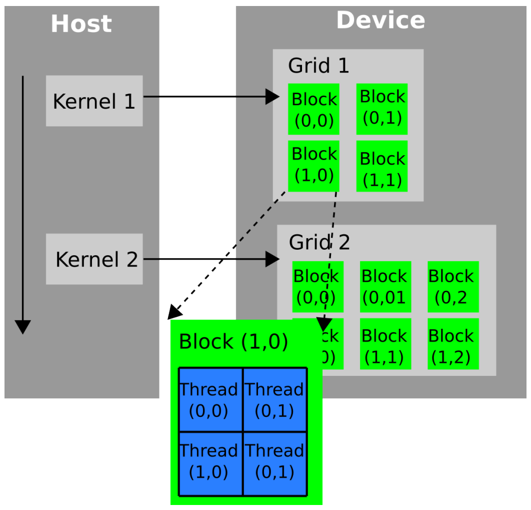


Figure 2.5: CUDA Memory Grid Configuration

is fixed for a specific GPU, 32 for example. It is up to the developer to set the number of threads to be executed. If the number of threads is more than the warp size, they are time-shared internally on the SM. A *block* is a group of threads running on a one SM at a given time. Multiple blocks can be assigned to a single SM. The set of all configured blocks is called a *grid* (Figure 2.5). All SM resources are divided equally between all threads of all blocks of the SM. A unique ID is associated to each single thread and block which is accessed by the thread during its execution. All threads will execute the same instructions set called *kernel*. Listing 2.4 gives an example of a matrix-vector multiplication with `CUDA`.

```
__global__ void matVecMul_CUDA(int n, int m, double *restrict a, double *restrict b, double *
    restrict c)
{
   int row = blockIdx.y*blockDim.y+threadIdx.y;
   float sum =0;
   if (row <n){
     for (int k=0; k<m; k++)
           sum += b[row*m+k] * a[k];
   }
   c[row] = sum;
}
```

Listing 2.4: CUDA Matrix-Vector Multiplication Source Code.

CUDA Memory Model All `CUDA` enabled GPUs have different memory spaces. Each type of memory has its own specifications such access latency, address space, scope and lifetime (table 2.1). Each memory type has some trade-offs to be considered when developing algorithms `CUDA` kernels. Developers have to be aware of how and when to use each type of memory to optimize the performance of their applications. For example, global memory has a very large address space, but the latency to access this memory type is very high. Shared memory has a very low access latency but the memory address is small compared to Global memory.

Table 2.1: CUDA Device Memory Specifications

Memory	Speed Rank	Location	Cached	Scope	Access	Lifetime
Register	1	On-chip	No	1 thread	R/W	Thread
Shared	2	On-chip	No	1 block	R/W	Block
Constant	3	Off-chip	Yes	All threads + host	R	Host allocation
Texture	4	Off-chip	Yes	All threads + host	R	Host allocation
Local	5	Off-chip	CC 2.0	All threads + host	R/W	Host allocation
Global	6	Off-chip	CC 2.0	All threads + host	R/W	Host allocation

2.3.2.2 OpenCL

Open computing language (`OpenCL`) [14] is an open programming standard proposed by Khronos group for heterogeneous parallel computing hardware. Currently there are plenty of `OpenCL` supported hardware in the market. Major chip manufactures all provide their `OpenCL` support for CPU, desktop GPU, and embedded GPU [44], [45], [46], [47]. There are also some researches on evaluating `OpenCL` on other platforms such as embedded multi-core digital signal processors [48]. In the literature, several implementations of CV applications with `OpenCL` have been performed. In [49], the authors proposed an implementation of a graph-based classifier (Optimum-Path Forest) in a SoC/FPGA board using the `OpenCL` language.

OpenCL Programming Model `OpenCL` controls multiple *Compute Devices*, for instance, GPUs. Each of these compute devices consists of multiple *Compute Units* (arithmetic processing unit, cores in multi-core) and within these are multiple *Processing Elements*. At the lowest level, these processing elements all execute `OpenCL` *Kernels*.

At the top level, the OpenCL host uses the OpenCL API platform layer to query and select compute devices, submit work to these devices and manage the workload across compute contexts and work queues. In contrast, at the lower end of the execution hierarchy , we find OpenCL *Kernels* running on the each processing element. Listing 2.5 illustrates a naive implementation of matrix-vector multiplication with OpenCL.

```
__kernel void matVecMul_OpenCL(int n, int m, __global double *restrict a, __global double *
    restrict b, __global double *restrict c)
{
  int tx = get_global_id(0);
  float sum = 0;
  for (unsigned int k = 0; k < m; ++k) {
    sum += b[tx*m + k] * a[k];
  }
  c[tx] = sum;
}
```

Listing 2.5: OpenCL Matrix-Vector Multiplication Source Code.

OpenCL Memory Model The OpenCL memory hierarchy is structured in such a way to resemble the physical memory configurations of NVIDIA hardware. The basic structure of top global memory vs local memory per work-group is consistent. Furthermore, the lowest level execution unit has a small private memory space for program registers.

An important issue to keep in mind when programming OpenCL Kernels is that memory access on the DRAM global and local memory blocks is not protected in any way. This means that segfaults are not reported when work-items dereference memory outside their own global storage [50].

2.3.3 Data Flow-Based Graphs for Parallel Programming

Among the proposed approaches for developing efficient CV applications for embedded systems, we find those based on data flow graphs [51]. These techniques profit from the data-flow nature of image processing algorithms. Graph-based techniques describe another parallel approach for accelerating CV applications. That is, image processing algorithms are expressed explicitly by the user as a graph of tasks [52]. In this case, tasks are managed based on the data-flow dependencies. All aspects related to task communication, synchronization and scheduling are hidden to the user.

In [52], the authors proposed a domain specific high-level parallel programming model for digital signal-processing applications referred as `SignalPU` ([53]). It is based on data-flow graph models of computation, and `StarPU` ([54]) as a dynamic run-time model of execution. `StarPU` is a runtime system designed to dynamically schedule a group of tasks on heterogeneous cores with different scheduling policies. The presented model in `SignalPU` targets heterogeneous clusters composed of CPUs and different accelerators such as GPUs. The execution takes advantage of the availability and the capability of the devices based on `StartPU` scheduler.

`OpenVX` [55] is a well known graph-based framework for accelerating image processing applications. `OpenVX` allow the programmer to optimize the algorithm at *system-level*. By system-level optimizations, we make reference to all optimizations applied to optimize the algorithm as a whole system such as data transfers, memory bandwidth, inter-processors communication, kernels fusion, etc. `OpenVX` framework is employed in this work. A more detailed description of its programming model and possible optimizations are given in Chapter 6.

2.3.4 DSL

There are some Domain-specific languages (DSLs) which provide a high-level language interface suitable for image processing algorithms such as `Halide` [16] and `NT2` [56]. They have been proposed to improve programmability as well as deliver performance for both general-purpose processors as well as accelerators like GPUs and FPGAs.

2.3.4.1 Halide

Halide is an open-source domain-specific language for the complex image processing pipelines found in modern computational vision applications. It is embedded in `C++` language. Compiler targets include x86/SSE, ARM v7/NEON, CUDA, Native Client, and OpenCL [16].

Halide Programming Model A `Halide` program has two sections: one for the *algorithm*, and one for the processing *schedule*. The schedule specifies the size and shape of the image chunks that each core needs to process at each step in the pipeline, and it can specify data dependencies. Once the schedule is drawn up, however, Halide handles all the accounting automatically. If we then want to export a program to a different architecture supported by `Halide`, we just need to change the schedule, not the algorithm description. If we want to add a new processing step to the pipeline, we just need to add it to the description of the new procedure, without having to modify the existing ones. However, a new step in the pipeline will require a corresponding specification in the schedule.

Listing 2.6 illustrates a naive example of a matrix vector multiplication with `Halide`. The first step consist on describing the algorithm functionality as shown in line 6. Then, the algorithm is scheduled on the target where optimizations can be applied at this level as illustrated in line 8. We selected to apply data tiling with tiles of size $[256, 32]$. We also applied vectorization corresponding to `SIMD` instructions. Finally, parallelization is applied at line level. However, for this simple example, actually `Halide` manages tp perform the scheduling without specifying it explicitly by the programmer in the code.

```
 1 Func matVecMul_Halide(Func b, Func a ) {
 2   Func c;
 3   Var x, y, xi, yi;
 4   RDom r(0, b.width(), 0, b.height );
 5   // The algorithm
 6   c(y) += (b(r.x,r.y) * a(r.x));
 7   // The schedule
 8   c.tile(x, y, xi, yi, 256, 32).vectorize(xi, 8).parallel(y);
 9   return mvMul;
10 }
```

Listing 2.6: Halide Matrix-Vector Multiplication Source Code

2.3.4.2 NT2

The Numerical Template Toolbox (`NT2`) is an open source C++ library aimed at simplifying the development, debugging and optimization of high-performance computing applications. It provides a `MATLAB`-like syntax that eases the transition between the prototype and the actual application.

`NT2` keeps a high level of expressiveness by exploiting the architecture-specific information as early as possible in the code generation process. It selects architectural features from either compiler-based options or user-defined preprocessor. In all cases, no additional code is required, the code is written once with `NT2` statements. The main asset of `NT2` is its ability to distinguish between architecture and runtime support. User can test different optimizations by setting different run-time supports for a given architecture until required performance is partially or completely satisfied.

NT2 Execution Model NT2 implements a subset of MATLAB language as a DSEL based on *Expression Template* C++ idiom. As such, developers can convert MATLAB code easily to NT2 by copying the original code to a C++ file and by performing minor changes (e.g., variables' definition, functions call instead of operators).

The main element of NT2 is the template class **table**. The latter's type can be parametrized with additional optional settings. The behavior of a **table** instance is similar to MATLAB multi-dimensional arrays. An important subset of MATLAB functions has been implemented on NT2 such as arithmetic, exponential, trigonometric and bitwise functions. Vectorization based on Boost.SIMD [57] is supported on all these functions. Boost.SIMD is a C++ template library which simplifies programming on SIMD by providing a high-level abstraction to handle vectorization computations on SSE, Altivec, AVX and NEON. The wrapper class **pack** of Boost.SIMD selects automatically the best SIMD register type according to its scalar template type.

NT2 Parallel Code Generation NT2 C++ library is designed based on Expression Template [58]. The latter is a well known meta-programming technique. It constructs at compilation a type representing the Abstract Syntax Tree (AST) of an arbitrary statement. Boost.proto [59] has been used for this purpose. Compared to traditional DSEL based on hand-written expression templates, Boost.proto provides a higher level of abstraction based on pattern matching of DSEL statements.

The expression evaluation of NT2 is based on **Generative Programming** [60]. This approach consists in defining a model to implement several components of a system. For instance, the Standard Template Library (STL) provides components that the user aggregates according to his configuration knowledge. Czarnecki [60] proposed a methodology called DERMAL[1]. DERMAL formalization of Generative Programming techniques. However, it does not manage architecture level during components generation. To cope with this issue, NT2 integrates the architecture support as another generative component. A new methodology Architecture Aware DERMAL (AA-DERMAL) [56] has then been introduced.

NT2 relies on Tag Dispatching technique to consider architecture in AA-DERMAL. With this technique, NT2 selects the correct implementation of a function while still being aware of its properties. The best implementation for the current architecture

[1]Domain Engineering Method for Reusable Algorithmic Libraries

is then selected. A tag is defined for every supported architecture such as `openmp_` for `OpenMP` shared memory parallelization. Nesting architecture tags is also possible at compile-time to generate automatically codes for different architecture levels.

NT2 Optimizations At `NT2` compilation, automatic rewriting may occur when architecture-driven optimizations are possible and user high-level algorithmic improvements are introduced. These optimizations are integrated within `Boost.SIMD` [57]. The most important optimizations include:

- *Vectorization* on sub-matrix access through the *colon* option `a(_, i)` (corresponding to : in MATLAB)
- *Loop fusion* to increase cache locality through the **tie** function which groups statements of compatible dimensions in a single loop nest. The multi-statement code snippet (lines 2 and 3) in Listing 2.7, can be converted to a single statement with `tie` function (line 4).

```
1 table<float> a, b, c;
2 a = b + 2.f*c;
3 c = a - 3.f/b;
4 tie(a,c) = tie(b + 2.f*c, a - 3.f/b);
```

Listing 2.7: Multi-statement and lopp fusion code snippet

Another important asset of `NT2` is its ability to extend its architecture and run-time back-ends support. The current supported architectures and runtimes include:

- *`SIMD`* extensions through `Altivec` on PowerPC, `AVX` and all `SSE` variations (from `SSE2` to `SSE4.2`) on x86, and `NEON` on ARM architecture.
- *Shared memory systems* through `OpenMP` or Intel Threading Building Blocks (`TBB`)
- *`LAPACK`-like runtimes* for `NETLIB LAPACK`, `MKL` and `MAGMA` on `CUDA` capable GPUs.
- *GPGPUs* by generating `CUDA` kernels directly from `C++` code with `NT2` statements. The kernel generator has been integrated on existing `CUDA` libraries like `cuBLAS` and `MAGMA`.

Listing 2.8 illustrates how to perform matrix-vector multiplication with `NT2`. We first give the `NT2` version without any optimization (lines 8 $\rightarrow$ 13) to show how to employ `NT2` tables. Then, we show the equivalent version by using the `colon "_"` option to accelerate the processing (lines 15, 16). This optimization allows to vectorize the operations at line level by enabling the SIMD compiling flags.

```
 1 #define N 10000
 2 using nt2::_;
 3 table<double> b = ones(N,N);
 4 table<double> a = ones(N);
 5 table<double> c(nt2::of_size(N));
 6 double sum;
 7 // NT2 without any optimization
 8 for (int i=1; i<=N; i++) {
 9   sum = 0;
10   for (int j=1; j<=N; j++)
11     sum += b(i,j)*a(j);
12   c(i) = sum;
13 }
14 // NT2 with colon (_) optimization
15 for (int j =1; j<=N; j++)
16   c(_) += b(_,j)*a(j);
```

Listing 2.8: NT2 Matrix-Vector Multiplication Source Code with/without Optimization

2.3.5 Vectorization: SIMD

SIMD instructions allow to achieve a form of data parallelism. This is sometimes also called *vector processing*. Let's take array multiplication to explain the principle.

```
float a[n], b[n], c[n];
for(int i = 0; i < n; i++) {
  c[i] = a[i] * b[i];
}
```

Listing 2.9: Array Multiplication: Scalar Version, One Element per Iteration.

In scalar version as shown in Listing 2.9, a loop that iterates through all array elements, multiplying each element of first array with its corresponding from second array, and storing the result in the destination array. What if instead of doing one multiplication per loop iteration we could do more?

```
float a[n], b[n], c[n];
for(int i = 0; i < n; i+=4) {
  c[i]   = a[i]   * b[i];
  c[i+1] = a[i+1] * b[i+1];
  c[i+2] = a[i+2] * b[i+2];
  c[i+3] = a[i+3] * b[i+3];
}
```

Listing 2.10: Array Multiplication: Scalar Version, Four Elements per Iteration.

```c
#include <xmmintrin.h>
float a[n], b[n], c[n];
__m128 A, B, C;
for (int i = 0; i < nElements; i += 4) {
// load data in a, b
  A = _mm_load_ps(&a[i]);
  B = _mm_load_ps(&b[i]);
// multiply data
  C = _mm_mul_ps(A, B);
// store result in c
  _mm_store_ps(&c[i], C);
}
```

Listing 2.11: Array Multiplication: `SSE` Version.

In Listing 2.10, we process four elements at once. This lets us iterate through the loop four times fewer. This optimization is called *loop unrolling*. In terms of efficiency we have gained very little since the same arithmetic operations are being done; this is where `SIMD` instructions come in. Instead of loop unrolling that performs four multiplications in series there is a single instruction that can calculate four multiplications simultaneously. `SIMD` uses special registers on the CPU that are 128 bits wide. These registers can hold any type of data that will fit in 128 bits, like two double precision numbers, four single precision, or 16 bytes.

The program first needs to explicitly load data into the `SIMD` registers by using the `_mm_load_ps()` intrinsic (Listing 2.11). The `__m128 _mm_mul_ps (__m128 a, __m128 b)` instruction multiplies packed single-precision (32-bit) floating-point elements in a and b, and store the results in destination register. The result is then moved from `SIMD` register into an output array using the `_mm_store_ps()` intrinsic (Listing 2.11).

2.4 Conclusion

In this chapter, we presented an overview of the existing hardwares and parallel programming tools for vision-based applications. It is obvious that the choice is not trivial on both aspect. First, from hardware point of view, different architectures exist, each one with its advantages and limitations. Second, from software point of view, several parallel approaches have been developed. Different techniques and approaches are employed while targeting different platforms. The developers of embedded vision-based applications meets several challenges and ask different questions such as: which hardware will fit well the selected application in terms of performance computing? which programming language allows to achieve the real-time conditions with less programming efforts? at which level adaptations are applied in the algorithm to reach the requirements? what is the best compromise between algorithm accuracy and performance? All these questions are discussed in this work. We investigate several programming tools on different hardwares. A complex use-case is selected based on stereo vision to detect vehicles. The results and the feedback of this work will help future embedded vision-based developers to develop CV applications with less programming effort while respecting the initial requirements.

3

ADAS Use Case

Stereo Vision Based Vehicles Detection Algorithm

In this chapter, we propose a coarse to fine algorithm for on-road vehicles detection and distance estimation based on disparity map segmentation [17]. The whole algorithm is supervised by stereo vision.

One of the common issues in on-road obstacles detection are the off-road information such as high walls, buildings and trees along the road which may affect the precision of the detection and increase the false alarms. Different approaches have been proposed in the literature such as constructing a convex hull [61]. The **first contribution** of this work is the approach proposed to remove the off-road objects based on the connected segments labeling algorithm. While this approach has been used in the literature [61] mainly for on-road obstacles detection, we propose to use it also for off-road objects subtraction. The **second contribution** of the proposed algorithm is the approach employed for obstacles classification. We rely on a set of tests based on the obstacles geometry as the width and height which are not measured in meters, they are expressed in pixels in terms of the disparity. The idea is inspired from the fact that close objects are projected with many pixels while far objects with only few pixels. This solution reduces the errors of disparity quantification and allows more accurate detection and classification.

3.1 Related Works

Stereo vision systems have recently emerged in the domain of robotics and autonomous cars. These systems provide the 3D perception of the environment which is employed in

Advanced Driver Assistance Systems (ADAS) to support a variety of functions including obstacles detection [62], [63], [61], lane departure warning [64], [65] and collision warning systems [66]. While the depth measurement precision of stereo vision systems is not as high as with active sensors such as RADAR and LIDAR, binocular stereo vision can compete with these active technologies due to the amount of traffic scene information it provides in one hand, and the low cost of cameras in the other hand.

The literature describes several works on stereo vision based vehicles detection. All approaches investigate the data provided by the stereo cameras to extract the necessary information. The study of the state of the art in this field reveals that the majority of proposed approaches rely on a depth map which can be obtained either directly from specialized 3D cameras [67] or from a disparity map. The later is generated through *stereo matching*; namely the search of points correspondences between two images in the horizontal direction after rectification. The literature shows two major axes in the field of stereo matching. In the first axis, both monocular and stereo visions are employed. The idea is to perform features tracking in the monocular image plane of one of the stereo cameras and 3-D localization in the disparity and depth maps [68]. Several works on vehicles detection based on the combination of stereo vision and motion analysis have been conducted [68], [69] ,[70]. In the second axis, the disparity map is transformed in a more representative and compact form including occupancy grid [71] and ground surface modeling [72]. These different transformations aim mainly to facilitate scene segmentation and reduce computation time such as the U-V disparity approach.

3.1.1 U-V Disparity

The V-disparity [72] is a popular approach for ground plane estimation and road scene analysis [67]. It is a compact representation of the disparity map in a new space, more robust and more representative for obstacles detection lying above the ground surface. Actually, the V-disparity transformation is a histogram of disparity values for pixels within the same image's line (v coordinate). This transformation models the road surface as a slanted line and vertical obstacles as vertical lines above the road's line. Lines can be detected using curve fitting techniques such as Hough transform [73].

The V-disparity has been widely used in stereo vision for road plane estimation and obstacles detection. It has been introduced for the first time in [72]. The authors propose an approach for road estimation and obstacles detection under the hypothesis that the road is flat. In [67], the V-disparity is used to detect the road based on a modified Hough transform.

The same principle has been used to generate the U-disparity map with a histogram of pixels locating on the same image's column (u coordinate). This map has been used for free space estimation [74] as well as for obstacles detection [67], [75].

Some works combine both maps–U-V, to perform road and obstacles detection. In [67], the V-disparity is used to detect the road as well as the vertical position of the obstacles. The U-disparity map is used to find the horizontal position of the obstacles. The authors in [76] follow the same approach but rely on a 3D camera to generate the depth map in a addition to a modified Hough transform which is placed to extract the straight line feature from the depth map with improved accuracy.

As mentioned previously, obstacles detection based on the U-V disparity maps relies on the detection of lines on those maps by using generally the Hough transformation. However, possible stereo matching errors and high structured scene make the results of these maps largely noisy and distorted. Another issue to take into consideration is the fact that the frontal vehicles are projected in the U-disparity map under the form of an horizontal segment, where as, vehicles close to camera or those coming in the inverse direction are projected with more that one segment which may be horizontal or slanted. The hough transform suffers with such cases. To overcome these issues, we propose to use connected segments labeling algorithm which recovers all vehicle's pixels projected on the U-disparity. This proposed solution increases the detection's precision and gives a better estimation of the distance.

3.2 Stereo Matching Basics

In this section, we describe the fundamental principles of stereo matching. We review the different techniques employed for finding a set of corresponding points between two images. This task involves selecting a suitable similarity measure and then employing local or global methods to identify correct matches.

3.2.1 Epipolar Geometry

Stereo matching algorithm consists on matching a given pixel in one image to its corresponding pixel in the other image. This definition implies that establishing correspon-

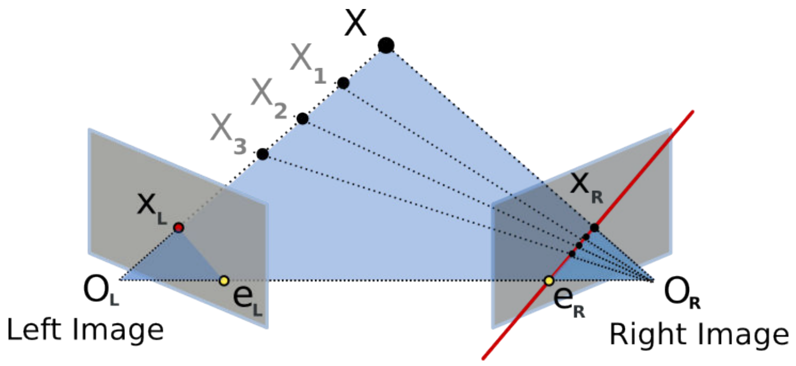


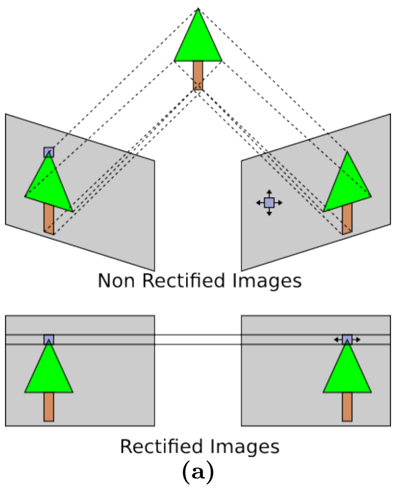
Figure 3.1: Epipolar Geometry in Stereo Vision

dences requires a search through the whole image. Fortunately, the *epipolar constraint* of stereo vision reduces this research to a single line.

In Figure 3.1, left and right stereo cameras are centered at O_L and O_R respectively. The point X is the point of interest. Points X_L and X_R are the projections of the point X on the left and right images planes respectively. We call *baseline* the line connecting the two cameras centers $(O_L - O_R)$. The gray planes in Figure 3.1 represent the epipolar planes. The intersection of the baseline with each image's plane gives the *epipoles*, e_L and e_R in the figure. The epipolar constraints states that for each point projected in an image plane, the same point must be projected in the other image on a known epipolar line [77]. In other words, the projection of point X in the right image plane X_R must be in the epipolar line $e_R - X_R$. This is true if only the projection X_L is known, and the epipolar line in the right image plane $(e_R - X_R)$ is also known. It is important to note that all points X, X_1, X_2, X_3 verify this constraint.

This constraint simplifies then the stereo matching process by limiting the research region of correspondences to a single line instead of the whole image. This implies that the position of corresponding points is only different in horizontal direction. To satisfy this condition, a transformation projects both stereo images onto a common image plane in such a way that the corresponding points have the same row coordinates. This image projection makes the image appear as though the two cameras are parallel. This transformation is referred as *rectification* and clarified in Figure 3.2 (a). Now, finding the corresponding point in the right image becomes more convenient.

As discussed previously, the main goal behind stereo matching is to recover the



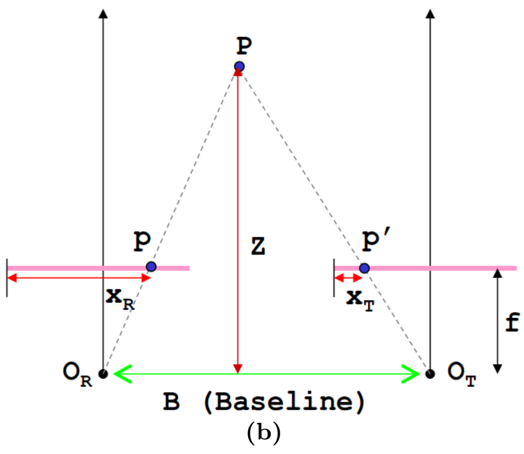


Figure 3.2: Rectification (a) and Depth Measurement (b) from Stereo Matching

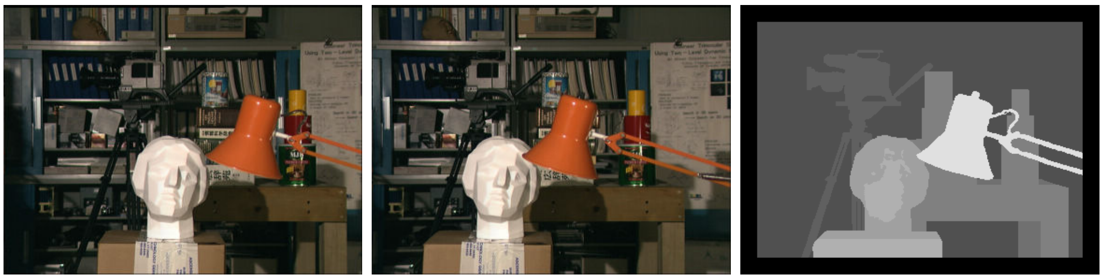
depth and perform 3D reconstruction for instance later on. After the rectification, it is easy to compute the depth of the point apart from the cameras [78]. From Figure 3.2 (b), based on the triangles (PO_RO_T) and (Ppp'), we could get the depth value (Z) through Equation 3.1. Z represents the real distance of the considered point P apart from the stereo camera baseline, f is the focal distance of the camera, and B is the baseline distance between the two stereo cameras. Finally, we replace $x_R - x_T$ with d referred as ***disparity***.

$$\frac{B}{Z} = \frac{(B + x_T) - x_R}{Z - f} \implies Z = \frac{B \times f}{x_R - x_T} = \frac{B \times f}{d} \tag{3.1}$$

Disparity is the horizontal difference (x coordinate) between two corresponding pixels in the stereo images. In computer vision and image processing, when the disparity of all pixels is known, a *disparity map* is generated where each pixel intensity presents the disparity of the corresponding pixel in gray scale. Based on Equation 3.1, disparity is inversely proportional to real depth, hence, pixels closer to the camera (with lower Z), have higher disparity and looks brighter. This property is illustrated in Figure 3.3.

3.2.2 Correspondences : Matching Pixels

Numerous stereo matching algorithms have been proposed over the years. The study of the state of the art shows two categories; *sparse* and *dense* stereo matching [78].



(a) Left Image **(b)** Right Image **(c)** True Disparity Map

Figure 3.3: Disparity Map Example from Middlebury Dataset [1] (Tsukuba2001)

In the first axis; *sparse stereo matching*, features such contours or edges are first extracted from the images. Features from one image are then matched to their corresponding ones in the other image. The resulting sparse depth map or disparity map may then be interpolated using surface fitting algorithms. Early work in finding sparse correspondences was motivated mainly by the limited computational requirements at the time, but also by the observation that certain features in an image give more reliable matches than others. Such features include edge segments and profile curves. These features occur along the occluding boundaries. However, these early algorithms required several closely-spaced camera viewpoints in order to stably recover features.

Recent research has focused on extracting features with greater robustness and more important repeatability. These features are then employed to interpolate the missing disparities. These algorithms are referred as *feature points detectors*. The literature is rich in this field. Numerous algorithms have been proposed such as SIFT [79] and SURF [80]. These approaches extract a set of discriminative points with different techniques.

The second axis; *dense stereo matching*, aims to find a dense set of correspondences between two or more images. The resulting dense depth map is useful for 3D modeling and rendering applications. These techniques typically involve the calculation and aggregation of matching costs, from which disparity values may then be computed and refined. In this axis, a lot of stereo matching algorithms have been proposed and the research of fast and precise stereo correspondence problem is still going on. A taxonomy and evaluation of dense stereo matching algorithms has been presented in [78]. From this taxonomy, most of dense stereo matching algorithms perform the following steps:

1. Matching cost computation.

2. Cost aggregation.
3. Disparity computation or optimization.
4. Disparity refinement (optional).

3.2.2.1 Matching Cost Computation

Regardless of the stereo matching algorithm used, determining the similarity between pixels in different images is the core to remove ambiguities between potential matches and to find correspondences. Let's take a given pixel $p(u, v)$ in the left image I_l. The objective is to find the corresponding point of pixel p in the right image I_r. To do so, we need to measure the *degree of similarity* between left and right pixels.
One of the basic measure is the Squared intensity Difference (SD) [[81], [82]], given by equation 3.2.

$$C_{SD}(u, d) = (I_l(u, v) - I_r(u - d, v))^2. \tag{3.2}$$

In the presence of image noise, the similarity measure may be made more robust to outliers within the window by imposing a penalty that grows less quickly than the quadratic SD term. A common robust measure is the Absolute intensity Difference (AD) [[83], [84]] given by equation 3.3.

$$C_{AD}(u, d) = |I_l(u, v) - I_r(u - d, v)|. \tag{3.3}$$

In this case, the cost function grows linearly with the residual error between the windows in the two images, thus reducing the influence of mismatches when the matching cost is aggregated. Besides measuring the residual error in window intensities, another commonly used similarity measure is the Cross Correlation where the maximum correlation value among all the candidate disparities corresponds to the most probable match. There exist other traditional matching costs such as Normalized Cross Correlation (NCC) [85], and binary matching costs [86] based on binary features such as edges or the sign of the Laplacian. It is worth noting that binary matching costs are not commonly used in dense stereo matching.

3.2.2.2 Cost Aggregation

Actually the differences (SD, AD) are *aggregated* in a support window surrounding the considered pixel. Window-based methods aggregate the matching cost by summing or averaging over a support region. The most known measures are the SSD (Sum of SD) and SAD (Sum of AD). The commonly known regions are two-dimensional but they also may be three-dimensional. Traditionally, they have been implemented using square windows or Gaussian convolution. The literature is rich in this field, we can find the sliding window where a window is shifted along the epipolar line in the right image [83]. We find also the adaptive window technique where the size of the window is modified for higher disparity precision [[87], [88]].

3.2.2.3 Disparity Computation

Dense stereo matching algorithms may be subdivided into two main categories: *local* and *global*.

Local approaches determine the correspondence of a point by selecting the candidate point along the epipolar lines that minimizes a cost function. To reduce matching ambiguity, the matching costs are aggregated over a support window rather than computed point-wise. There are different techniques to get the disparity, the common one is the Winner-Takes-All (WTA) [[84], [83], [88]].

Global methods do not aggregate the matching costs, but instead rely on explicit smoothness assumptions to ensure accuracy. The objective of global stereo algorithms is to find the disparity assignment which minimizes an energy function. In fact, cost aggregation is not required with global global matching since each pixel has a range of disparities. Hence, there is an important number of disparities combinations for all pixels in the whole image. It is then obvious that global methods require more computations and memory capacity too. However, global methods give better disparity map in terms of quality. Among the most common methods, we find belief propagation [[89], [90]], dynamic programming [91] and graph cuts [92] techniques.

3.2.2.4 Disparity Refinement

Usually, the disparity is estimated over a discretized space such as for integers disparities. Discrete disparities are adequate for some applications as robot navigation and people tracking. However, for image-based rendering applications as quantized maps, discrete disparities gives a disparity or depth map with shearing layers. To cope with this issue, some stereo matching algorithms apply sub-pixel refinement to make the disparity map smoother. Finally, a set of post processing operations are applied such

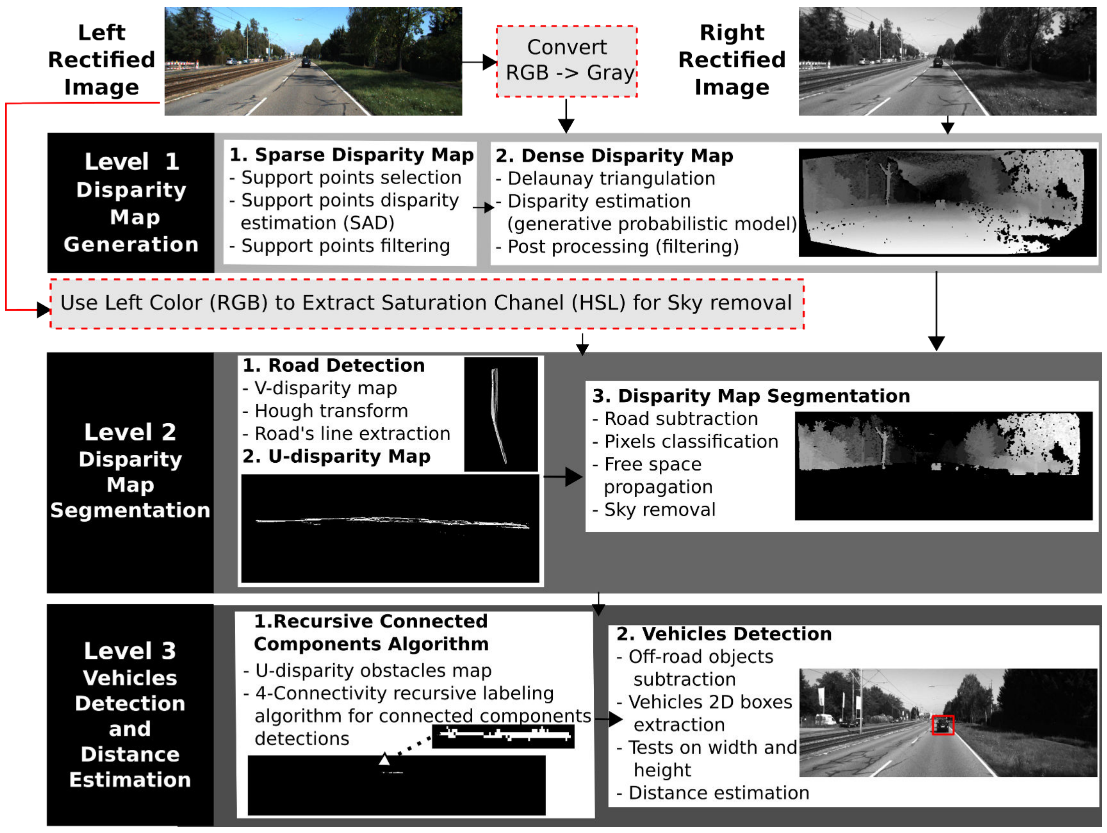

Left Rectified Image
Convert RGB -> Gray
Right Rectified Image
Level 1 Disparity Map Generation
1. Sparse Disparity Map
- Support points selection
- Support points disparity estimation (SAD)
- Support points filtering
2. Dense Disparity Map
- Delaunay triangulation
- Disparity estimation (generative probabilistic model)
- Post processing (filtering)
Use Left Color (RGB) to Extract Saturation Chanel (HSL) for Sky removal
Level 2 Disparity Map Segmentation
1. Road Detection
- V-disparity map
- Hough transform
- Road's line extraction
2. U-disparity Map
3. Disparity Map Segmentation
- Road subtraction
- Pixels classification
- Free space propagation
- Sky removal
Level 3 Vehicles Detection and Distance Estimation
1.Recursive Connected Components Algorithm
- U-disparity obstacles map
- 4-Connectivity recursive labeling algorithm for connected components detections
2. Vehicles Detection
- Off-road objects subtraction
- Vehicles 2D boxes extraction
- Tests on width and height
- Distance estimation

Figure 3.4: Proposed Vehicles Detection Algorithm's Functional Diagram

as left and right consistency check to detect occluded areas. Holes resulted from occlusions can be filled by gap interpolation techniques based on neighboring disparities to estimate holes' disparity [93]. A median filter can also applied to remove mismatches.

3.3 Vehicles Detection Proposed Algorithm

In this section, we describe the proposed algorithm for vehicles detection and distance estimation based on stereo vision. The approach can be divided into three processing levels as shown in Figure 3.4.
The first one deals with the generation of a dense disparity map based on the grabbed rectified left and right stereo images, the algorithm proposed in [94] has been used for this purpose. We selected this algorithm based on two important criteria. First, it is considered as one of the most accurate and efficient stereo matching algorithm which

has been widely employed in the literature [95], [96], [97], [98]. Second, the algorithm is most of the time sequential and difficult to parallelize.
The second level performs scene segmentation. This segmentation relies on the V-disparity map and a set of post processing. It provides all necessary information to facilitate the vehicles detection task for the next level, also it increases the detection efficiency and reduces the computation cost by limiting the region of interest. The result of this level is the extraction of the road plane profile with two disparity maps; a free space disparity map and an obstacles disparity map. Finally, vehicles detection is processed in the third level based on the U-disparity map generated from the obstacles disparity map.

3.3.1 Level 1 : Disparity Map Generation

In this work, we worked on the Efficient LArge scale Stereo matching (ELAS) algorithm [94] to generate the disparity map. The approach relies on the observation that not all pixels are difficult to match, but instead there should be a set of discriminative points which can be matched easily and used to generate a sparse disparity map.

As depicted in Figure 3.6, in the first level, a descriptor is associated to all pixels of both stereo images. Figure 3.5 illustrates how to compute each pixel's descriptor. Sobel gradients are used for this purpose. For both images and for all pixels we compute Sobel vertical and horizontal gradients. Then, we take a patch of size 5×5 centered on the considered horizontal and vertical gradients corresponding to the considered pixel. Actually, in the published paper [94], authors propose a descriptor of 25 elements consisting of the 25 elements of each patch concatenated together. However, for fast implementations, in the code they selected only 16 elements as depicted in Figure 3.5 which is said to give empirically the best results in terms of disparity quality and precision according.

Then a set of discriminative support points candidates are selected from the reference image. The second level is the core of the whole algorithm. First, the disparity of these support points is determined based on the classic Sum of Absolute Difference (SAD) method. These support points are then used to build a 2D mesh via Delaunay triangulation. This mesh helps to find the disparity of the remaining pixels based on a generative probabilistic model. Finally, some post-processing operations are applied

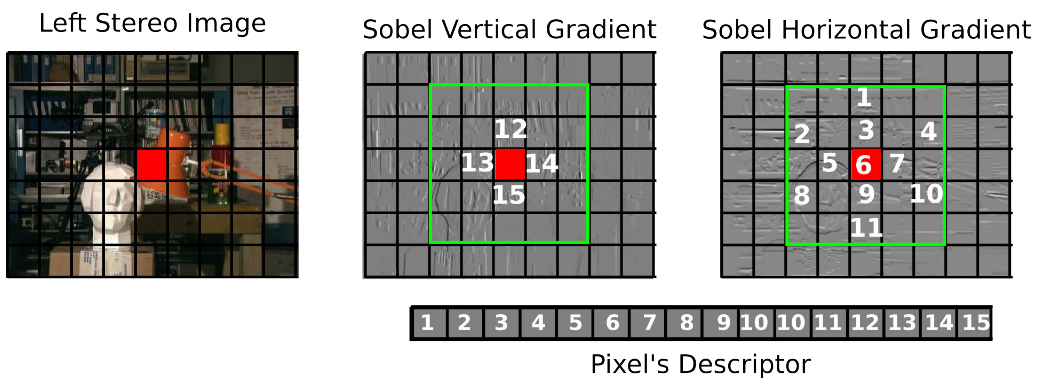


Figure 3.5: Pixel's Descriptor in Elas Algorithm

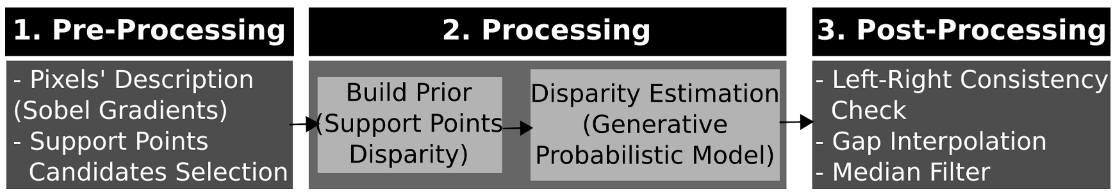

1. Pre-Processing
- Pixels' Description (Sobel Gradients)
- Support Points Candidates Selection
2. Processing
Build Prior (Support Points Disparity)
Disparity Estimation (Generative Probabilistic Model)
3. Post-Processing
- Left-Right Consistency Check
- Gap Interpolation
- Median Filter

Figure 3.6: ELAS Functional Diagram

to remove noise from generated map and interpolate the gaps resulted from occlusion areas. Figure 3.6 summaries the different operations encountered in ELAS algorithm.

3.3.1.1 Supports Points

ELAS algorithm generates a dense disparity map, i.e., it computes the disparity of all pixels. It is a Bayesian approach which builds a prior from a set of support points. In other words, it first builds a sparse disparity map providing the disparity of some points which will be used to estimate the disparity of all pixels. By support points, we refer to some pixels which can be robustly matched due to their texture and uniqueness [94]. Different approaches [83],[99], [100] have been proposed in the literature to find stable stereo correspondences. ELAS algorithm relies on a different approach. The idea consists on describing each pixel with a descriptor. The later is formed by concatenating the horizontal and vertical Sobel gradients of 9×9 pixel windows. A mask of size 3×3 is used to compute Sobel gradients. As cost function, the *l1* distance is employed between pixel's descriptors. The authors [94] experimented with sparse interest points descriptors of SURF [80]. They found that SURF descriptors do not improve disparity accuracy while being slow to compute compared to Sobel based descriptors.

3.3.1.2 Generative Probabilistic Model for Dense Stereo Matching

This section describes the generative probabilistic model used to construct the dense disparity map based on the support points disparity. Given the support points and an observation in the left image taken as the reference image, samples from the corresponding observation in the right image can be obtained based on a generative probabilistic model.
Let's formalize this idea:

- Let $\mathbf{S}$, be a set of support points $\mathbf{S} = \{s_1, \ldots, s_M\}$.
- Each support point $s_m = (u_m, v_m, d_m)^T$ is defined as the concatenation of its image coordinates $(u_m, v_m) \in N^2$ and its disparity $d_m \in N$.
- Let $\mathbf{O} = \{o_1, \ldots, o_N\}$ be a set of image observations.
- Each observation $o_n = (u_n, v_n, \mathbf{f}_n)^T$ is formed as the concatenation of its image coordinates $(u_m, v_m) \in N^2$ and a feature vector $\mathbf{f}_n \in R^Q$
- $\mathbf{o}_n^{(l)}$ and $\mathbf{o}_n^{(r)}$ are the observation in the left and right image respectively.

The left image is taken as the reference. Let's assume that the observations $\{\mathbf{o}_n^{(l)}, \mathbf{o}_n^{(r)}\}$ and the support points $\mathbf{S}$ are *conditionally independent*. Then, given their disparities d_n the *joint distribution* factorizes

$$p(d_n, \mathbf{o}_n^{(l)}, \mathbf{o}_n^{(r)}, \mathbf{S}) \propto p(d_n|\mathbf{S}, \mathbf{o}_n^{(l)})p(\mathbf{o}_n^{(r)}|\mathbf{o}_n^{(l)}, d_n) \tag{3.4}$$

- $p(d_n|\mathbf{S}, \mathbf{o}_n^{(l)})$ is the prior.
- $p(\mathbf{o}_n^{(r)}|\mathbf{o}_n^{(l)}, d_n)$ is the likelihood.

The authors took the prior to be proportional to a combination of a uniform distribution and sampled Gaussian.

$$p(d_n|\mathbf{S}, \mathbf{o}_n^{(l)}) \propto \begin{cases} \gamma + exp(\frac{-(d_n - \mu(\mathbf{S}, \mathbf{o}_n^{(l)}))^2}{2\sigma^2}) & if |d_n - \mu| < 3\sigma \vee d_n \in N_s \\ 0 & otherwise \end{cases} \tag{3.5}$$

With:

- N_s is a set of all support point disparities in a small 20×20 pixel neighborhood around the considered pixel $(u_n^{(l)}, v_n^{(l)})$.
  The condition $d_n \in N_s$ is to take into consideration discontinuities of disparity in some pixels where linearity is not verified.

- $\mu(\mathbf{S}, \mathbf{o}_n^{(l)})$ is a mean function which relates the support points to the observations.It is expressed as a piecewise linear function that interpolates the disparities using the Delaunay triangulation computed based on the support points.
  For each triangle in the mesh, we obtain a plane defined by :

$$u_i(\mathbf{o}_n^{(l)}) = a_i u_n + b_i v_n + c_i \tag{3.6}$$

  Where

  - i is the index of the triangle the pixel (u_n, v_n) belongs to.
  - $\mathbf{o}_n = (u_n, v_n, \mathbf{f}_n)^T$ is an observation.

For each triangle i, the plane parameters (a_i, b_i, c_i) are obtained by solving a linear system of three equations. These equations are formed by replacing the vertices's coordinates of the triangle in equation 3.6.
As an example, let's $\vee_1(u_1, v_1, d_1)$, $\vee_2(u_2, v_2, d_2)$ and $\vee_3(u_3, v_3, d_3)$ be the three vertices of triangle i. If we replace these three points in equation 3.6, we get:

$$\begin{cases} d_1 & = a_i u1 + b_i v1 + c_i \\ d_2 & = a_i u1 + b_i v2 + c_i \\ d_3 & = a_i u3 + b_i v3 + c_i \end{cases} \tag{3.7}$$

These equations can be transformed to a simple linear system $Ax = b$ That can be solved easily.

$$\begin{bmatrix} d_1 \\ d_2 \\ d_3 \end{bmatrix} = \begin{bmatrix} u_1 & v_1 & 1 \\ u_2 & v_2 & 1 \\ u_3 & v_3 & 1 \end{bmatrix} \times \begin{bmatrix} a_i \\ b_i \\ c_i \end{bmatrix}$$

Once the prior is evaluated, let's express the second part of equation 3.4 which is the likelihood. The authors have chosen to express the image likelihood as a constrained Laplace distribution as follow:

$$p(\mathbf{o}_n^{(r)} | \mathbf{o}_n^{(l)}, d_n) \propto \begin{cases} exp(-\beta ||\mathbf{f}_n^{(l)} - \mathbf{f}_n^{(r)}||_1) & if \begin{bmatrix} u_n^{(l)} \\ v_n^{(l)} \end{bmatrix} = \begin{bmatrix} u_n^{(r)} + d_n \\ v_n^{(r)} \end{bmatrix} \\ 0 & otherwise \end{cases} \tag{3.8}$$

$\mathbf{f}_n^{(l)}$ and $\mathbf{f}_n^{(r)}$ are the feature vectors of $\mathbf{o}_n^{(l)}$ and $\mathbf{o}_n^{(r)}$ respectively. Classical descriptors such as SIFT, SURF, BRIEF, ... could be used in such case to describe each pixel (observation) with respect to its neighborhood. The authors made a choice to use Sobel filter to form a feature descriptor by concatenating horizontal and vertical gradients.

3.3.1.3 Samples' Selection on the Right Image

Once the prior and the likelihood have been calculated, samples or observation on the right image can be extracted based onto these two measures and by using equation 3.4. The idea is to extract a set of samples or points in the right image which match most likely an observation in the left image.
To do so, we need to:

1. Obtain a disparity d_n from the prior $p(d_n|\mathbf{S}, \mathbf{o}_n^{(l)})$ given $\mathbf{S}$(support points) and $\mathbf{o}_n^{(l)}$ (left image observations).
2. Draw an observation $\mathbf{o}_n^{(r)}$ from the likelihood $p(\mathbf{o}_n^{(r)}|\mathbf{o}_n^{(l)}, d_n)$ given $\mathbf{o}_n^{(l)}$ and d_n.

3.3.1.4 Disparity Estimation

The purpose behind defining the prior (equation 3.5) and the likelihood (equation 3.8) is to determine at the end the disparity of the remaining pixels if we exclude the support points to construct a dense disparity map. As discussed in the previous section, samples on the right image can be drawn for each observation in the left image. Based on these samples, we can estimate the disparity, the authors have relied on Maximum a-posteriori **MAP** to compute the disparity given by the following formula:

$$d_n^* = argmax \quad p(d_n|\mathbf{o}_n^{(l)}, \mathbf{o}_1^{(r)}, \ldots, \mathbf{o}_N^{(r)}, \mathbf{s}) \tag{3.9}$$

The authors in [94] proposed a stereo matching algorithm which is able to compute accurate disparity maps of high resolution images at frame rates close to real time. They have shown that a prior distribution estimated from robust support points reduces stereo matching ambiguities. ELAS algorithm has been evaluated on the Middlebury benchmark and real-world imagery. The results show that the proposed approach performs better with respect to the state-of-the-art approaches.

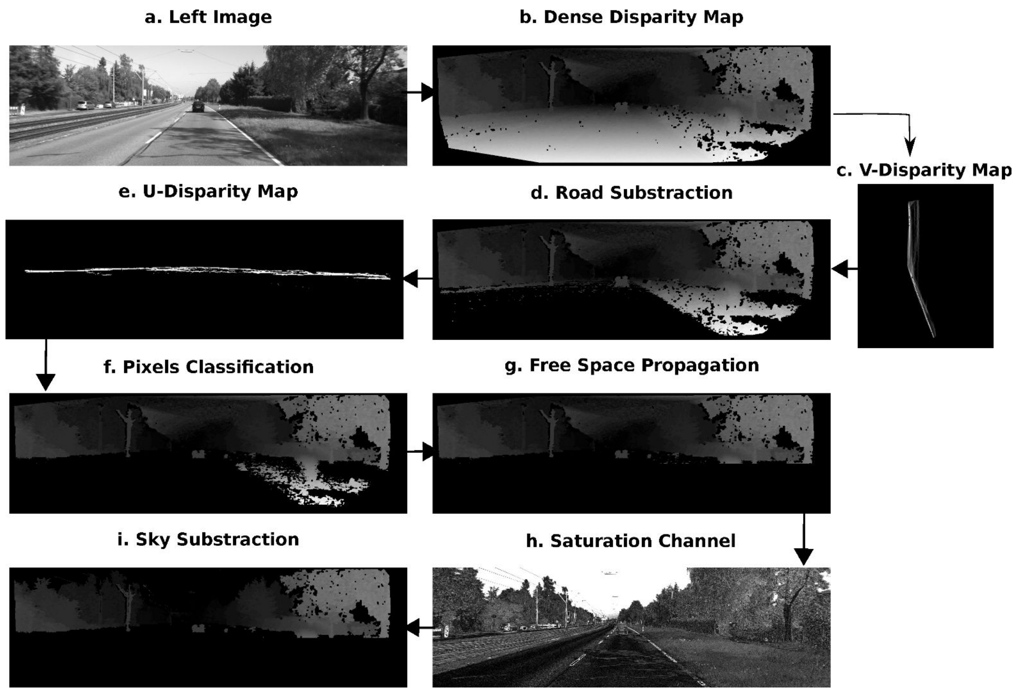


Figure 3.7: Scene Segmentation Approach

3.3.2 Level 2 : Scene Segmentation

The dense disparity map is segmented into two distinctive spaces; free space and obstacles space (Figure. 3.7). The former one includes road, sidewalks and sky, the later one covers static obstacles (trees, building, panels) and dynamic ones (moving vehicles and pedestrians). The obstacles map can be viewed as a confident map since the probability of detecting vehicles is higher in this map compared to the complete dense disparity map.

3.3.2.1 Road Detection

The detection of the road profile is based on the V-disparity space [72] where the x axis plots the disparity d and the y axis plots the image row number. The intensity value of a pixel $p_{vdisp}(d, v)$ on this map is identified According to Eq. (3.10).

$$p_{vdisp}(d, v) = \sum_{u=0}^{cols} \lambda, \ \lambda = \begin{cases} 1 & \text{if } p_{disp}(u, v) = d \ . \\ 0 & \text{otherwise} \ . \end{cases} \tag{3.10}$$

The ground plane is projected as a slanted line under the hypothesis that the road's surface is plane and horizontal. The Hough transform [73] has been used to detect this line and to identify its equation $ad + b$. Then, for each pixel $p_{disp}(u, v)$ in the disparity map, we calculate its distance $dist$ (Eq. (3.11)) with respect to the road's line which should be less than a threshold ϵ (few pixels) to classify this pixel as road's pixel. Through this process, road's pixels are subtracted from the disparity map. Figure 3.7 (b, c, d) shows results of road segmentation (d) based on the dense disparity map (b) and the V-disparity map (c).

$$p_{disp}(u, v) = \begin{cases} 0 & \text{if } dist < \epsilon \\ p_{disp}(u, v) & \text{otherwise} \end{cases}, \; dist = \frac{|au - vb|}{\sqrt{1 + a^2}} \; . \tag{3.11}$$

3.3.2.2 Pixels' Classification

The classification phase aims to recover the non-road free space pixels by using the U-disparity map. The intensity of a pixel $p_{udisp}(u, d)$ in the U-disparity map is determined according to Eq. (3.12)).

$$p_{udisp}(u, d) = \sum_{v=0}^{rows} \lambda, \;\; \lambda = \begin{cases} 1 & \text{if } p_{disp}(u, v) = d \; . \\ 0 & \text{otherwise} \; . \end{cases} \tag{3.12}$$

If the intensity of a pixel on the U-disparity map is higher than a certain threshold τ, this means that in a certain column u of the disparity map, there are too many pixels with the same distance to the camera and these points belong to potential obstacles. Based on this observation, pixels with high intensity are kept and the others are set to 0 (Eq. (3.13)). The threshold τ refers to the height of obstacles measured in pixels, in our case, it has been set to 40 pixels. Figure 3.7 (d, e, f) shows the results of pixels classification (f) based on the disparity map after road subtraction (d) and by using the U-disparity map (e).

$$p_{disp}(u, v) = \begin{cases} p_{disp}(u, v) & \text{if } p_{udisp}(u, d) > \tau, d = p_{disp}(u, v) \; . \\ 0 & \text{otherwise} \; . \end{cases} \tag{3.13}$$

3.3.2.3 Free Space Propagation

Pixels classification has been performed to get two sets of pixels candidates belonging to free space and obstacles space. These classified pixels are taken as initial seed points to determine the class of the non-classified pixels. The idea is to count the contribution of classified free space and obstacles space pixels on the neighbor of each pixel. An accumulator *accum* has been used to count this contribution. If the neighbor's pixel belongs to free space, *accum* is decremented, otherwise it is incremented (Eq. (3.14)). Figure 3.7 (g) shows the results of free space propagation.

$$d(u,v) = \begin{cases} 0 & \text{(free space pixel) if } accum < 0 \\ d(u,v) & \text{(obstacles space pixel) otherwise} \end{cases} \tag{3.14}$$

3.3.2.4 Sky Subtraction

While many free space pixels have been recovered through free space propagation, wrong classification may happen concerning the sky's pixels classified as obstacles pixels (Figure 3.7(g)). The reason is that, the propagation task relies on the analysis of the surrounding pixels, since the sky's pixels are on the top part of the image far from the free space and close to obstacles space, the contribution of obstacles pixels is higher. To remove sky from disparity map, the saturation (S) channel (Figure 3.7(h)) of the HSL color space is used and applied as a mask on the last segmented disparity map by using Eq. (3.15).

$$d(u,v) = \begin{cases} 0 & \text{if } s(u,v) \text{ is } (\text{black}(0) \vee \text{white}(255)) \wedge (v < b - \epsilon) \\ d(u,v) & \text{otherwise} \end{cases} \tag{3.15}$$

The saturation channel has been used because the sky's pixel $s(u,v)$ on this channel is either white or black. It is white in case the sky is blue or grey on the RGB color space and it is black when the sky is white. We may then apply the S channel as a mask to each pixel $d(u,v)$. However, to avoid removing pixels belonging to potential vehicles, the mask is applied only for the part above the horizon line b identified previously (see section 3.3.2.1) with a tolerance range ϵ (30 pixels). The result is shown on Figure 3.7(i).

3.3.3 Level 3 : Vehicles Detection

Vehicles detection task relies on the U-disparity map which is less noisy compared to the the V-disparity map. This choice is based on the fact that, in the U-disparity map, obstacles in general are represented by separate horizontal lines even side to side vehicles which is not the case on the V-disparity map where obstacles at the same distance are

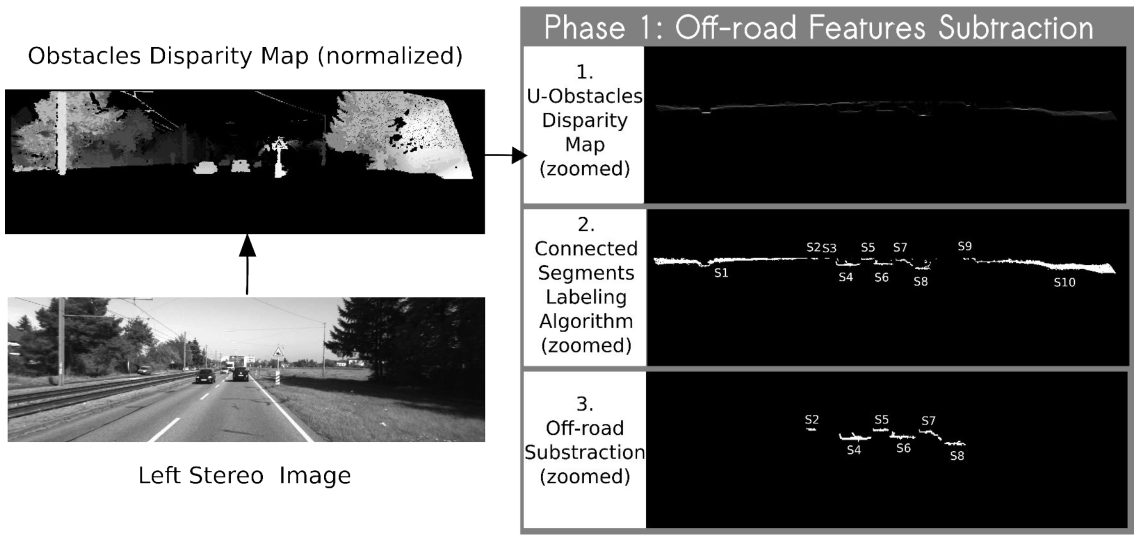

Obstacles Disparity Map (normalized)
Left Stereo Image
Phase 1: Off-road Features Subtraction
1. U-Obstacles Disparity Map (zoomed)
2. Connected Segments Labeling Algorithm (zoomed)
S1
S2
S3
S4
S5
S6
S7
S8
S9
S10
3. Off-road Substraction (zoomed)
S2
S4
S5
S6
S7
S8

Figure 3.8: Vehicles Detection Algorithm: *Phase 1* Off-road Features Subtraction

overlapped and represented by the same line. However, the crucial point with the U-disparity map is how to remove the off-road features. To cope with this issue, connected segments labeling algorithm has been used. Finally, on-road vehicles are detected and recognized based on their geometry. The whole process can be divided into two phases: an off-road features subtraction phase and an on-road vehicles detection phase.

3.3.3.1 Phase 1 : Off-Road Features Subtraction

Figure 3.8 shows an example of the complete off-road subtraction phase. First, the U-disparity map is generated from the obstacles disparity map. Then, connected segments algorithm is applied based on the 4-connected neighborhood approach (Figure 3.10 (a)) as follows:

1. Scan the U-disparity map from left to right and from top to bottom
2. The first non zero pixel is taken as the seed point
3. check for the horizontal (left and right) and the vertical (top and bottom) neighbors
4. Join each non-zero neighbor to the seed point
5. Apply recursively the 4-connected neighborhood for each new joined pixel (steps 3 and 4)

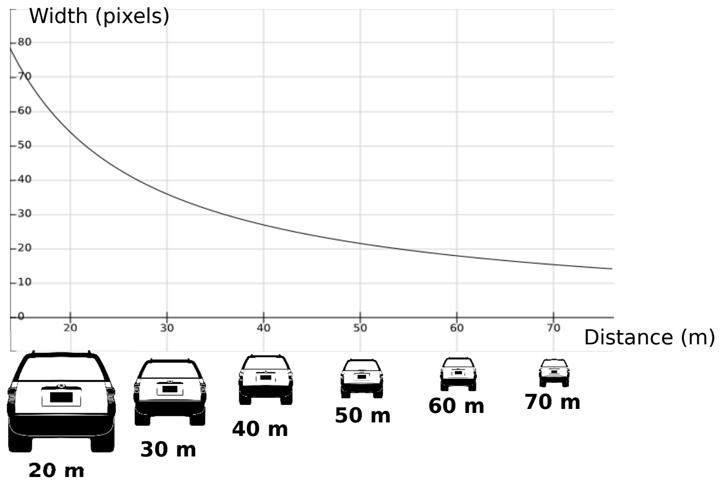


Figure 3.9: Vehicle's Width Variation in Pixels with Distance for a 1.5 m Vehicle's Width

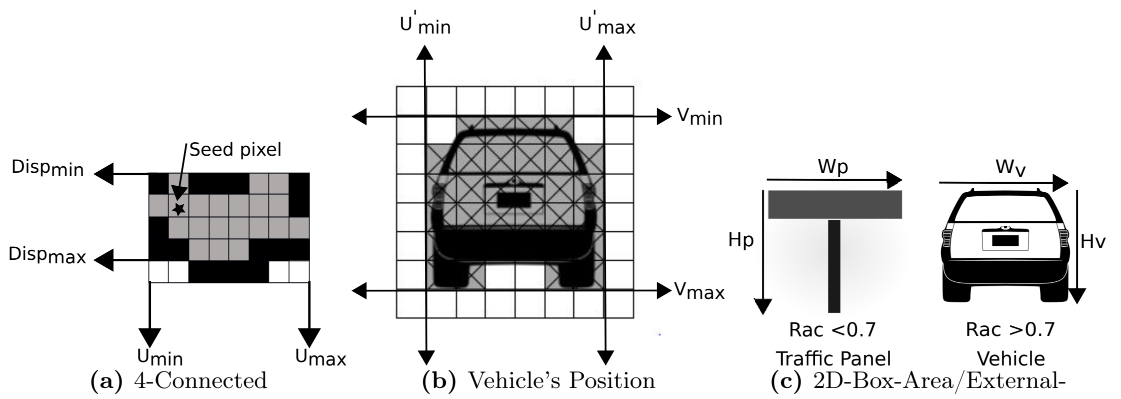
Once the whole U-disparity map is processed, a list of segments is recovered and each one is treated according to its width. To take into consideration the fact that an obstacle looks smaller when it is far and bigger when it is close, the width is measured in function of the disparity. Figure 3.9 illustrates the principle.
Let the pixel $P_i(u, v)$ be a projection on the image plane of a point $P_w(X, Y, Z)$ in the real world plane. To recover X based on the image coordinate system we rely on Eq. (3.16). C_u is the projection of the x coordinate of the camera's optical center in the image plane, Z is the distance and f is the focal distance.

$$X = (u - C_u) \times Z/f \ . \tag{3.16}$$

Let suppose u_{min} and u_{max} are the minimum and maximum vertical limits of a vehicle in the image plane. By using Eq. (3.16), we can determine the width W in meter as shown in Eq.(3.17). Figure 3.9 shows the variation of the width in pixels $(u_{max} - u_{min})$ with distance for a fixed vehicle's width of 1.5 m.

$$\begin{cases} X_{min} = (u_{min} - C_u) \times Z/f \\ X_{max} = (u_{max} - C_u) \times Z/f \end{cases} \Rightarrow W = (X_{max} - X_{min}) = (u_{max} - u_{min}) \times Z/f. \tag{3.17}$$

To remove off-road segments represented by long segments, we have fixed the width interval from 1.5m to 3m and the detection distance from 30m to 70m. Based on this, we have determined the variation interval of the width in pixel to take into consideration. Figure 3.8 shows the result of applying this to remove the off-road features.



(a) 4-Connected Neighborhood **(b)** Vehicle's Position **(c)** 2D-Box-Area/External-Contour Ratio

Figure 3.10: Objects Classification and Vehicles Identification

3.3.3.2 Phase 2 : On-Road Vehicles Detection

Hypothesis Generation To detect on-road vehicles, the list of on-road segments generated is investigated. To distinguish between on-road vehicles and other on-road objects we rely on the geometry features of vehicles. From the previous phase, the vertical position of each segment on the image plane (u_{min}, u_{max}) is recovered (Figure 3.10 (a)) and hence the width in meter is deduced according to Eq. (3.17). Also, the disparity range is determined (Figure 3.10 (a)). Then, for each on-road segment, in the disparity map region limited by u_{min} and u_{max}, we recover all pixels having disparity in the disparity range $[disp_{min}, disp_{max}]$ (Figure 3.10 (a)). The horizontal position is then determined (v_{min}, v_{max}) and the height is found. Also we refine the vertical position (U'_{min}, U'_{max}) as shown in Figure 3.10 (b). Figure 3.11 shows these different steps at the hypothesis generation level.

Hypothesis Verification To select **on-road** vehicles among other on-road obstacles, two tests are applied (Figure 3.11). First,the height in pixels is checked (Eq. 3.17). Vehicles have height between 1.5m and 3.5m, any obstacle with height outside this range is rejected. If the test is verified, the ratio between the 2D box area and the external contour is computed. The external contour is limited by the pixels which have been recovered (Figure 3.10 (b)). The reason of using this ratio as a metric is that, for vehicles, this ratio is supposed to be high which is not the case for panels as shown in Figure 3.10 (c).

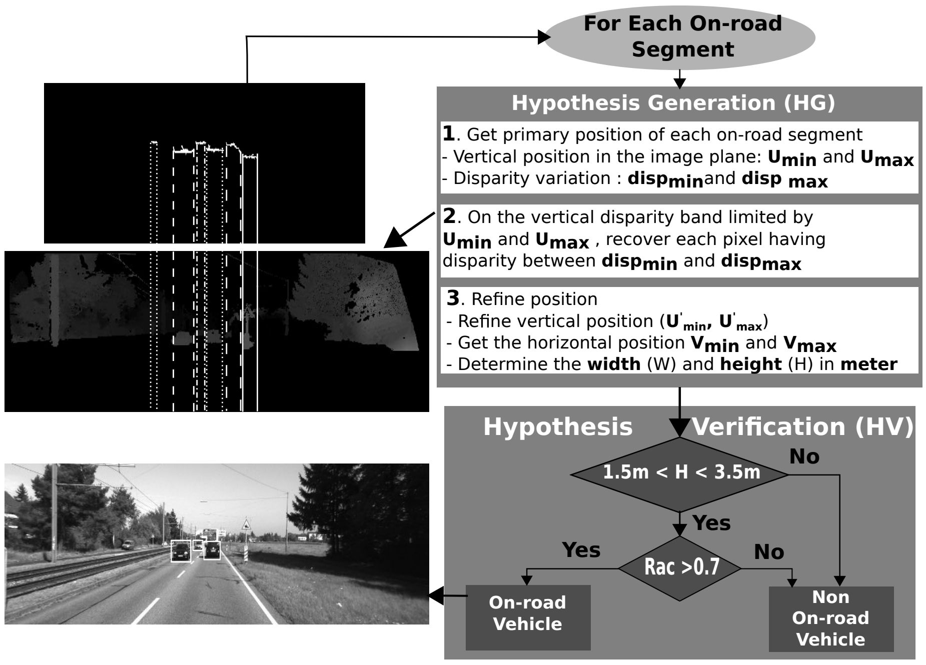

For Each On-road Segment
Hypothesis Generation (HG)
1. Get primary position of each on-road segment
- Vertical position in the image plane: Umin and Umax
- Disparity variation : dispmin and disp max
2. On the vertical disparity band limited by Umin and Umax , recover each pixel having disparity between dispmin and dispmax
3. Refine position
- Refine vertical position (U'min, U'max)
- Get the horizontal position Vmin and Vmax
- Determine the width (W) and height (H) in meter
Hypothesis Verification (HV)
1.5m < H < 3.5m
No
Yes
Rac >0.7
Yes
No
On-road Vehicle
Non On-road Vehicle

Figure 3.11: Vehicles Detection Algorithm: *Phase 2* On-Road Vehicles Detection

3.4 Experimental Results

It is worth noting that the proposed approach performs on-road vehicles detection **without tracking**. The algorithm has been evaluated on KITTI datasets. For each dataset, a list of tracklets is available as the ground truth representing the different objects available on each frame. Each tracklet is presented in velodyne coordinates as a 3D box with its corresponding width, height and length. The algorithm is implemented on a standard PC with an Intel CPU (i5) of 1.8 GHz. The operating system is Ubuntu 14.04. The disparity map as mentionned previously is generated from LibELAS library based on the approach explained on section 3.3.1.

3.4.1 Experimental Design

To evaluate the algorithm, KITTI [101] datasets have been used. The set up of the system employed to grab the stereo images is detailed in [102] and [103]. Table 3.1 give some specifications of the employed cameras.

Before using the tracklets for evaluation, there are some points to take into consid-

Table 3.1: KITTI Datasets Specifications

Feature	Set-up
Baseline	$54cm$
Non-rectified image size	1382×512
Rectified image size	1242×375
$2\times$ grayscale cameras	(FL2-14S3M-C), 1.4 Mp,CCD, global shutter
$2\times$ color cameras	(FL2-14S3C-C), 1.4 Mp, CCD, global shutter

eration. The first one deals with the obstacles type; since our algorithm deals only with vehicles, we need first to filter all non-vehicles tracklets. The second point concerns the detection distance range of our algorithm which is set up from $30m$ to $70m$, hence, we have to take into consideration only tracklets having distance within this range. To cope with these issues the position and motion history data of the tracklets have been used, for instance, to remove non-vehicles objects, the object's type has been employed.

For evaluation, we determine the rate of correct detections or missed ones and false alarms. For better evaluation, these criteria are checked manually when the data provided by the tracklets is not sufficient. To test, we need first to perform a 2D projection of the 3D boxes representing the tracklets. Then, we determine the percentage of intersection between the 2D box of each tracklet taken as a ground truth and the 2D box generated by our algorithm. A detection is then validated once the percentage of intersection is higher than 70%. For evaluation we have selected 3 datasets:

- Dataset 1: High way
- Dataset 2: Urban road
- Dataset 3: Rural road

3.4.2 Analysis of Obtained Results

Table 3.2 shows the evaluation results of the algorithm whiteout the aid of a tracking module on the 3 different datasets. Two hundreds frames have been selected for each dataset. Dataset 1 contains 270 on-road detections, 241 of them have been well detected. In dataset 2 there exists 236 detections, 211 have been well detected, while the rest have been missed. Dataset 3 contains 128 on-road vehicles associated with tracklets, 211 have

Table 3.2: On-road Vehicles Detection Evaluation

Dataset	Correct detections	Missed detections	False alarms
Dataset 1	89.26%	10.74%	8.88%
Dataset 2	89.40%	10.59%	2.96%
Dataset 3	92.19%	7.81%	11.71%

been correctely detected. For evaluation, the three criteria previously presented have been determined for each dataset. During the evaluation, we noticed the absence of redundant detections due to the use of connected component algorithm to recover the segments on the U-disparity map. This solution also increases the accuracy of distance estimation. The results show that high successful detection rate can be achieved as shown in the top left image in Figure 3.12 and the estimated distance reaches 70m. Also the classification task works well since different types of vehicles have been detected like cars and trucks, the bottom left image in Figure 3.12 illustrates this situation.

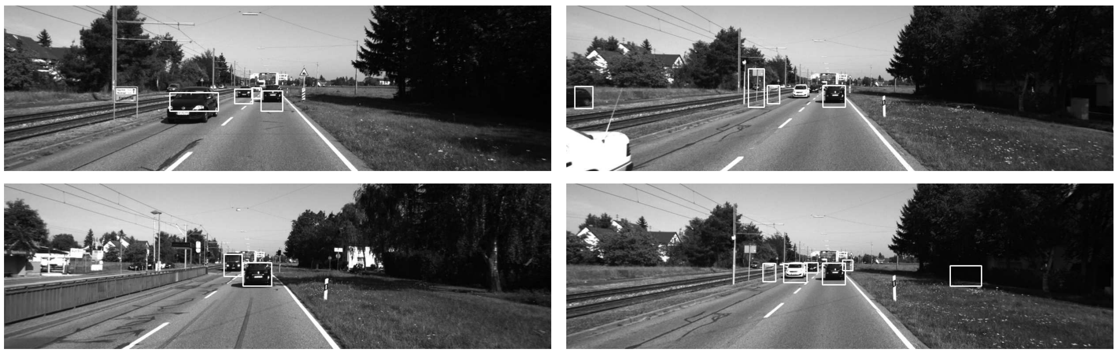


Figure 3.12: Some Detections Results Obtained from KITTI Dataset

Although the algorithm gives sufficient results, it has still some deficiencies. The first point concerns the use of the connected component algorithm to remove off-road features. This technique fails when we deal with on-road vehicles close to high off-road features, in this case, we will loose the vehicles. Also, this approach is highly sensitive to the stereo matching errors since it is related to the U-disparity map generated from the projection of the disparity map. This may increase the false alarms and the rate of missed detections as shown in the top right image of Figure 3.12. The false alarms are generally sidewalks panels and road markers as illustrated on the top and bottom right images in Figure 3.12. They are usually generated because of the stereo matching errors that affect directly the scene segmentation task and hence the on-road vehicles detection phase. This can be solved by merging the proposed algorithm with a lane marking detection module.

We have compared the performances of our algorithm with some contributions cited in Table 3.3 in terms of the Farthest Detection Distance (FDD), the Correct Detections Rate (CDR) and the False Detections FD).

Table 3.3: Comparison of our Algorithm with other On-Road Vehicles Detection Algorithms

Article	FDD	CDR	FD	Detection Type
Sun et al. [104]	32×32 image region	98.5%	2%	Rear
Alonso et al. [105]	-	92.63%	3.63%	Rear and front
Bergmiler et al. [106]	-	83.12%	16.7%	Rear
Southall et al. [107]	$40m$	99%	1.7%	Single lane rear
Chang et Cho [108]	32×32 image region	99%	12%	Rear
Kowsari et al. [109]	$120m$	98.6%	13%	Multi view
Our algorithm	$70m$	90.28%	7.85%	Multi view

3.5 Discussion

In this chapter, a stereo vision based scene segmentation and on-road vehicles detection algorithm has been proposed. While a variety of vehicles detection algorithms exist in the literature, the proposed algorithm provides improvements to cope with some crucial issues. The first one concerns the off-road features subtraction. It is an important point to deal with when on-road objects are targeted for detection. We proposed to use the connected segments labeling algorithm which has been also used to recover the on-road segments instead of the traditional Hough transform for better and fast obstacles detection. For the second improvement, width and height of objects are not measured in meter but rather in pixels in function of the disparity. This solution increases the detection precision and distance estimation. Also, some cues have been presented for objects classification like the ratio between the 2D box area and the external contour of the object. The algorithm has been evaluated on the KITTI vision dataset and the experimental results show that it can detect the most on-road vehicles and determine their distance up to $70m$.

4

Kernel-Level Optimizations on CPU

Vectorization and Shared Memory Parallelization

This chapter concentrates on optimizing and evaluating the stereo matching algorithm at kernel-level on CPU-based systems. Vectorization and shared memory parallelization are targeted based on different approaches and programming languages. The stereo matching algorithm is evaluated from the execution time aspect. The main contributions of this chapter are summarized in the following paragraph.

We start by analyzing the algorithm to **(1)** identify the most time consuming kernels. After that, **(2)** we propose to optimize the stereo matching approach at algorithmic level to fit the parallelization requirements such as having regular dense operations. We then start parallelizing the algorithm with various approaches at different levels. The first approach consist on **(3)** parallelizing the cost function (SAD) of the stereo matching algorithm through vectorization via `SIMD` instructions. Then, **(4)** we optimize the algorithm at global level through shared memory parallelization via `OpenMP` on multi-core systems. We then **(5)** combine both vectorization (`SIMD`) and shared memory parallelization (`OpenMP`) for a maximum performance. After that, **(6)** we use `OpenACC` directives on CPU to parallelize on multi-core systems. As a last approach, **(7)** we use a template-based technique referred as `NT2` to parallelize the algorithm on CPU-based systems. Finally, **(8)** an evaluation of the obtained results is performed. We discuss the contributions of each approach from different aspects such as the obtained performance and the required productivity. We also present their limitations and the issues met during our experiments.

4.1 CPU Based Stereo Matching Algorithm

In this section, the original CPU-based stereo matching algorithm is described. We analyze the algorithm to determine potential parallelism. Bottlenecks, or time consuming functions are identified through profiling before optimizing the algorithm. We also present some improvements applied to the algorithm for better and efficient parallelization.

4.1.1 ELAS : Original CPU Based Algorithm

ELAS algorithm [94] is based on a generative probabilistic model to generate a dense disparity map. The approach relies on a set of support points to compute a sparse disparity map. The latter is then used as a depth prior to enable efficient sampling of disparities in a dense fashion (refer to chapter 3, section 3.3.1 for more details). While ELAS generates a disparity map with an important precision, it is not fast enough to respond to real time requirements. Indeed, some optimizations at algorithmic level are required before focusing on accelerating the algorithm.

It is well known that parallelization is possible whenever regular operations are present, i.e, the same operation is applied to a set of data such as pixels within an image. Stereo matching algorithm responds well to this criteria since we compute the disparity for all pixels based on some **matching cost** function also called *similarity measure*. The later defines the *core* of many stereo matching algorithms.

Based on the required data to compute the cost function, we distinguish two categories: *window-based* and *pixel-based* cost functions. Pixel-based cost function is the simplest matching costs which assumes constant intensities at matching image locations. In other words, it depends only on values L_i (considered pixel's intensity in the left image) and $R_i + d$ (candidate pixel's intensity in the right image). Common pixel-based matching costs include Absolute Differences (AD) and Squared Differences (SD) presented in chapter 3. Window-based cost functions rely on the neighboring pixels within a specific window of $n \times n$ pixels of L_i and $R_i - d$. Common window-based matching costs include the Sum of Absolute or Squared Differences (SAD / SSD) and Normalized Cross Correlation (NCC). Figure 4.1 illustrates these two approaches of stereo matching. The red lines both represent the epipolar line (pixels) which is the same since the images are rectified. The green pixels are all pixels candidates in the

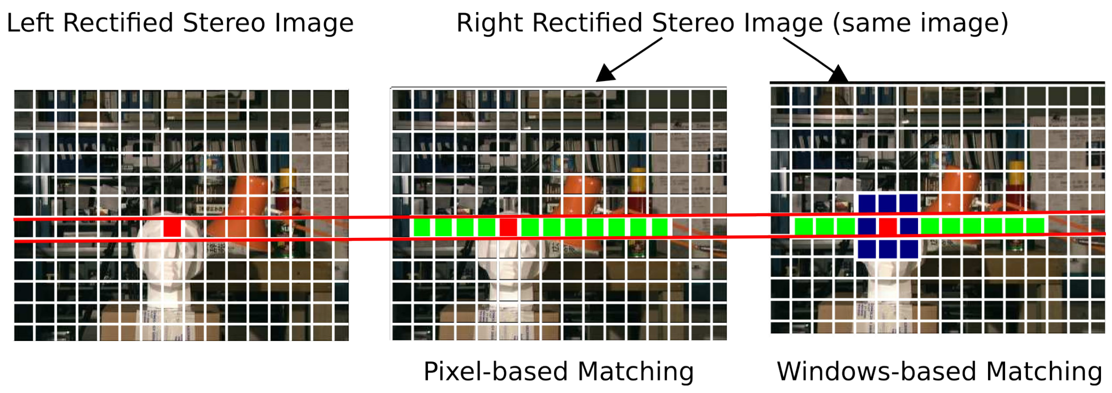

Left Rectified Stereo Image
Right Rectified Stereo Image (same image)
Pixel-based Matching
Windows-based Matching

Figure 4.1: Pixel-based vs Windows-based Stereo Matching

right image– here the left image is taken as a reference. The boundaries of the green strip is the same on both approaches, however, in window-based technique we also use the neighboring pixels depicted with the blue window in Figure 4.1.

In ELAS algorithm, window-based cost function is used. However, pixels' intensity is not directly used to compute the cost function. A descriptor of 16 elements – extracted from Sobel horizontal and vertical gradients, is associated to each pixel. The **SAD** cost function is applied to the left and right descriptors. To compute the disparity of each pixel, we first compute the boundaries –minimum and maximum, of the research region in the right image. Based on some prior data from support points' disparity, we determine these boundaries for each pixel. Hence, we do not have the same research region for all pixels withing the same line.

Algorithm 1 illustrates part of the stereo matching process in ELAS algorithm for the left image. The algorithm goes through all the generated triangles from the Delaunay triangulation. For each triangle, slight lines from corners A to B, and B to C and their corresponding data (corners, planes ...) are respectively used to estimated the disparity. The pseudo code 1 is **only** for the first slight line; $A \to B$. The same algorithm is used for the second part; $B \to C$. For each slight, the horizontal range (over the colons) is computed from the `y` coordinates of the corners (line 2 in Algorithm 1). Concerning the vertical range (over the rows), we use line's equation parameters of both lines `AC` and `BC` (lines 3, 4). After that, we compute the disparity range over which we look for the homologue pixel in the other image. To do so, we use the triangle's plane parameters; `PA, PB, PC` and `PD`. Each triangle plane can be presented with the following Equation (4.1) :

$$PD = PA \times col + PB \times row + PC \tag{4.1}$$

Since parameters `PA`, `PB` and `PC` are already known from the Delaunay triangulation, we can compute `PD` for pixel P_1 located at position (row_1, col_1) as follow:

$$PD_1 = PA \times col_1 + PB \times row_1 + PC \tag{4.2}$$

Algorithm 1 Original Sequential Stereo Matching in ELAS (CDDL)

```
Require: TRI[M], LD[H x W x 16], RD[H x W x 16], PRIO[[256]
 1: for M=0 to m do          ▷ for all triangles do: treat A→B(below), then B→C(same)
 2:     for col=A.y to B.y do          ▷ A.y, B.y are y coordinates of corners A & B
 3:         row1=$a_1$×col+$b_1$            ▷ use AC line'equation parameters ($a_1$, $b_1$)
 4:         row2=$a_2$×col+$b_2$            ▷ use BC line'equation parameters ($a_2$, $b_2$)
 5:         for row=min(row1,row2) to max(row1,row2) do
 6:             DMIN=f1(PA,PB,PC)          ▷ PA, PB, PC are triangle planes parameters
 7:             DMAX=f2(PA,PB,PC)
 8:             for d=DMIN to DMAX do                  ▷ disparity reasearch distance
 9:                 SAD = PRIOR[|d-PD|]       ▷ add prior, PD=PA×col+PB×row+PC
10:                 for i=0 to 15 do              ▷ go through pixel's descriptor elements
11:                     SAD += abs(LD[row][col][i] - RD[row][col-d][i])
12:                 end for
13:                 if SAD<$SAD_{min}$ then
14:                     $SAD_{min}$ = SAD
15:                     disparity = d
16:                 end if
17:             end for
18:             D[row][col] = disparity
19:         end for
20:     end for
21: end for
```

We then have the necessary data to compute the disparity range (lines 6, 7) as illustrated in Equation 4.3. A radius of `r` is taken into consideration, it has been set to `2` in `ELAS` code. the maximum disparity we can get is set to `256` (disp_{max})

$$\begin{aligned} \texttt{DMIN = max(PD-r, 0)} &\rightarrow \text{f1 in Algorithm 1 line 6} \\ \texttt{DMAX = min(PD+r, disp}_{\texttt{max}}\texttt{)} &\rightarrow \text{f2 in Algorithm 1, line 7} \end{aligned} \tag{4.3}$$

Once the disparity range is computed, at each disparity, we estimate the disparity. We start by adding the prior data. The prior has been presented in the previous chapter. Listing 4.1 illustrates how to compute the prior. We notice a set of parameters which are employed. These parameters have been set in `ELAS` code to KITTI dataset as shown in the comment. They have been also set to Middlebury dataset.

```
for (int delta=0; delta<disp_max; delta++)
// for KITTI dataset: gamma=5, sigma= 1, beta=0.02
        PRIOR[delta] = (int)((-log(gamma+exp(-delta*delta/2*sigma*sigma))+log(gamma))/beta);
```

Listing 4.1: PRIOR Computation in ELAS

4.1.2 A-ELAS: Adapt to Parallelize

From the previous discussion of the original `ELAS` stereo matching algorithm, we notice two main issues from the parallelization point of view. First, the global loop iterates over the triangles. The number of triangles is not known from the beginning since it depends on the number of the supports points which is itself not fixed. Hence, we cannot estimate the required performance precisely and whether the available hardware can respond to these requirements. Secondly, each pixel has its own research region of the disparity (`DMAX-DMIN`), hence, when parallelizing, it may happen that threads do not receive the same amount of work which is not desired if we want to achieve work balance for a maximum performance.

In order to maximize the potential parallelism, we propose to improve the algorithm. The idea consists on having the same boundaries of disparity research region of all pixels on the same image's line. These boundaries are computed separately before performing the stereo matching. Algorithm 2 gives the pseudo code of this proposed approach applied both to `CDDL` and `CDDR` functions. We notice that we do have the same algorithm for both function expect when the SAD is computed between the left descriptor(`LD`) and the right one (`RD`) as illustrated in pseudo code 2: line 7 for the forward left disparity and line 9 for the backward right disparity. It is worth noting that even if in this algorithm we presented the forward and backward disparities in the same code, in our

Algorithm 2 Adapted Sequential Stereo Matching (CDDL/CDDR)

```
Require: LD[H x W x 16], RD[H x W x 16], PRIOR[256], DMIN[H], DMAX[H],
    PD[H x W]
 1: for row=2 to H-2 do
 2:     for col=2 to W-2 do
 3:         for d=DMIN[row] to DMAX[row] do          ▷ disparity reasearch distance
 4:             SAD = PRIOR[|d-PD[row][col]|]
 5:             for i=0 to 15 do                     ▷ go through pixel's descriptor elements
 6:                 if left then                     ▷ left disparity (forward)
 7:                     SAD += abs(LD[row][col][i] - RD[row][col-d][i])
 8:                 else                             ▷ right disparity (backward)
 9:                     SAD += abs(LR[row][col][i] - LD[row][col+d][i])
10:                 end if
11:             end for
12:             if SAD<$SAD_{min}$ then
13:                 $SAD_{min}$ = SAD
14:                 disparity = d
15:             end if
16:         end for
17:         D[row][col] = disparity
18:     end for
19: end for
```

implementation, we developed 2 independent functions.

In this algorithm we go through all image's pixels (lines 1, 2). However, we notice that we have the same disparity research boundaries (DMIN, DMAX) for all pixels in the same image's line which have been computed as before (Equation 4.3). These boundaries, PD and the PRIOR are computed in a separate function in contrast to original ELAS (Algorithm 1)

This optimization allows better use of cache memory since all pixels in each line loaded from right image are used by all pixels in the left image within the same line. The efficiency of this improvement will be discussed also in the next chapter with shared memory in GPUs. Also it gives more regular operations to facilitate the parallelization process. We call this improved algorithm **A-ELAS**, **A** is for Adaptive. We could then have the following algorithm of dense stereo matching with several choices of parallelization.

After the CPU based `A-ELAS` stereo matching is implemented, we will investigate possible parallelization or vectorization approaches. We have several choices:

1. Parallelize the first and the second for loops.
2. Parallelize the third for loop only.
3. Parallelize the fourth for loop only.
4. Parallelize the first, second and third for loops.
5. Parallelize the fourth loop only.

At first glance, the fourth option seems to be the best as it can parallelize the CPU solution mostly. In this case, an important number of iterations need to be parallelized. However, in this chapter we target parallelization in multi-core systems with only few threads. Hence, we do not have enough threads to accelerate the algorithm. Thus, this option should be denied.

Secondly, for the second and the third options, generally image size (`H` or `W`) is much greater than descriptor size (`16`) or disparity range(`DMAX-DMIN`) , so parallelizing the computation on each pixel is not a wise strategy.

Based on the above discussion, the rest we can do is the first and/or the fifth options. The first option seems to be the best solution where all pixels can run concurrently (depending on the number of threads) and within each pixel the procedure is sequential. This option allows then parallelization at global level. The fifth choice seems also to work since the size of SIMD registers allows to hold a descriptor and hence perform the SAD at once. SSE registers size is 128 bits which is enough to hold one descriptor ($16 \times 8 - bits$) while in NEON since registers size is 64 bits, 2 registers are concatenated in this case. This option is performed at low level. Hence, if we combine both first and second options we expect to achieve an important performance.

4.1.3 A-ELAS : Profiling and Analysis

To determine the most time consuming functions, we profiled the algorithm with `gprof` profiler by adding the `-pg` compiling option. The algorithm has been profiled in the four selected platforms on KITTI dataset. Figure.4.2 shows the obtained results.

- `CSM` (ComputeSupportMatches) Computes supports points' disparity.

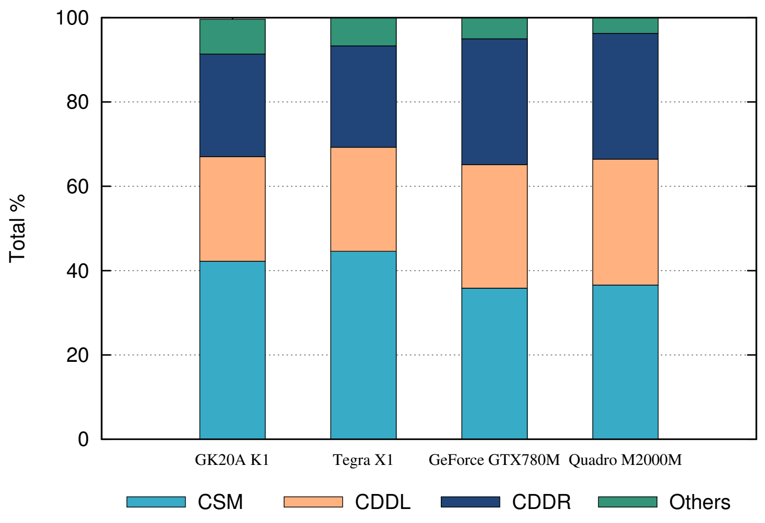


Figure 4.2: A-ELAS Profiling Results on KITTI Dataset.

- `CDDL` (ComputeDenseDisparityLeft) Computes dense disparity of left image.
- `CDDR` (ComputeDenseDisparityRight) Computes dense disparity of right image.
- `Others` Include pre-processing and post-processing.

The most important observation concerns the 3 functions; `CSM`, `CDDL` and `CDDR`, which take all together more than **90%** of the total algorithm's execution time on all platforms. By others, we mean *pre-processing* such as Sobel filter, and *post-processing* as left and right consistency check. According to these results, to accelerate the stereo matching algorithm, we need to focus on `CSM`, `CDDL` and `CDDR` functions.

4.2 Experimental Design

In our experiments, we use KITTI [101] raw stereo dataset of 1242x375 pixels to discuss each proposed optimization. Then, we will apply the obtained optimized algorithm on 9 image pairs with different resolutions from Middlebury stereo datasets [1]. Table 4.1 depicts the CPU hardware specifications of the platforms employed in these experiments.

It is worth mentioning that ARMv8 in NVIDIA Tegra X1 is packing four high performance Cortex-A57 big cores and four high efficiency Cortex-A53 little cores. The big cores are fast and use more power, but the little cores are great for background

Table 4.1: Employed Platforms CPU Specification

Specification	i7-6700HQ	i7-4600M	ARMv7 rev3	ARMv8 rev1
#cores	4	2	4	8
#threads	8	4	4	4
Max CPU Clock	3.50GHz	3.60GHz	2.32GHz	1.73GHz
L1 Cache (KB)	256	128	32	2048(A57), 512(A53)
L2 Cache (KB)	1024	512	2048	32(A57), 32(A53)
L3 Cache (KB)	6144	4096	_	_
RAM Size (GB)	16	16	2	4
SIMD	sse[1]	sse[1]	neon	asimd[2]
fp/fpu/vfp	fpu	fpu	vfpv4, vfpv3	fp

[1] sse, sse2, ssse3, sse4_1, sse4_2

[2] advanced simd

processing and are much more power-efficient. Most chips that use this eight-core configuration are tied together using a system from ARM called big.LITTLE. Instead of using ARM's method to control the eight cores, NVIDIA is using cluster migration with a custom cache coherence system to move data between the two islands. Under this model, the OS scheduler only sees one cluster (either big or little) at a time. Actually, Tegra X1 can only run processes on one set of cores at a time, but the data can be moved back and forth between the big cores and the small cores.

4.3 SIMD Implementation

The first approach to accelerate the stereo matching algorithm consists on using the `SIMD` intrinsics. The idea comes from the fact that pixel's descriptor in ELAS algorithm includes 16 elements of 8-bits each corresponding to the size of `SSE` registers (128-bits) and double the size of `NEON` registers. Actually, in the original ELAS [94], the authors proposed a descriptor of 50 elements, 25 Sobel horizontal gradients concatenated with 25 Sobel vertical gradients. Each 25 gradients are recovered from a patch of size 5×5 around the considered pixel. For performance purpose, the authors decided to use only 16 elements while writing the code.

4.3.1 Intel(SSE) vs ARM(NEON)

On fixed platforms with Intel processor, SSE instructions set is employed. On embedded platforms with ARM processor, NEON instructions set is used. Adaptations have been made to port code from Intel to ARM processor. First, code has been rewritten with NEON intrinsics since the SIMD instruction set is not the same. Second, There are more advanced instructions like SAD operation in SSE which are not available in NEON and need to be implemented explicitly by the programmer.

The part of the algorithm affected with this optimization is the match energy computation in the 3 functions (CDDL, CDDR, CSM). The cost matching function used in ELAS algorithm is the Sum of Absolute Differences (SAD). The later is computed by using the descriptor of the considered pixel in the left image and its corresponding descriptor in the right image. Each descriptor consists of 16 elements of 8-bits each. This configuration fits well SIMD registers. In Intel processors, SSE registers are of 128-bits size which can then hold one descriptor. In ARM processors, NEON registers' size is 64-bits, in this case two registers are concatenated together to hold one descriptor at once. The SAD operation is implemented in SSE with `_mm_sad_epu8` intrinsic. However, in NEON, `vabdq_u8` intrinsic is provided which computes only the absolute difference between registers elements. Hence, we have implemented the SAD version in NEON by using the available instructions.

```
__m128i sad_tmp = _mm_sad_epu8(x1,x2);
uint32_t sad   = _mm_extract_epi16(sad_tmp,0) + _mm_extract_epi16(sad_tmp,4);
```

Listing 4.2: SSE Version of SAD

Listing 4.2 shows the SSE intrinsics employed to compute the SAD between two descriptors of 16-byte elements each. Only **two** SSE intrinsics are employed. First, `__m128i _mm_sad_epu8(__m128i a, __m128i b)` is used to compute the absolute differences (AD) of packed unsigned 8-bit integers in a and b. Figure 4.3 illustrates the function of this intrinsic. It **horizontally sums** each consecutive 8 differences to produce two unsigned 16-bit integers, and pack these unsigned 16-bit integers in the low 16 bits of 64-bit elements in the output register. Second, `int _mm_extract_epi16 (__m128i a, int imm8)` is used to extract a 16-bit integer from a, selected with imm8, and store the result in the lower element of a scalar output. It is called twice to extract

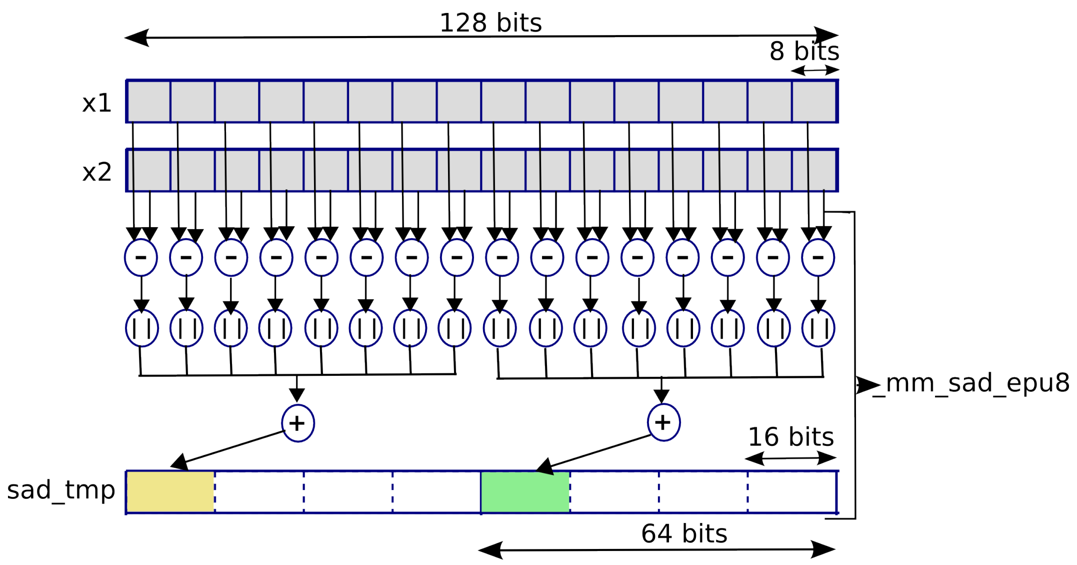


Figure 4.3: Functional diagram of SSE SAD Intrinsic

the two generated 16-bit of `_mm_sad_epu8` instruction. We then sum the two extracted values to obtain the final SAD.

Listing 4.3 shows the `NEON` snippet code to compute the SAD between two descriptors of 16-byte elements each. **Five** `NEON` intrinsics are employed. Figure 4.4 illustrates the function of each intrinsic.

```
uint64x2_t sad_tmp = vabdq_u8((uint8x16_t)x1, (uint8x16_t)x2);
sad_tmp = vpaddlq_u8((uint8x16_t)sad_tmp);
sad_tmp = vpaddlq_u16((uint16x8_t)sad_tmp);
sad_tmp = vpaddlq_u32((uint32x4_t)sad_tmp);
uint32_t sad = vgetq_lane_u64((uint64x2_t)sad_tmp, 0) + vgetq_lane_u64((uint64x2_t)sad_tmp, 1);
```

Listing 4.3: NEON Version of SAD

First, `uint8x16_t vabdq_u8(uint8x16_t a, uint8x16_t b)` is employed to compute the absolute differences (AD) of packed unsigned 8-bit integers in a and b. It subtracts the elements of vector b from the corresponding elements of vector a and stores the absolute values of the result into the elements of the destination vector. Second, `uint16x8_t vpaddlq_u8(uint8x16_t a)` is used to perform a long pairwise addition. It adds adjacent pairs of elements of a vector, sign or zero extends the results to twice their original width (16 bits in this case), and places the final results in the destination vector. Third, `uint32x4_t vpaddlq_u16(uint16x8_t a)` does the same thing but works on 16-bits data and store the results in a 32-bits vector. Fourth, `uint64x2_t`

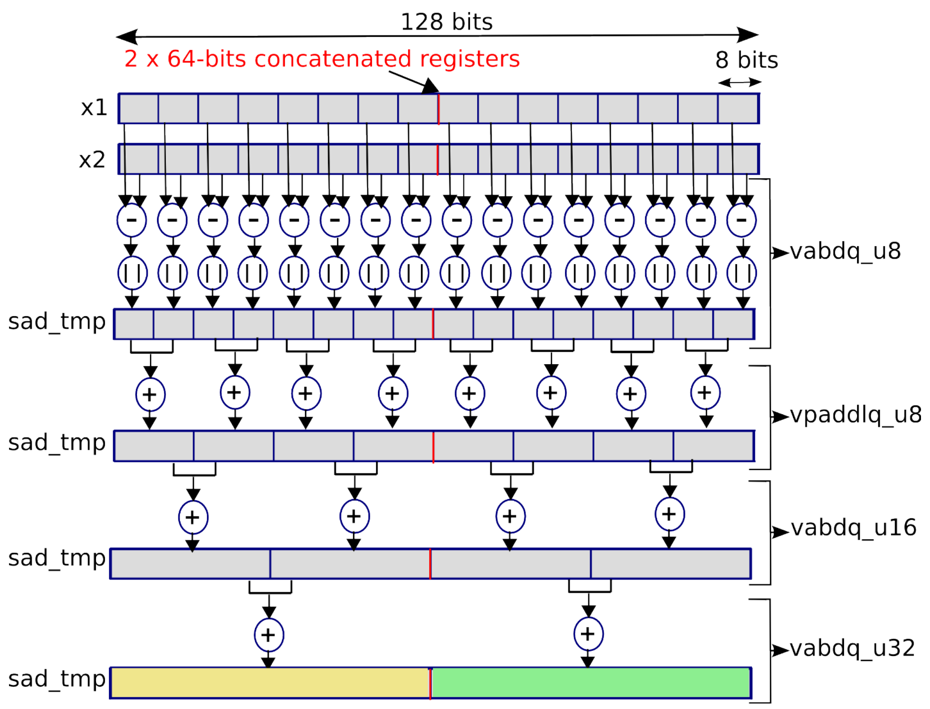


Figure 4.4: Functional diagram of NEON Equivalent SAD Intrinsics

`vpaddlq_u16(uint32x4_t a)` adds the 32-bits adjacent data in a vector and store the results in a 64-bits vector. At this level, we get the same output as the `_mm_sad_epu8` SSE intrinsic. Finally, to extract the two elements, `uint64_t vgetq_lane_u64(uint64x2_t vec, __constrange(0,1) int lane)` is used. We then sum the two extracted values to obtain the final SAD.

```
// ARMv7
mul r0, r0, r1 //Scalar
vmul d0, d0, d1 //SIMD
// ARMv8
mul x0, x0, x1 //Scalar
mul v0.u8, v0.u8, v1.u8 //SIMD
)
```

Listing 4.4: Scalar vs SIMD in ARMv7 and ARMv8

4.3.2 ARMv7(Cortex-A15) vs ARMv8(Cortex-A57)

There are NEON instruction sets for both AArch32 (equivalent to the ARMv7 NEON instructions) and for AArch64. Both can be used to significantly accelerate repetitive

Table 4.2: `SIMD` Compiling Options

Processor	Compiling Options
Intel Core i7-4600M 2.90GHz	`-O3 -ffast-math -msse4.2`
Intel Core i7-6700HQ 2.60GHz	`-O3 -ffast-math -msse4.2`
Jetson TK1 ARMv7 rev 3 (v7l)	`-O3 -ffast-math -mfpu=neon+vfp4`
Tegra X1 ARMv8 rev 1 (v8l)	`-O3 -ffast-math -mcpu=cortex-a57+simd+fp`

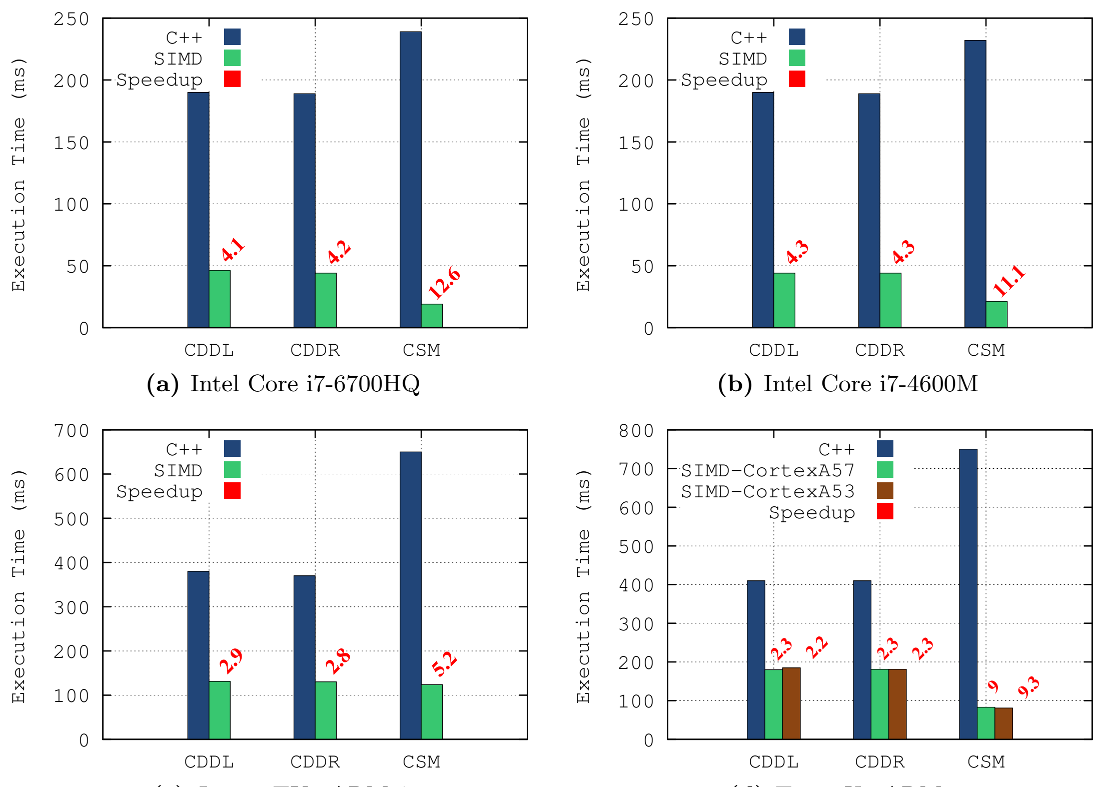
operations on large data sets. The `NEON` architecture for `AArch64` uses 32 × 128-bit register, twice as many as for `ARMv7`. These are the same registers used by the floating-point instructions. The compiler is free to use any `NEON/VFP` registers for floating-point values or `NEON` data at any point in the code. In `ARMv8`, the same mnemonics as for general purpose registers are used as shown in Listing 4.4. The only way to distinguish between scalar and SIMD instruction in assembly code of an `ARMv8` is to check the registers employed. Both floating-point and `NEON` are required in all standard `ARMv8` implementations.

4.3.3 Obtained Results

By following the aforementioned details, the cost matching function (SAD) has been programmed with `SIMD` instructions in selected platforms and applied to the three functions; `CSM, CDDL` and `CDDR`. Table 4.2 shows the compiling option employed. To compile in Cortex-A53, we just need to change the compiling option `-mcpu=cortex-a57` shown in Table 4.2 to `-mcpu=cortex-a53`.

Figure 4.5 shows the scalar version (`C++`) versus the `SIMD` one. The code has been profiled in KITTI dataset(1242x375) by using `gprof` profiler with `-pg` compiling option. The red labels above the green bars present the obtained speedup of the `SIMD` version with respect to the scalar one.

If we compare the scalar version, we notice that we have the same execution time in the first two fixed platforms since we have the same processor family (`Intel core i7`) with different model and hardware specifications. However, we have the same `SIMD` enabled flags. In these two platforms, we obtain a speedup of more that `4` in `CDDL` and `CDDR` functions and a speedup of almost `12` with `CSM` function.



(a) Intel Core i7-6700HQ **(b)** Intel Core i7-4600M

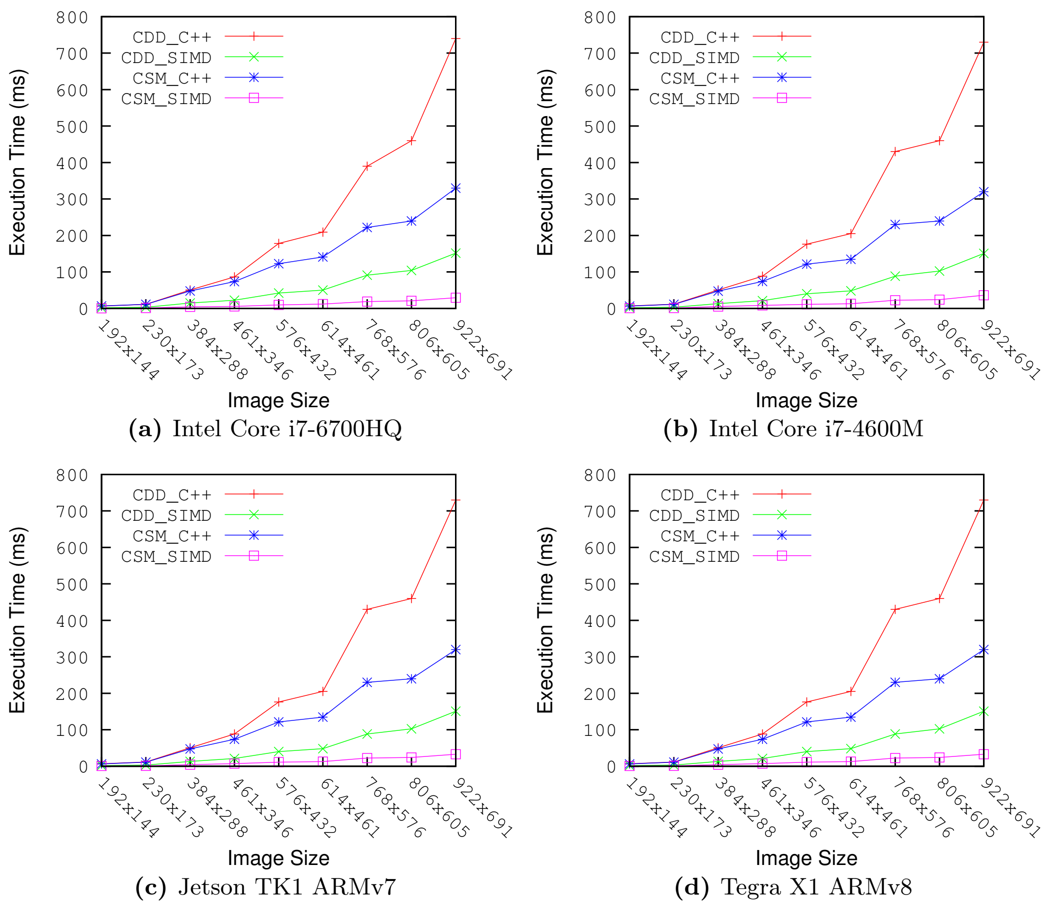
(c) Jetson TK1 ARMv7 **(d)** Tegra X1 ARMv8

Figure 4.5: C++ vs SIMD Execution Time on KITTI(1242x375) dataset.

In Jetson TK1 platform, `CDDL` and `CDDR` are `3` times faster with NEON. A speedup of `5` is achieved with `CSM` function. In Tegra X1, the `NEON` code has been executed and profiled on both ARM processors, `CortexA57` and `CortexA53`. Figure 4.5 shows that there is no difference in terms of execution time. While a speedup of `2.2` is achieved with `CDDL` and `CDDR`, `CSM` function runs `9` times faster in `NEON` compared to its scalar(`C++`) version. The difference in speedup between `Intel` and ARM processors is mainly related to the intrinsics employed to compute the SAD. As explained before, `SSE` instructions set provides an intrinsics to compute the SAD in contrast to `NEON` which provides an intrinsic to compute only the absolute differences. Table 4.3 depicts the latency and throughput of the employed intrinsics. If we compute the total latency in `Intel` processor required to compute the SAD, we get $3 + 2 \times 3 = 9$ cycles. In `ARM` processors, we get $3 + 3 \times 3 + 5 \times 2 = 22$ cycles. Hence, more cycles are required in `ARM` processors to compute the SAD compared to `Intel` processors.

Table 4.3: SIMD Intrinsics Latency

Intrinsic	Execution Latency	Execution throughput
`_mm_sad_epu8()`	3	1
`_mm_extract_epi16()`	3	1
`vabdq_u8`	3	1
`vpaddlq_u8`	3	2
`vpaddlq_u16`	3	2
`vpaddlq_u32`	3	2
`vgetq_lane_u64`	5	1



(a) Intel Core i7-6700HQ

(b) Intel Core i7-4600M

(c) Jetson TK1 ARMv7

(d) Tegra X1 ARMv8

Figure 4.6: C++ vs SSE Execution Time on Middlebury dataset.

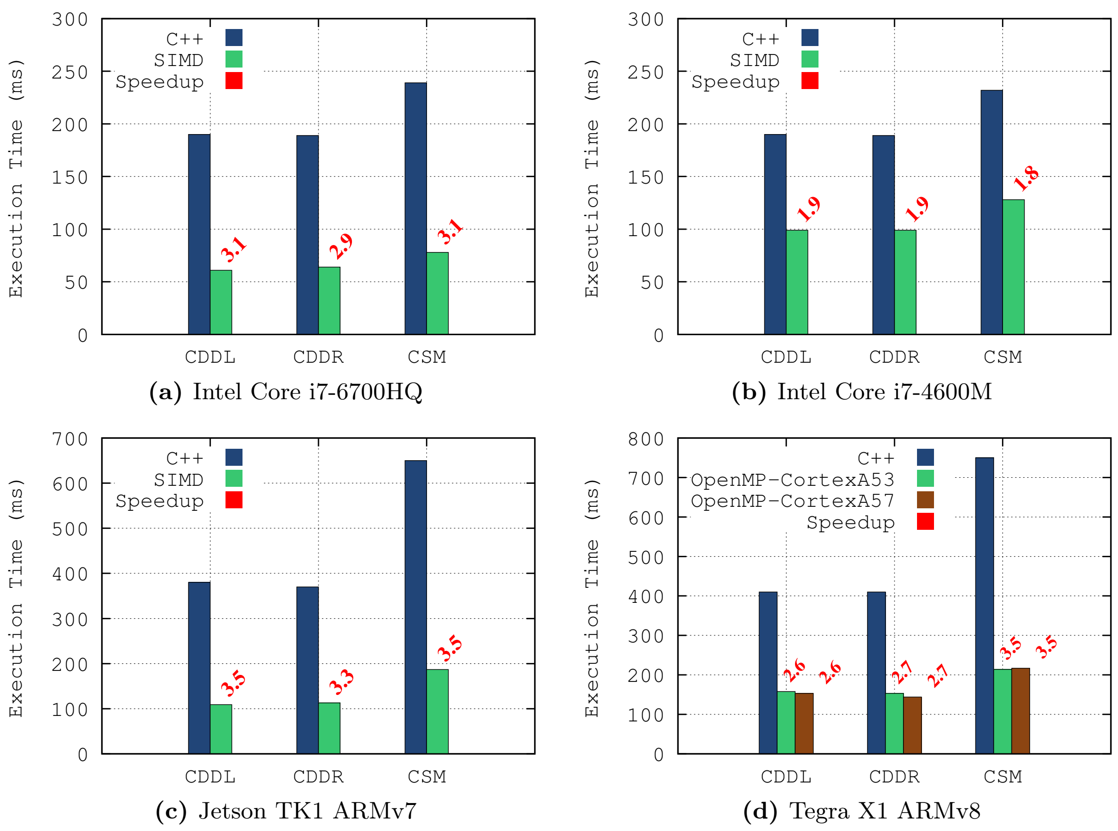
Figure 4.6 depicts the execution time of the scalar(`C++`) and `SIMD` versions on different image size. Middlebury dataset is used in this experiment. Since execution time of CDDL and CDDR is quite the same, only CDDL results are depicted in Figure 4.6 and referred as CDD. the results of CSM function are also depicted. We notice an important speedup with SIMD of both functions (CDD, CSM) on all architectures. While scalar curves in all cases increase rapidly with image size, SIMD curves increase in a slower fashion.

Algorithm 3 Parallel Stereo Matching of A-ELAS (CDDL/CDDR) with OpenMP

```
Require: LD[H x W x 16], RD[H x W x 16], PRIOR[256], DMIN[H], DMAX[H],
    PD[H x W]
 1: for row=2 to H-2 parallel do                         ▷ use #pragma omp parallel for
 2:     for col=2 to W-2 parallel do                     ▷ use #pragma omp parallel for
 3:         for d=DMIN[row] to DMAX[row] do              ▷ disparity reasearch distance
 4:             SAD = PRIOR[|d-PD[row][col]|]
 5:             for i=0 to 15 do                 ▷ go through pixel's descriptor elements
 6:                 if left then                           ▷ left disparity (forward)
 7:                     SAD += abs(LD[row][col][i] - RD[row][col-d][i])
 8:                 else                                  ▷ right disparity (backward)
 9:                     SAD += abs(LR[row][col][i] - LD[row][col+d][i])
10:                 end if
11:             end for
12:             if SAD<$SAD_{min}$ then
13:                 $SAD_{min}$ = SAD
14:                 disparity = d
15:             end if
16:         end for
17:         D[row][col] = disparity
18:     end for
19: end for
```

4.4 OpenMP Implementation

Open specifications for Multi Processing (OpenMP) is an Application Program Interface (API) which allows explicit and portable model for multi-threaded and shared memory parallelization. OpenMP is an extension to C/C++ and Fortran languages. The first OpenMP standard was released in 1997, OpenMP 3.0 was finalized in 2008. It supports



(a) Intel Core i7-6700HQ **(b)** Intel Core i7-4600M **(c)** Jetson TK1 ARMv7 **(d)** Tegra X1 ARMv8

Figure 4.7: C++ vs OpenMP Execution Time on KITTI(1242x375) Dataset.

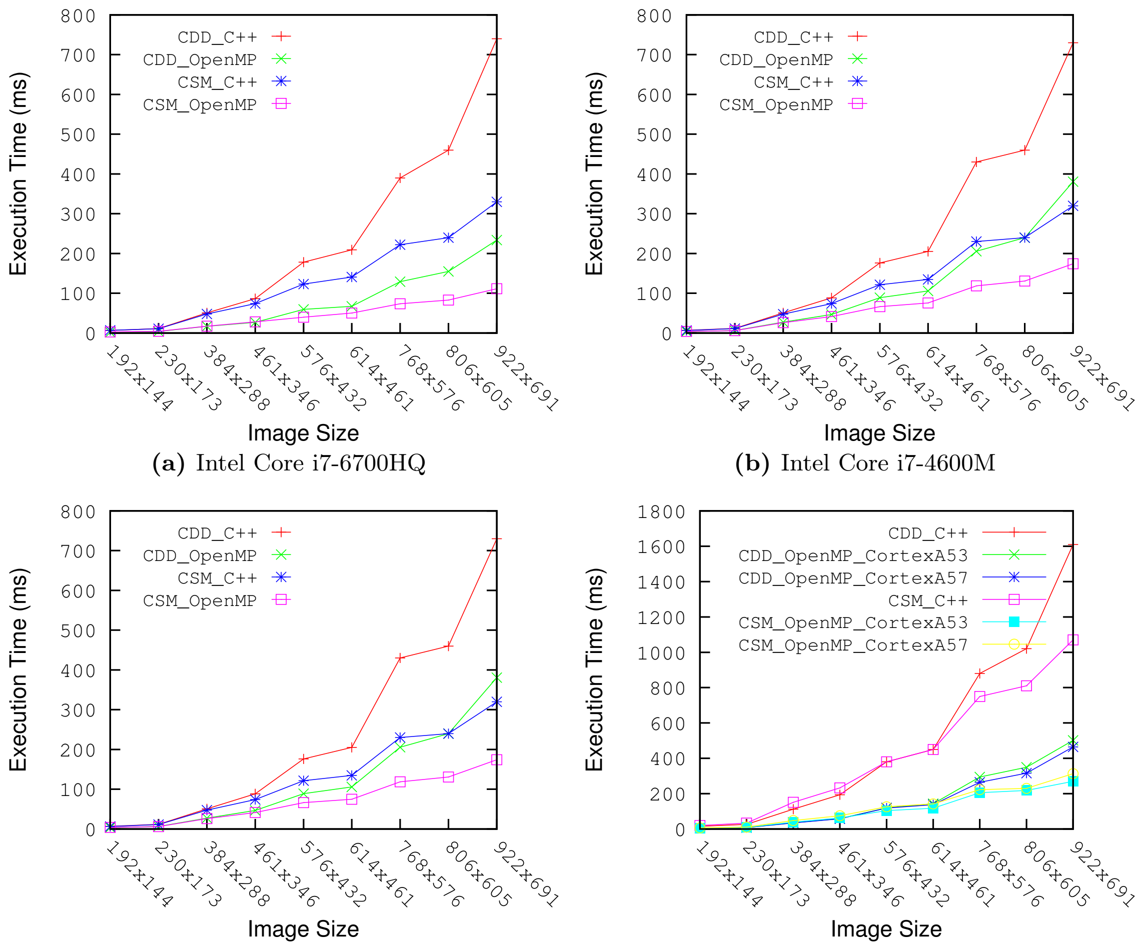
a variety of machines from multi/many-core to GPUs in the last release (4.0). In the following experiments , `OpenMP.3.1` version is used whith `gcc.4.8.5` compiler.

4.4.1 Obtained Results

We use `OpenMP` directives to parallelize the three bottlenecks identified previously –CSM, CDDL and CDDR.

Algorithm 3 illustrates the `OpenMP` implementation of `CDDL` and `CDDR`. The parallelization is performed at the first and the second loop to go through all image pixels in such a way that each thread computes the disparity of one pixel.

Figure 4.7 shows the obtained results in KITTI dataset. The CPU hardware specifications of each employed processor have been presented previously in Table 4.1. In Intel Core i7-6700HQ, with **4** physical cores, we obtained an acceleration factor of around **3** in all cases. With Intel Core i7-4600M having **2** physical cores, a speedup of around



(a) Intel Core i7-6700HQ

(b) Intel Core i7-4600M

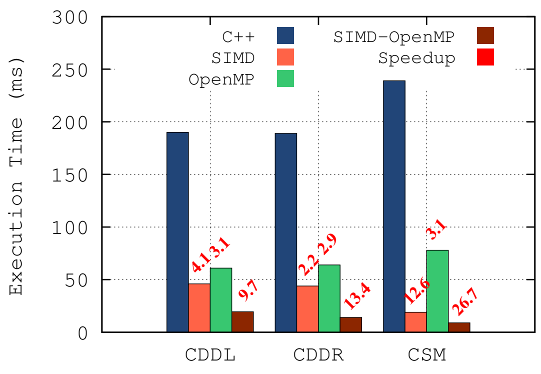
(c) Jetson TK1 ARMv7

(d) Tegra X1 ARMv8

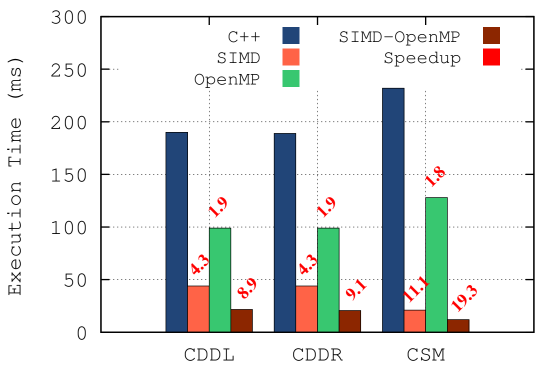
Figure 4.8: C++ vs OpenMP Execution Time on Middlebury Datasets.

2 is achieved – 1.9 with CDDL and CDDR, 1.8 with CSM. In NVIDIA Jetson TK1 with **4** ARM Cortex-A15 cores, we obtained an average speedup of **3.4**. In NVIDIA Tegra X1, we executed the code on both Cortex-A53 and Cortex-A57 processors. In both cases, we have **4** distinct cores. The results are depicted in Figure 4.7(d). First, we notice that there is no important difference between Cortex-A53 and Cortex-A57 results. Second, we obtained a speedup of 3 with `CDDR` and `CDDL`, and a speedup of **3.5** with `CSM` function.

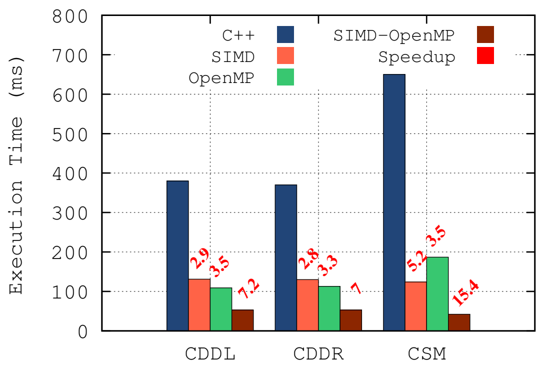
To evaluate the execution time of the `OpenMP` version with respect to image size, we executed the code on Middlebury dataset with different image resolutions. Figure 4.8 depicts the obtained results. Since the execution time of `CDDL` and `CDDR` function is



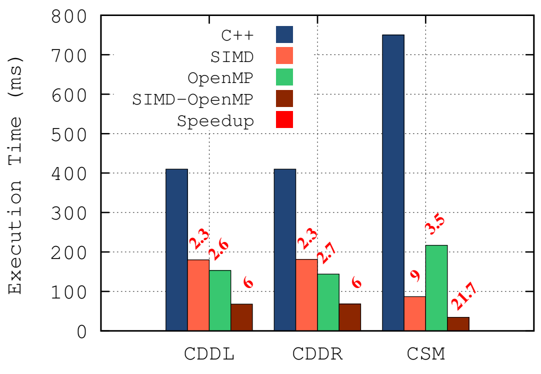
(a) Intel Core i7-6700HQ



(b) Intel Core i7-4600M



(c) Jetson TK1 ARMv7



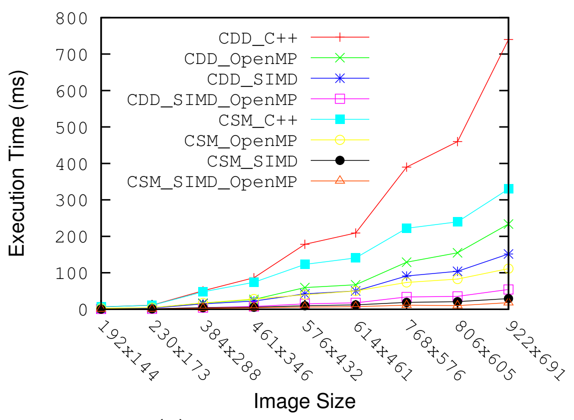
(d) Tegra X1 ARMv8, CortexA57

Figure 4.9: C++ vs SIMD vs OpenMP Performance on KITTI Dataset.

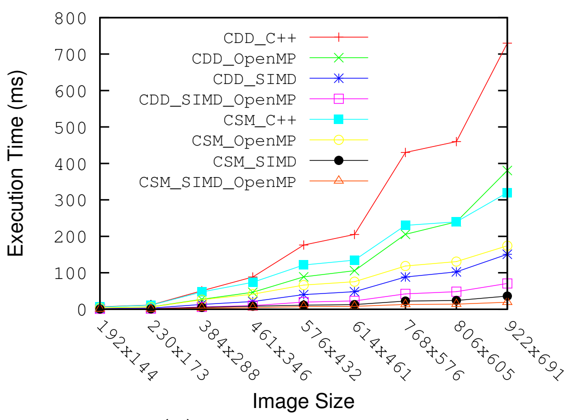
quite similar, we depict only the results of `CDDL` function referred as `CDD` in Figure 4.8 for better visibility of the curves. We notice that with `OpenMP`, the curves increase slowly compared to scalar version with `C++`. In NVIDIA Tegra X1, we observe that Cortex-A57 and Cortex-A53 both give the same performances with a very small difference at high resolution with both functions.

4.5 SIMD + OpenMP

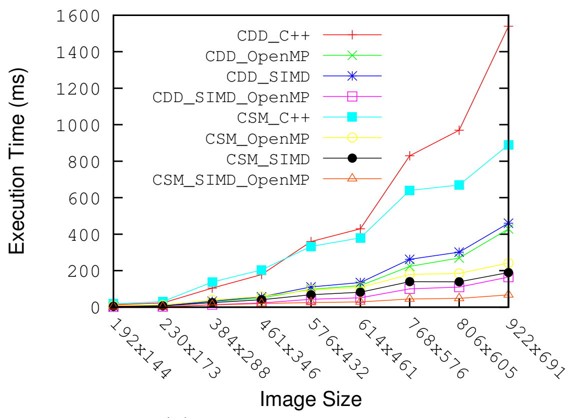
This section describes the obtained performance by merging both `OpenMP` directives and `SIMD` instructions to accelerate the stereo matching algorithm. The `SIMD` code is wrapped within `OpenMP` directives to take advantage of the maximum performance computing of the CPU.



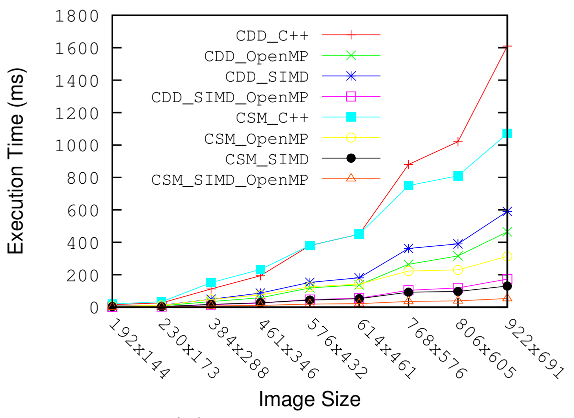
(a) Intel Core i7-6700HQ



(b) Intel Core i7-4600M



(c) Jetson TK1 ARMv7



(d) Tegra X1 ARMv8

Figure 4.10: C++ vs SIMD vs OpenMP Performance on Middlebury Dataset.

4.5.1 Obtained Results

Figure 4.9 depicts the first obtained results on KITTI dataset. In all architectures, the best results are obtained with both `SIMD` and `OpenMP`. In fixed platforms, we notice that Intel Core i7-6700HQ CPU with 4 physical cores outperforms Intel Core i7-4600M with only 2 physical cores. The same observation concerning Jetson TK1 which gives better performance with `CDDL` and `CDDR` compared to Tegra X1. The results depicted with Tegra X1 are obtained from Cortex-A57, actually we got the same performances with Cortex-A53.

Figure 4.10 shows the performance obtained from benching the algorithm on Middlebury dataset with different image resolutions. For better readability of the graphs, only `CDDL` function's results are depicted referred as `CDD`, indeed, `CDDL` and `CDDR` give the same performance in all cases. Also, in NVIDIA Tegra X1, since ARM Cortex-A57 outperforms lightly ARM Cortex-A53, the results of the later are not depicted. We notice the same observation as in KITTI dataset(Figure 4.9), The best performance in all platforms with both function (`CDD`, `CSM`) and with all image resolutions are obtained by using `OpenMP` directives and `SIMD` instructions.

4.6 OpenACC Parallelization on CPU

4.6.1 Obtained Results

We used `OpenACC` directives to parallelize the identified bottlenecks – `CSM, CDDL` and `CDDL`. The approach is similar as with `OpenMP` since they are both based on directives. The idea is to keep the scalar source code and add directives before the loops to parallelize. It is worth noting that according to our best knowledge, there is non `OpenACC` compiler for ARM architecture. Hence, the code has been executed only on platforms with Intel processors (Intel Core i7-6700HQ, Intel Core i7-4600M). Release 16.10 of PGI compiler has been employed. Hardware specification of both architectures are presented in Table 4.1. It is important to mention that scalar (`C++`) results in all experiments are obtained with `g++-4.8` compiler and `gprof` profiler. It is also important to mention that `OpenACC` is used only on CPU-based systems in this chapter.

4.6.1.1 CDD Function

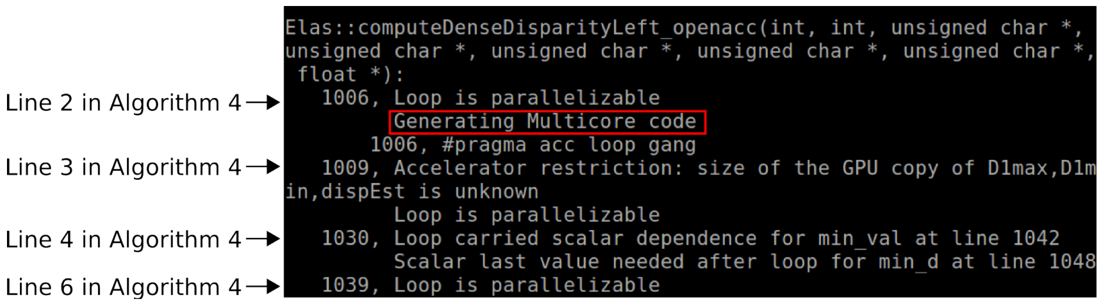
We implemented `CDDL` and `CDDR` functions with `OpenACC` directives. The original `C++` source code has been used. Directives have been added to guide the compiler to parallelize the loops. We used `pgc++` compiler of PGI. To compile `OpenACC` on multi-core systems, we need to add the `-ta=multicore` option to select multi-core system as the target accelerator for which to compile. We used the following compiling options : `-O3 -acc -ta=multicore -Minfo=accel`. The `-Minfo=accel` option reports relevant information on the optimizations applied. As first experiment, we selected to test both approaches; with `kernels` and `parallel` constructs.

Algorithm 4 illustrates the pseudo code employed to accelerate `CDDL` and `CDDR` functions with `OpenACC` on multi-core systems. As illustrated, we use only `kernels` clause at the beginning of the first for loop to give hint to the compiler concerning the region we wish to parallelize. The rest of the algorithm as well as the scalar code `C++` were not touched.

Algorithm 4 Parallel Stereo Matching of A-ELAS (CDDL/CDDR) with `OpenACC`

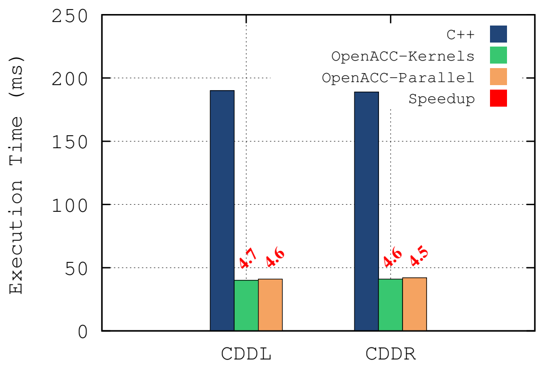
```
Require: LD[H x W x 16], RD[H x W x 16], PRIOR[256], DMIN[H], DMAX[H],
    PD[H x W]
 1: #pragma acc kernels                      ▷ parallel region: line 2 → line 20
 2: for row=2 to H-2 parallel do
 3:     for col=2 to W-2 parallel do
 4:         for d=DMIN[row] to DMAX[row] do          ▷ disparity reasearch distance
 5:             SAD = PRIOR[|d-PD[row][col]|]
 6:             for i=0 to 15 do           ▷ go through pixel's descriptor elements
 7:                 if left then                        ▷ left disparity (forward)
 8:                     SAD += abs(LD[row][col][i] - RD[row][col-d][i])
 9:                 else                               ▷ right disparity (backward)
10:                     SAD += abs(LR[row][col][i] - LD[row][col+d][i])
11:                 end if
12:             end for
13:             if SAD<SAD_min then
14:                 SAD_min = SAD
15:                 disparity = d
16:             end if
17:         end for
18:         D[row][col] = disparity
19:     end for
20: end for
```

Figure 4.11 depicts the compiling information feedback of PGI compiler of `CDDL` function with `OpenACC`. From this figure we first notice that PGI compiler detects the four loops presented in the algorithm. Second, only the outermost for loop has been parallelized at `gang` level which confirms the aforementioned limitations of `OpenACC`. At line 1009 in Figure 4.11, the compiler gives back a restriction concerning input data employed which have not been allocated on GPU. This a bug on the compiler since we are working on multi-core systems. On line 1030 corresponding to the for loop at line 4 in Algorithm 4, the compiler detects a data dependency which does not allow to loop to be parallelizable
.

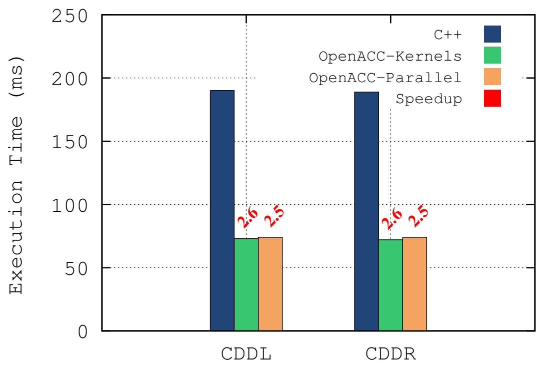


```
Elas::computeDenseDisparityLeft_openacc(int, int, unsigned char *,
unsigned char *, unsigned char *, unsigned char *, unsigned char *,
 float *):
   1006, Loop is parallelizable
         Generating Multicore code
       1006, #pragma acc loop gang
   1009, Accelerator restriction: size of the GPU copy of D1max,D1m
in,dispEst is unknown
         Loop is parallelizable
   1030, Loop carried scalar dependence for min_val at line 1042
         Scalar last value needed after loop for min_d at line 1048
   1039, Loop is parallelizable
```

Figure 4.11: PGI Compiling Information Feedback of CDDL Function with OpenACC



(a) Intel Core i7-6700HQ

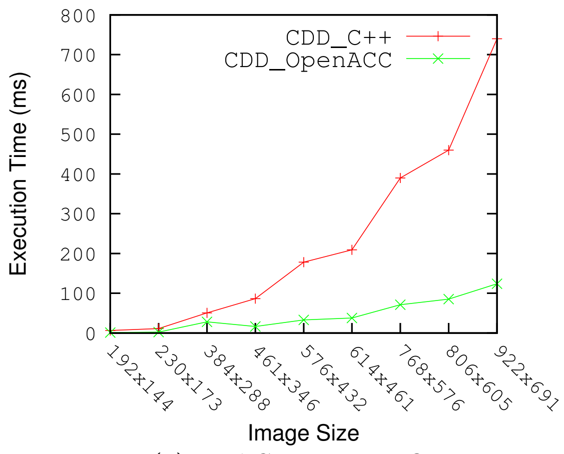


(b) Intel Core i7-4600M

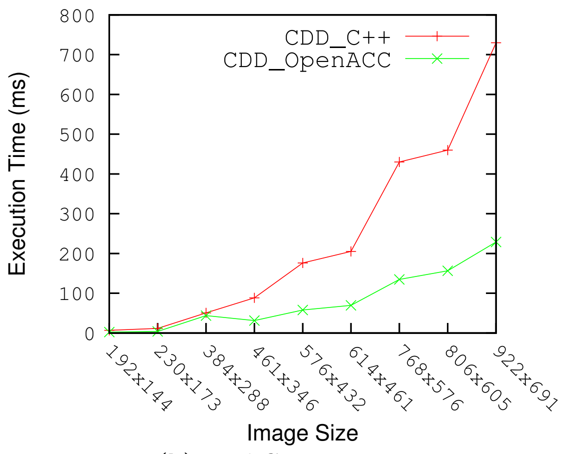
Figure 4.12: C++ vs OpenACC Execution Time on KITTI Dataset of CDD Function.

In the second experiment, we replaced the `kernel` pragma in Algorithm 4 with a parallel construct : `#pragma acc parallel loop`. Since the parallelization is only possible at one level, we did not add this pragma to the other loops. As complier feedback, we got the same information as illustrated in Figure 4.11.

Figure 4.12 depicts the obtained results on KITTI dataset with both approaches; `kernels` and `parallel`. First, we notice that on both architectures, we got almost the same results in terms of execution time with both `kernels` and `parallel` clauses. Red labels associated to the top of each bar corresponds to the speedup obtained with respect to the scalar version (`C++`). In Intel Core i7-6700HQ with 4 physical cores and 8 threads (2 threads per each core), `OpenACC` version of `CDDL/CDDR` achieves a speedup of `4.75`. In the second platform –Intel Core i7-6700HQ, with 2 physical cores and 4 threads (2 threads per each core), we achieve a speedup of `2.6`. We notice that in both cased the speedup is greater than the number of physical cores. Indeed, `OpenACC` applies



(a) Intel Core i7-6700HQ



(b) Intel Core i7-4600M

Figure 4.13: C++ vs OpenACC Execution Time on Middlebury Dataset of CDD Function.

also auto vectorization included on `-O3` option whenever its possible. Figure 4.13 shows the execution time of `CDDL` referred as `CDD`, in Middlebury dataset with different image resolutions. Only results obtained with `kernels` are depicted for better visibility since there is a very small difference with respect to `parallel` clause. We clearly notice the the important speedup obtained with `OpenACC` on both architectures. We also notice that the `OpenACC` curves rise slower than the scalar ones.

4.6.1.2 CSM Function

We followed the same approach to parallelize `CSM` function with `OpenACC` directives. The same compilers – `g++-4.8` for `C++` and `pgc++` for `OPenACC`. The same compiling options are also used. We used `kernels` clause in this case. Figure 4.14 illustrates the approach employed to compute the cost function in `CSM` function. There two important differences between `CDDL/CDDR` functions and `CSM` in terms of computing the disparity:

1. *Disparity research distance* In `CDDL/CDDR` functions, we limit the boundaries (min, max) when looking for the homologue pixel thanks to the prior data provided by the support points. However, for `CSM` function, since at this level we do not have any prior information, the disparity research distance is set to $[0, 255]$ corresponding to the maximum distance.

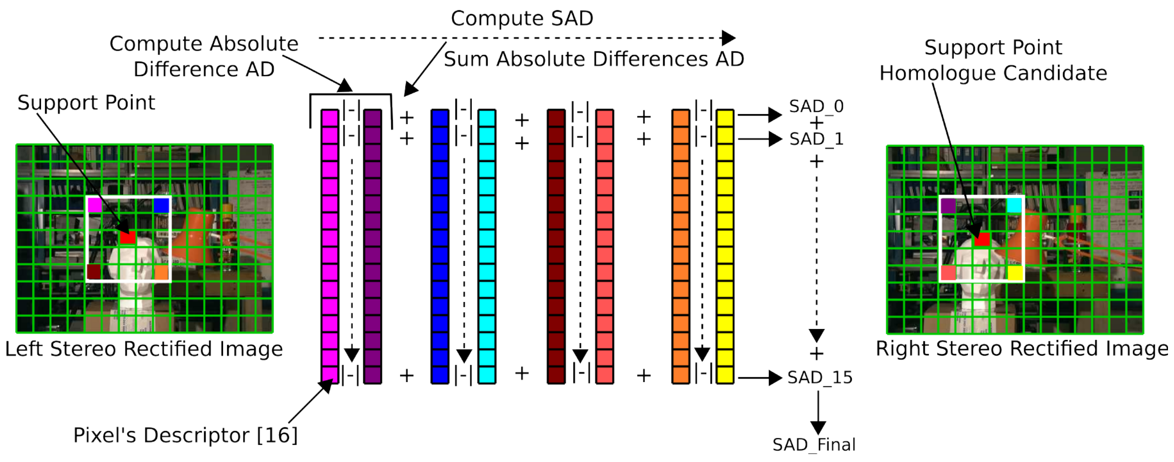

Compute SAD
Compute Absolute Difference AD
Sum Absolute Differences AD
Support Point
Support Point Homologue Candidate
Left Stereo Rectified Image
Right Stereo Rectified Image
Pixel's Descriptor [16]
SAD_0
SAD_1
SAD_15
SAD_Final

Figure 4.14: CSM Cost Function Approach

2. *Cost aggregation* As depicted in Figure 4.14, we use the four corners of a 5×5 window centered at the considered support point to compute the cost function. We use pixel's descriptor of 16 elements associated to each pixel/corner.

3. *All in one* Figure 4.15 depicts a flowchart of the algorithm employed in `CSM` function to compute the disparity of one support point. For the final dense disparity, forward disparity, backward disparity and consistency check are three distinct functions. Functions `CDDL` and `CDDR` are completely independent. However, as illustrated in Figure 4.15, backward disparity depends on forward disparity. Also consistency check between left and right disparity is performed at the end before validating the disparity.

With `OpenACC`, we used the same approach as with `CDDL/CDDR` functions. We put `OpenACC` pragmas on the outermost loop with **`kernels`** construct. We expected to get as previously with `CDD` function a speedup of at least **4** and **2** in Intel Core i7-6700HQ and Intel Core i7-4600M respectively. Figure 4.16 (a) depicts the results of the first implementation. It is clear that the obtained results are far from the expectations. While in in Intel Core i7-6700HQ, a speedup of less than **2** is achieved, in Intel Core i7-4600M, we notice almost no speedup.

To understand the obtained results, we wanted first to check the execution time of the scalar `C++` version of `CSM` function compiled with `pgc++` compiler. The results are depicted in Figure 4.16 (b). We notice clearly that `pgc++` version is slower than `g++`

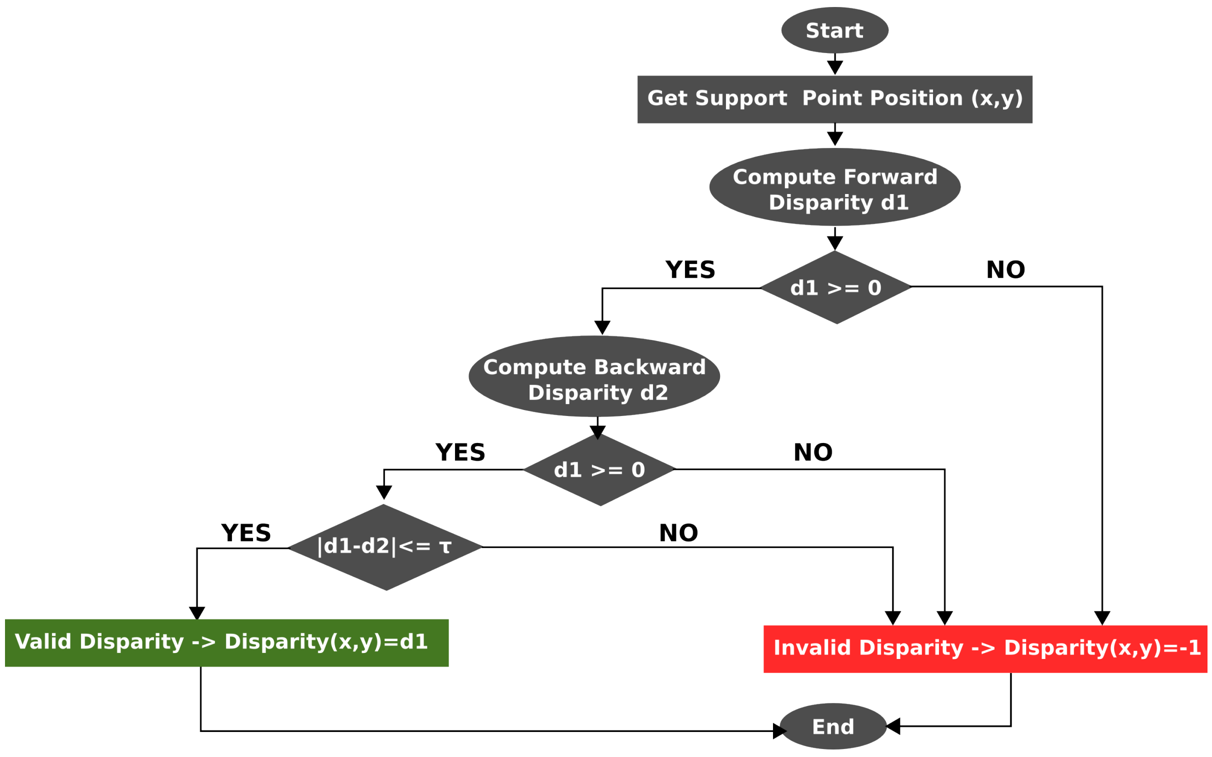

Start
Get Support Point Position (x,y)
Compute Forward Disparity d1
d1 >= 0
YES
NO
Compute Backward Disparity d2
d1 >= 0
YES
NO
|d1-d2|<= τ
YES
NO
Valid Disparity -> Disparity(x,y)=d1
Invalid Disparity -> Disparity(x,y)=-1
End

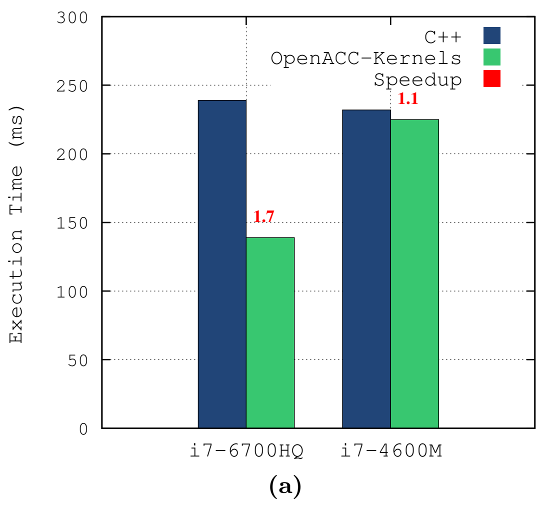
Figure 4.15: CSM Algorithm Flowchart

one. These results have been also noticed by some users of `pgc++` compiler and posted question on PGI forum without getting clear explanation.

The first track we investigated concerns the compiling options. Since PGI compiler optimize the code accordingly. We focused on `-O3` option. The later specifies aggressive global optimization. It performs all level-one (`-O`) and level-two (`-O2`) optimizations and enables more aggressive hosting and scalar replacement optimizations that may or may not be profitable. Globally, it proposes three options:

- `-Mcache_align` aligns large objects on cache-line boundaries
- `-Mpre` enables partial redundancy elimination
- `-Mvect=sse` controls automatic `SSE` vectorization.

To understand why `pgc++` is slower than `gc++`, the three options are tested. We enable and disable each option and we verify its effect on the execution time. With the first option (`-Mcache_align`) we did not notice any effect neither with the second one (`-Mpre`). To test the third option, it is a bit more complicated since it consists of a set



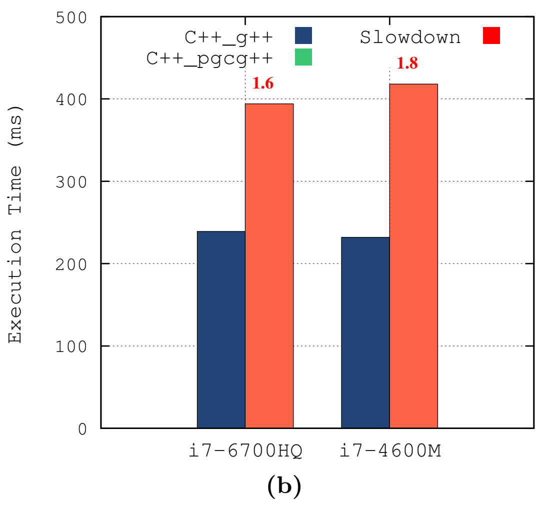


Figure 4.16: C++ vs OpenACC Execution Time on KITTI Dataset of CSM Function.

Compiling Option	**Time(ms) C++(pgc++)**				**Time(ms) C++(g++)**
	-O3	**-O3 -Mvect=nosse**	**-O3 -Mvect=levels:x**	**-O3 -Mvect=nosse,levels:1**	**-O3**
Intel Core i7-6700HQ	394	292	x=1 : 255 x=2 : 396	259	239
Intel Core i7-4600M	418	293	x=1 : 258 x=2 : 415	263	232

Table 4.4: C++ Execution Time of CSM with pgc++ Compiling Options vs g++ on KITTI Dataset

of sub-options (17 exactly) such as `fuse` to enable loop fusion, `simd` to generate `SIMD` instructions and `tile` to enable loop tiling. We tested all these sub-options one by one. We found that only two sub-options have effects on execution time which are : `[no]sse` and `levels:<n>`. The formal one generates or not the `SSE` instructions, the later one sets the maximum nest level of loops to optimize.

Table 4.4 shows the execution time of the scalar version (`C++`) of `CSM` function with different configurations of aforementioned options (`sse,levels:<x>`). We notice clearly the difference with only `-O3` option between `g++` and `pgc++` compilers. When `nosse` option is used, the execution time decreases but is still greater than the `C++` version with `g++`. The second option (`levels:<x>`) gives rather better results, we reach the smallest execution time with `pgc++` by setting the maximum nest loops to optimize

Compiling Option	**Time(ms) OpenACC(pgc++)**				**Time(ms) C++(pgc++)**
	-O3	**-O3 -Mvect=nosse**	**-O3 -Mvect=levels:x**	**-O3 -Mvect=nosse,levels:1**	**-O3 -Mvect=levels:x**
Intel Core i7-6700HQ	139	105	x=1 : 86 x=2 : 135	92	255
Intel Core i7-4600M	230	160	x=1 : 136 x=2 : 244	141	258

Table 4.5: OpenACC Execution Time of CSM with pgc++ Compiling Options on KITTI Dataset

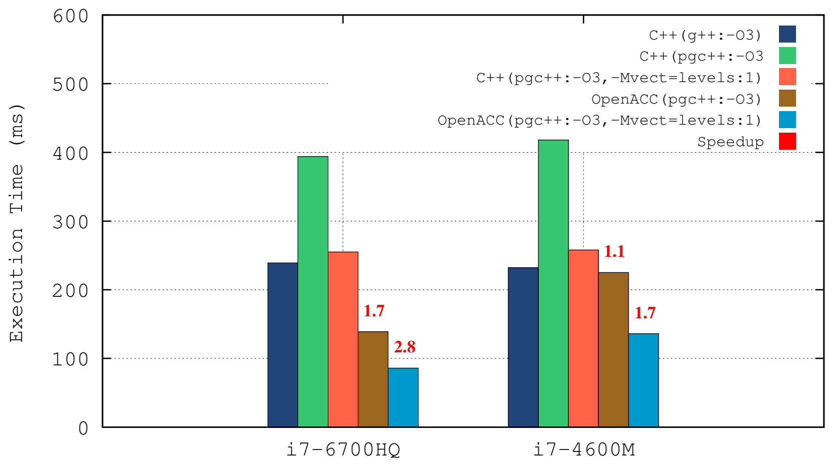
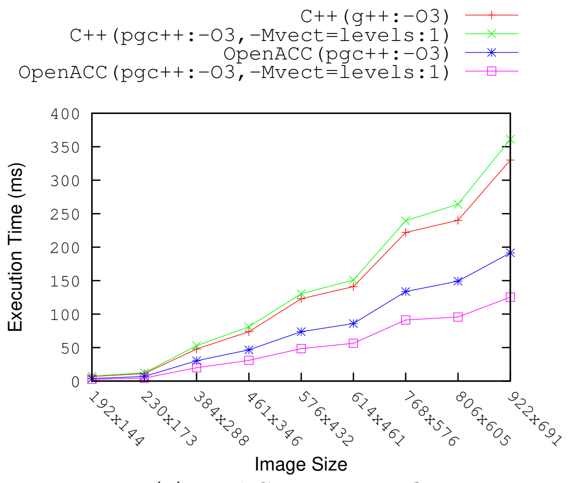


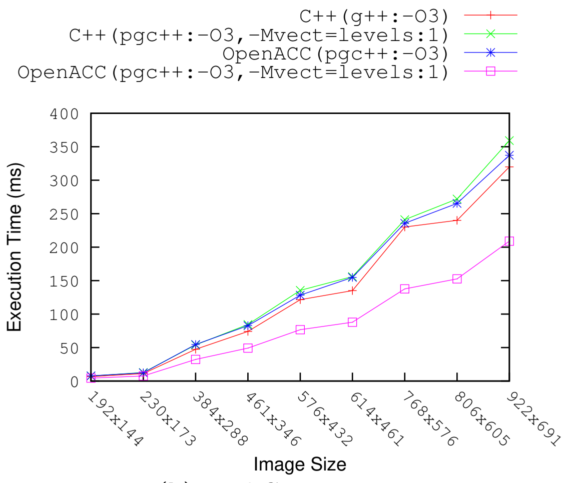
Figure 4.17: C++ vs OpenACC Execution Time on KITTI Dataset of CSM Function.

to only **one** level. When both options are merged, the execution time is smaller than with only `-O3` option, however, it is lightly greater than the execution time with only `levels:1` option.

We did the same experiment with `OpenACC`, we measured the execution time with different configurations of `-O3, nosse` and `levels:<x>` options. Table 4.5 depicts the obtained results. With only `-O3` option, we notice a very small speedup on Intel Core i7-4600M architecture with **2** physical cores and **4** threads. In Intel Core i7-6700HQ with **4** physical cores and **8** threads, we achieved a speedup of `1.8`. As previously with scalar version, the option `levels:1` together with `-O3` gives the best speedup; **1.9** in Intel Core i7-4600M and **2.9** in Intel Core i7-6700HGQ. Figure 4.17 summarizes the obtained results on KITTI dataset. We clearly notice the improvement on the speedup when compiling options are configured for a maximum performance.



(a) Intel Core i7-6700HQ



(b) Intel Core i7-4600M

Figure 4.18: C++ vs OpenACC Execution Time on Middlebury Dataset of CSM Function.

To evaluate the implemented `OpenACC` version of `CMS` function, the later has been benched on different image resolutions from Middlebury dataset. Figure 4.18 depicts the obtained results. We selected the worst results with default compiling options (`-O3`) and the best results we obtained with `-O3, -Mvect=levels:1`. In scalar version (`C++`), we notice a small difference before and after setting the right compiling options. However, this difference is more important with `OpenACC` on both architectures and gets greater with image resolutions. Also this difference with `OpenACC` is higher on Intel Core i7-4600M processor with only two physical core compared to the Intel Core i7-6700HQ with 4 physical cores.

4.7 NT2 Implementation

In this section, we present the obtained results of accelerating `CDD` and `CSM` functions with `NT2`. `NT2` implements a subset of `MATLAB` language as a DSEL based on *Expression Template* C++ idiom. `NT2` has been presented previously in chapter 2, for more details, refer to section 2.3.4.2. At `NT2` compilation, automatic rewriting may occur when architecture-driven optimizations are possible and user high-level algorithmic improvements are introduced. These compile-time optimizations are integrated within `Boost.SIMD` [57].

4.7.1 Obtained Results

NT2 proposes different optimizations (2.3.4.2) such as the colon option "_". The latter applies vectorization on sub-matrix access which corresponds to ":" in MATLAB. We have used this option to accelerate both CDD and CSM functions.

4.7.1.1 CDD Function

To accelerate the CDD function with NT2, we selected the colon (_) for vectorization. The idea consists in optimizing at line granularity, in other words, vectorization is applied on one image's line through the colon option. Algorithm 5 illustrates the global pseudo code employed to compute the disparity of all image's line in parallel with NT2 "_" option.

Algorithm 5 Parallel Stereo Matching of A-ELAS (CDDL) with NT2

```
Require: LD[H×W×16], RD[H×W×16], PRIOR[256], DMIN[H], DMAX[H],
    PD[H×W], DISP_OUT[H×W]
1: for row=2 to H-2 parallel do
2:     disp_max = (int)*(DMIN + row)
3:     disp_min = (int)*(DMAX + row)
4:     computeDisparityLeftLine(LD, RD, row, disp_min, disp_max, PD,
    PRIOR, DISP_OUT
5: end for
```

As illustrated in Algorithm 5, we have one `for loop` which iterates over image's lines (line 1). Then we get the boundaries of the disparity range (lines 2, 3). After that we call the function `computeDisparityLeftLine()` (line 4) for left disparity in CDDL function to compute the disparity of all pixels in each line (`row`). It is worth noting that the same pseudo code is employed with CDDR function by using another function referred as `computeDisparityRightLine()`. The only difference as explained previously is that with CDDR function, since we compute the backward disparity, we take the right descriptors (RD) as the reference from which we subtract the values of the left descriptors (LD). Listing 4.5 illustrates how to use NT2 to program the `computeDisparityLeftLine()` function.

```
 1 computeDisparityLeftLine(T &ld, T &rd, int32_t v, int32_t disp_min, int32_t disp_max, uint8_t* pd
       , int32_t* prior, float* disp_out )
 2 {
 3   using nt2::_;
 4   nt2::table<float, nt2::settings(nt2::shared_,nt2::_1D)> disp_out_shared( nt2::_1D(w), nt2::
        share((float*)disp_out+v*w, (float*)disp_out+v*w + w));
 5   nt2::table<int16_t> min_valueT(nt2::of_size(w));
 6   nt2::table<int16_t> minval(nt2::of_size(w));
 7   nt2::table<float> mind(nt2::of_size(w));
 8   nt2::table<int16_t> prior_line(nt2::of_size(w));
 9   minval(_) = (int16_t)(10000);
10   mind(_)  = (float)(-1);
11   min_valueT(_) = (int16_t)(0);
12   for(int32_t disp_range = disp_min; disp_range <disp_max; ++disp_range) {
13     int32_t uwarp_begin = 2-disp_range ;
14     int32_t uwarp_end = w-2-disp_range;
15     for (int ii = 1; ii<=w; ii++){
16       uint8_t d_plane = pd[ii-1 + v*w];
17       prior_line(ii) = *(prior+abs(disp_range- d_plane));
18     }
19     min_valueT(_(2, w-2)) = prior_line(_(2, w-2));
20     for (int32_t jj = 1; jj<=16; jj++)
21       min_valueT(_(2, w-2)) += nt2::abs(desc1(_(2, w-2),v_offset, jj) - desc2(_(uwarp_begin,
            uwarp_end),v_offset, jj)) ;
22     minval(_(2, w-2)) = nt2::if_else( min_valueT(_(2, w-2)) < minval(_(2, w-2)), min_valueT(_(2,
          w-2)), minval(_(2, w-2)));
23     mind(_(2, w-2)) = nt2::if_else( min_valueT(_(2, w-2)) == minval(_(2, w-2)), nt2::cast<float>(
          disp_range), mind(_(2, w-2)));
24   }
25   disp_out_shared(_(2, w-2)) = nt2::if_else( mind(_(2, w-2)) >= 0.f, mind(_(2, w-2)), (float)(-1)
        );
26 }
```

Listing 4.5: Code Snippet of computeDisparityLeftLine() Function with NT2

While programming with NT2, we work on NT2 table class. Hence, adaptations are required to fit the specifications of this class. Here are the most important features to take into consideration:

- To avoid creating an NT2 table of the output image and copying back the results to the C++ array (disp_out, line 1, Listing 4.5), we use the NT2 shared option of table class. The idea consists on creating an NT2 table which is actually a view of the C++ array without any data copy. This allows us to have access as input to the selected C++ array. Also, when the NT2 shared table is updated, the C++ equivalent data are also updated automatically without any data copy. This is

Table 4.6: Execution Time (ms) of CDDL Function (Listing 4.5) with NT2

Platform	i7-6700HQ	i7-4600M	Jetson K1 ARMv7	Tegra X1 ARMv8
C++	190	190	380	410
NT2	231	230	642	854
NT2_SIMD	228	226	425	898
NT2_OpenMP[1]	132	147	965	500
NT2_SIMD_OpenMP	131	150	548	555

[1] add `-fopenmp` compiling option

illustrated in Listing 4.5, line 4. we created a view to the `C++` array data (pixels) at line `v`.

- All operations have to be performed between tables. In other words, if in `C++` version we used some constants variables to be added or subtracted, then with `NT2`, we need to create an `NT2 table` holding the same constant. This is also applied to temporary variables as illustrated in Listing 4.5, lines 5 $\rightarrow$ 11.
- Control operators such as `"if then else"` in `C++` code, have to be substituted with their `NT2` equivalent which can be applied to a set of data simultaneously (Listing 4.5, lines 22, 23, 25).

To enable code acceleration with `NT2`, we rely on compiling options. In our case, since we employed the `colon (_)` option which is supposed to apply vectorization along the data, we added the following options :

- Intel CPU: `-DBOOST_SIMD_NO_STRICT_ALIASING -fno-strict-aliasing -ffast-math -msse2 -msse3 -msse4.1 -msse4.2`
- ARM Jetson K1: `-DBOOST_SIMD_NO_STRICT_ALIASING -fno-strict-aliasing -mfpu=neon -ffast-math`
- ARM Tegra X1: `-DBOOST_SIMD_NO_STRICT_ALIASING -fno-strict-aliasing -mcpu=cortex-a57+simd+fp -ffast-math`

We then executed the `NT2` code of `CDD` function on our selected CPU-based platforms. Table 4.6 gives the execution time in ms with different compiling options. In addition to enabling `SIMD` vectorization, we also tested `OpenMP` parallelization by adding the `-fopenmp` compiling option on both Intel and ARM CPUs. The first line (`C++`) gives

the execution time (ms) of the scalar version taken as the reference without any NT2 code or any specific compiling option. In the second line (NT2), we have the execution time of the NT2 version corresponding to the code illustrated in Listing 4.5. If we compare C++ to NT2, we notice that code rewriting with NT2 adds a cost at the execution time which is significantly important at ARM-based architectures.
The third line (NT2_SIMD) shows the obtained execution time with NT2 by enabling the SIMD instructions. The code of this version is the same as the one employed in the second line (NT2). To enable SIMD, we just need to add the required compiling options as explained previously, no code rewriting is necessary. We notice different results among the four platforms. First, in Intel-based CPUs (i7-6700HQ, i7-4600M), we have got almost no performance compared to NT2 results. In Jetson K1 ARM processor, we obtained a speedup of **1.5** with NT2_SIMD compared to NT2. In Tegra X1 ARM processor, the execution time has rather increased. The absence of performance with SIMD can be explained by the fact that NT2 did not mange to unroll the instructions with SIMD. In this case, it lets the compiler the freedom to try to unroll or apply any other optimization to accelerate the code. This may happen when the arithmetic intensity is not high enough for NT2.
To accelerate the code, we added OpenMP compiling option to the same code. The obtained results are shown in line 4 in Table 4.6 (NT2_OpenMP). We observe a small speedup in Intel CPUs, **1.4** in i7-6700HQ and **1.3** in i7-4600M processor. However, in Jetson TK1 ARM processor, the execution time has increased compared to both NT2 and NT2_SIMD results. In Tegra X1, we managed to obtain a speedup of **1.7** compared to NT2 execution time. The low performance with OpenMP can be explained by the memory bandwidth at L3 cache which is shared among the processor and may be small compared to L2 cache or DRAM.

4.7.1.2 CSM Function

With CDD function, we parallelized at line granularity in such a way to compute the disparity of all image's line pixels simultaneously. In CSM function, we do not compute the disparity for all pixels, but only for the supports points candidates. Supports points are separated from each other horizontally and vertically by 5 pixels. Hence, we cannot keep the same programming model as in CDD function, otherwise we loose in terms of performance. The approach we followed consists in vectorizing the loop scanning pixels in the right image (here we take the left image as a reference) to compute the cost matching of each support point. In CSM function, we do not have any prior data at this level to estimate the disparity research range. the cost aggregation window is slided along 255 distance, which is the maximum disparity we can get.

```
 1 // for each support point condidate do
 2   for(int16_t jj = disp_min; jj <=disp_max; jj += BLOCK) {
 3     if (jj + BLOCK < disp_max) {
 4       if (!right_image) u_warp = u-jj; //left/forward disparity
 5       else             u_warp = u+jj; //right/backward disparity
 6       sums(_) = (int16_t)(0);
 7       if (!right_image) { // compute left/forward disparity
 8         for(int i=1;i<=16;i++){
 9           sums(_) += nt2::abs( LD(_(u-u_step, u-u_step+BLOCK-1), v-v_step, i) - RD(_(u_warp-u_step
                 , u_warp-u_step+BLOCK-1), v-v_step, i));
10           sums(_) += nt2::abs( LD(_(u+u_step, u+u_step+BLOCK-1), v-v_step, i) - RD(_(u_warp+u_step
                 , u_warp+u_step+BLOCK-1), v-v_step, i));
11           sums(_) += nt2::abs( LD(_(u-u_step, u-u_step+BLOCK-1), v+v_step, i) - RD(_(u_warp-u_step
                 , u_warp-u_step+BLOCK-1), v+v_step, i));
12           sums(_) += nt2::abs( LD(_(u+u_step, u+u_step+BLOCK-1), v+v_step, i) - RD(_(u_warp+u_step
                 , u_warp+u_step+BLOCK-1), v+v_step, i));
13         }
14       }
15       else { /* compute right/backward disparity */ }
16       // disparity refinement and optimization (update maximum a posteriori (MAP))
17     }
18     // if number of pixels < BLOCK size
19     else { /* compute left and right disparity in C++ */ }
20   }
21   // check for best match, if valid, return disparity value, or return -1
```

Listing 4.6: Code Snippet of CSM() Function with NT2

To parallelize, we propose to divide the sliding region, from minimum disparity to maximum, into equal blocks. We vary the size of block to find the optimal one and analyze the evolution of the obtained performance while varying the block size.

Listing 4.6 illustrates part of the code. The disparity research region (`[disp_min, disp_max]`) is divided into equal regions of `BLOCK` size each (line 2). If the last region size is less than `BLOCK`, then the remaining pixels will be processed sequentially (`C++`) as illustrated in line 19. In line 8, we have a loop which iterates over the descriptors (16 elements). The four sums (lines 9, 10, 11, 12) corresponds to the four corners in a window of 5×5 surrounding the corresponding support points. The colon (`_`) option is employed compute the SAD between the left (`LD`) and right (`RD`) descriptors. The same approach is employed to compute the backward disparity of the right image by

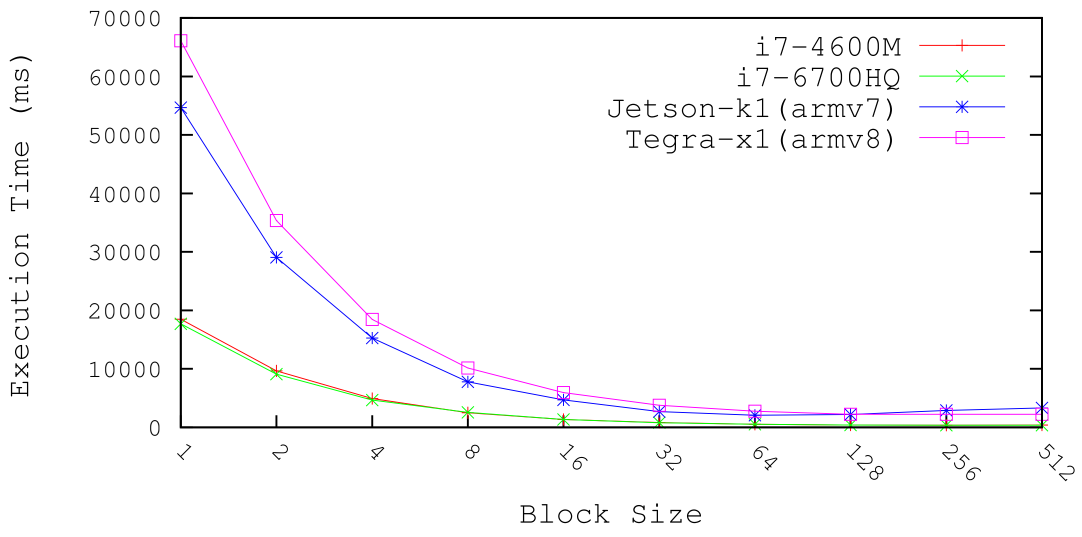


Figure 4.19: Execution Time of `CSM` Function with`NT2` on KITTI Dataset

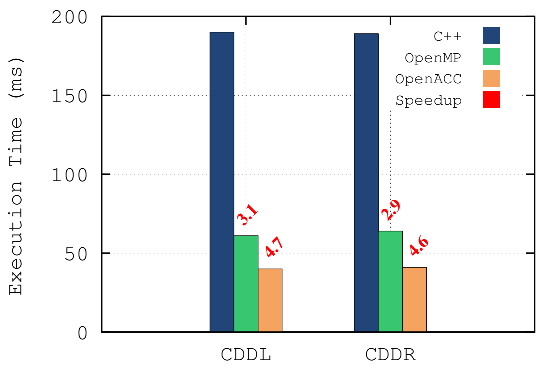
interchanging the left and right descriptors (`LD, RD`) while computing the sums (lines 9 → 12) and using the appropriate disparity (line 5).

Figure 4.19 depicts the obtained execution time (ms) of `CSM` function written with `NT2` executed on the four platforms. In this case, we have just enabled the `SIMD` flags previously presented; we did not use `OpenMP`. First, we notice high execution time in all architectures at `BLOCK=1`. Then, the execution time decreases significantly while increasing the `BLOCK` size corresponding actually to the `SIMD` registers size. However, the best performance is far away from the expected ones.

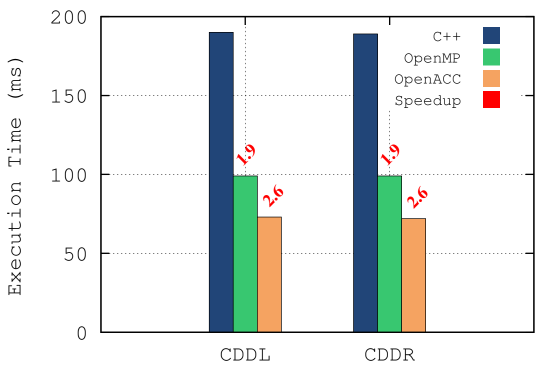
4.8 Evaluation

To evaluate the employed tools in this chapter, we propose to compare :

1. `OpenMP` to `OpenACC`, since they are both directive-based approaches
2. `SIMD` to `NT2` since they both apply vectorization. While with `SIMD`, we used explicit intrinsics to write the code, in `NT2`, we wrote the code with the `colon "_"` option to hint the compiler to generate a vectorized code.



(a) Intel Core i7-6700HQ



(b) Intel Core i7-4600M

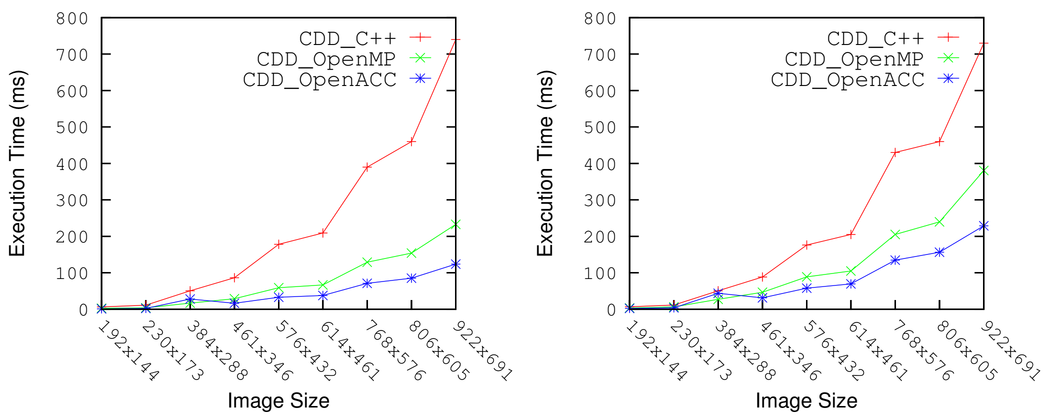
Figure 4.20: C++ vs OpenMP vs OpenACC Execution Time on KITTI Dataset.

4.8.1 OpenMP vs OpenACC

In this section, we discuss the obtained results with shared parallelization approach. We compare the two tools employed for this purpose; `OpenMP` and `OpenACC`. Since we could not use `OpenACC` on ARM-based systems – lack of compatible compiler, we compare the obtained results inly on x86-based systems (Intel Core i7-6700HQ, Intel Core i7-46008).

4.8.1.1 CDD Function

If we compare from programing productivity between `OpenMP` and `OpenACC`, we can say that it is quite similar. As discussed before when results are presented, we did not meet any special difficulty in CDD function with both tools. All we did is we took the scalar version (`C++`) and we added `OpenMP/OpenACC` directives before the for loops we want to parallelize. Figure 4.20 shows the difference in terms of execution time between `OpenMP` and `OpenACC` on KITTI dataset as well as the obtained speedup w.r.t `C++`. The scalar (`C++`) results are obtained by using `g++-4.8` compiler. We clearly observe that `OpenACC` gives better performances in both architectures compared to `OpenMP`. This difference may be explained by the optimizations applied by PGI compiler such as *auto-vectorization* and *loop unrolling* in addition to the shared memory parallelization applied through the directives explicitly by the programmer. Figure 4.21 depicts the obtained execution time on Middlebury dataset with different image resolutions.



(a) Intel Core i7-6700HQ **(b)** Intel Core i7-4600M

Figure 4.21: C++ vs OpenMP vs OpenACC Execution Time of CDD Function on Middlebury Dataset.

4.8.1.2 CSM Function

As discussed previously, we encountered some difficulties to accelerate `CSM` function with `OpenACC` compared to `OpenMP`. With `OpenMP`, the approach was straightforward as we did with `CDD` function; adding directives before the `for loops` we wish to parallelize was enough. However, with `OpenACC`, we noticed that compiling options affected significantly the obtained performance. Specifying explicitly some options – number of nested loops to parallelize, was required to accelerate the algorithm. Hence, from programming productivity, it took more time to accelerate the function, to understand and analyze the reasons behind it.

Figure 4.22 depicts the execution time of `CSM` function with `OpenMP` versus `OpenACC`. Different compiling options are employed with `pgc++` compiler to obtain the best performance as close as possible to `OpenMP` ones. We observe that `OpenMP` outperforms lightly `OpenACC`. Figure 4.23 depicts the execution time of `CSM` on Middlebury dataset with both `OpenMP` and `OpenMP`. We notice the same observation as in KITTI dataset, `OpenMP` is lightly faster compared to `OpenACC`.

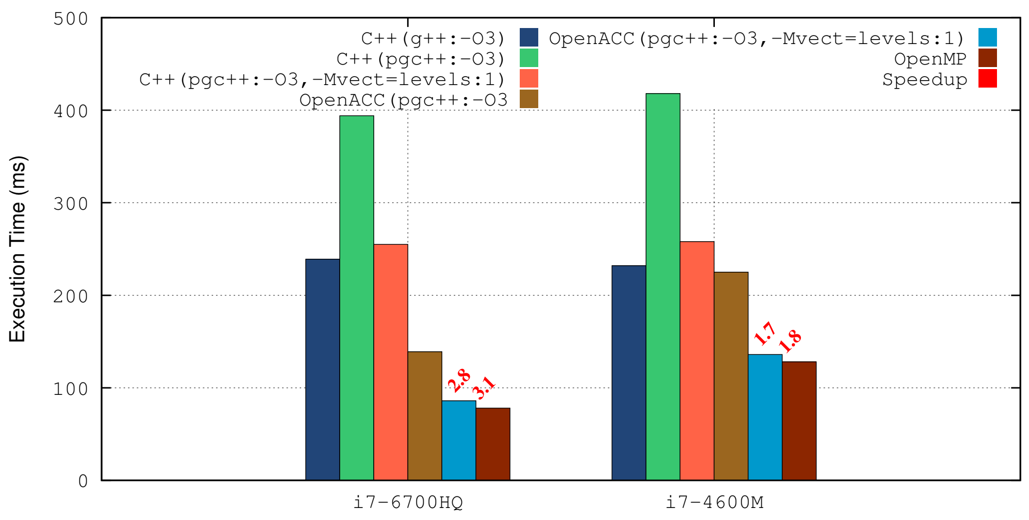
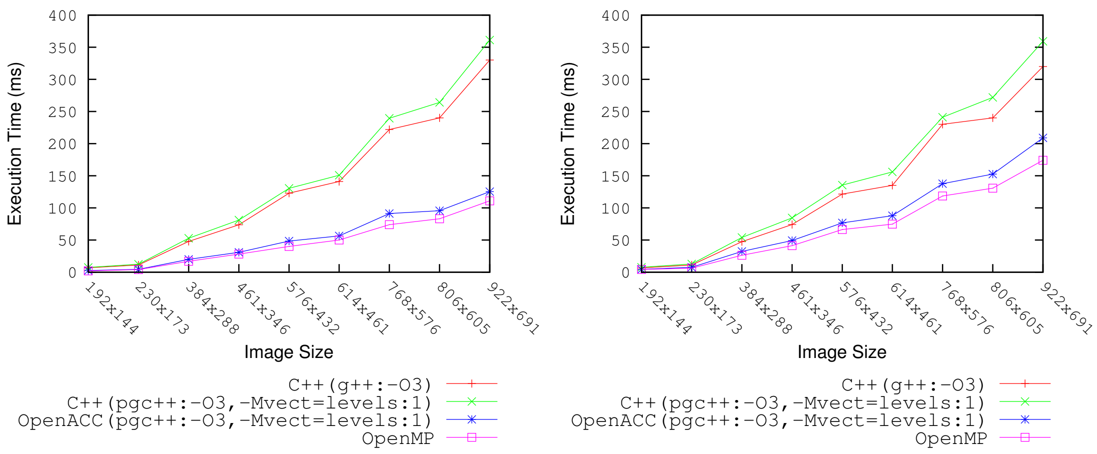


Figure 4.22: Execution Time of `CSM` Function



(a) Intel Core i7-6700HQ **(b)** Intel Core i7-4600M

Figure 4.23: C++ vs OpenMP vs OpenACC Execution Time of CSM Function on Middlebury Dataset.

4.8.1.3 Limitations of OpenACC on Multi-core Systems

Does OpenACC support multi-level parallelism in multi-core systems? Release 16.10 of `OpenACC` is employed in this work which have few limitations. First, while `OpenACC` on GPU accelerators manifests three levels of parallelism ; gang, worker and vector–are presented in more details in the next chapter, multi-level parallelism is not yet supported on multi-core systems. Hence, the collapse clause is ignored, so only the outer loop is parallelized at gang level. The worker level of parallelism is ignored; PGI is still exploring how best to generate parallel code that includes gang, worker and vector parallelism.

Is it possible to create a unified binary for GPU and multi-core ? The current release of `OpenACC` does not support a unified binary. Actually, we cannot use GPU compiling option `-ta = tesla` and multi-core one `-ta=multicore`. Hence, putting `-ta = tesla, multicore` is not yet supported. This feature could be important on a cluster where some of the nodes are GPU-accelerated and other nodes are not.

4.8.2 NT2 vs SIMD

From programming productivity, `NT2` requires much more time especially at debugging stage. Unfortunately, the tool is still new to provide enough documentations and details for the programmer. In the other hand, we did not meet any issues while programming with `SIMD` intrinsics. With Intel processor, users can refer to Intel intrinsics guide available online ([110]). With ARM CPUs, some advanced instructions are not available such as the SAD operations which has to be then implemented by the programmer based on the available intrinsics as we did in section 4.3.1.

From performance point of view, unfortunately, `NT2` did not manage to accelerate the selected functions (`CDD, CSM`). Despite the fact that we obtained a small speedup with `CDD` function on Intel processors by enabling the `OpenMP` flag, The performances globally are significantly low. The vectorization with the `colon "_"` option which is supposed to unroll the instructions in a `SIMD` model did not work.

4.9 Discussion

In this chapter, we investigated and evaluated different parallel tools on CPU-based systems. In the first part, we tested two directives-based approaches for shared memory parallelization: `OpenMP` and `OpenACC`. The programming model is similar with both techniques. The main purpose behind these approaches is to accelerate the process of parallelizing algorithms with a **minimal** cost of code rewriting. The original code (`C++`) is maintained and directives are added to give hints to the compiler. We obtained important performances and speedup with both approaches. While with OpenMP the process of parallelization was straightforward, with OpenACC, we met some issues on parallelizing CSM function requiring deeper analysis to cope with those issues. However, from performance point of view, OpenACC often outperforms OpenMP due to the additional optimizations applied by the compiler such as *auto-vectorization* and *loop unrolling*.

In the second part, we investigated two techniques of vectorization based on SIMD model. The first approach consists on rewriting the code by using the SIMD intrinsics. The second technique–NT2– relies on automatic code generation after rewriting the code with appropriate options to give hints to the compiler. The obtained results show that automatic generation of accelerated image processing algorithms with NT2 are not yet enough mature to provide the required performances compared to employing the SIMD intrinsics explicitly.

5

Kernel-Level Optimizations

GPU Implementation

Parallel programming with compiler directives, such as `OpenMP`, is widely employed in scientific computing to accelerate algorithms. It is mainly used when parallelism appears in regular repeated operations such as `C/C++ for` loops. `OpenACC` [12] is a compiler directives-based tool that allows parallel computing on different accelerators such as GPUs and multi-cores . In contrast to the well known mainstream GPU programming tools, such as `CUDA` [13] and `OpenCL` [111], where more explicit compute and data management is necessary, porting GPU-based applications with `OpenACC` requires only code annotations without any significant changes in the original code. This allows considerable simplifications and productivity improvements with existing applications particularly if heterogeneous architecture–with different computing units– is targeted.

This chapter concentrates on optimizing and evaluating the stereo matching algorithm at kernel-level on `GPUs` platforms. Two different programming `APIs` are used : `CUDA` and `OpenACC`. Different optimizations at several levels are tested. To understand the contributions and limitations of each employed tool, an evaluation of the obtained results is performed with both `CUDA` and `OpenACC`. Three research questions are discussed in this chapter: (1) Is `OpenACC` programming time shorter than `CUDA` for the same parallelizable problem? (2) Is `CUDA` obtained performance (execution time) better than `OpenACC` performance? (3) Does `OpenACC` provide the same optimizations as `CUDA`?

We conduct this investigation and evaluation between `CUDA` and `OpenACC` mainly based on the following two factors: (1) `CUDA` is one of the most popular parallel programming tools on GPUs and `OpenACC` is an easily learned and simplified high-level parallel language, especially for parallel programming beginners with basic knowledges on parallel computing; (2) One motivation for `OpenACC` is to simplify low-level parallel language such as `CUDA`.

5.1 GPU Architecture Overview

Graphics Processing Units (GPUs) were originally designed to perform rendering and shading of 2D and 3D graphics data. They have been used to accelerate graphics operations based on mathematical computing required in 3D games such as geometry shading, bilinear, texture mapping and depth buffering. It did not take long time for programmers to realize that these powerful co-processors can also be used for applications other than computer graphics. Harris [112] introduced in 2003 the term General Purpose computations on GPUs (GPGPU) to describe non-graphics applications running on GPUs.

The programming paradigm of GPUs has changed dramatically when the two main GPU manufacturers, NVIDIA and AMD, modified the hardware architecture. They changed dedicated graphics rendering pipeline to a multi-core computing platform. They implemented shader algorithms in software running on these cores, and explicitly support general-purpose computations on GPUs by offering programming languages and software development toolchains.

While CPUs are designed for low latency computations, GPUs are optimized for high throughput. Low latency on memory-intensive applications typically requires large caches, which use a lot of silicon area. Additional transistors are used to greater effect in GPU architectures because they are applied to additional processors that increase throughput. In addition, programmable GPUs are inexpensive, readily available and compatible with multiple operating systems and hardware architectures.

5.2 Performance Bounds on GPUs

GPUs are considered as very interesting computing platforms for many algorithms due to their pure computing power. One might expect a higher speed up computations of GPUs, but as the examples in the following sections show, this is not the case for many applications. The reason is that in order to make use of the computational power of GPUs, applications need to fulfill some conditions to avoid some bottlenecks affecting

GPU performance. Usually the performance of an application is limited by one or more factor. Here we discuss the most important bottlenecks.

Effective memory bandwidth usage Bandwidth is the rate at which data can be transfered. It is one of the most important gating factors for performance. To measure performance accurately, it is useful to calculate *theoretical* and *effective* bandwidth [113]. When the later is much lower than the former, the developer has to review the implementation to increase the bandwidth which should be the primary goal of optimization efforts.

$$Mem\_bandwidth(GPU) = memFrec(MHz) \times memWidth(Byte) \times 2 \tag{5.1}$$

Equation 5.1 is used to compute the theoretical bandwidth. The factor of 2 in the numerator is due to the double data rate, which means that data is transfered on both the rising and falling edges of the clock signal within computer bus operations. Take the Quadro M2000M used in this work as an example. Its memFreq is 2505 MHz, the memWidth is 128 bits (16 bytes), hence its theoretical memory bandwidth is :

$$Mem\_bandwidth(Quadro\,M2000M) = 2505 \times 16 \times 2 = 80.16\,GHz \tag{5.2}$$

Equation 5.3 is used to compute the effective bandwidth. Here, the effective bandwidth is in GB/sec, B_r is the number of bytes read from global memory per kernel. B_w is the number of bytes written to global memory per kernel, and **time**, is the time taken by the kernel in seconds.

$$Effective\_bandwidth(GPU) = \frac{B_r + B_w}{time} \tag{5.3}$$

Memory access pattern Memory-access patterns is crucial point to take into consideration on GPU-based systems compared to CPU-based ones. Indeed, most of the chip area in GPUs consists of ALUs while in CPUs, a large part of the chip area is reserved to fast caches that reduce load and store latencies. Computations that can keep the active set of data in the available registers benefit from the large computational power of the ALUs. However, the high latencies of device-memory loads and stores result in huge performance penalties in applications that cannot.

Some applications can use the shared memory on NVIDIA GPUs, Fermi GPUs make this easy by using a configurable amount of shared memory as transparent cache. Shared memory fits situations where the same data is required by all threads. However, if each thread requires different data in cache, and the amount of available shared memory is not sufficient then the compilers use device memory for register spills. Another way to deal with high memory latencies is to run more threads and thus hide the latencies. Note that this comes at the price of a smaller number of registers per thread and even higher requirements on data-level parallelism.

Data transfer between host memory and device memory Another potential bottleneck is data transfer between host memory and device memory. Transferring data between the host and device is a very costly move. It is not uncommon to have code making multiple transactions between the host and device without the programmer's knowledge. In order to get the most bang for your buck in your application, you really need to minimize the host↔device data transfers. Cleverly structuring code can save tons of processing time. Also, it is imperative to understand the cost of these host↔device transfers. In some cases, it may be more beneficial to run certain algorithms or pieces of code on the host due to the costly transfer time associated to transferring data to the device.

5.3 Optimizations Techniques on GPUs

In order to get rid of the aforementioned limitations, we apply a set of optimizations to our application presented in the CUDA Best Practice guide [114]. The proposed optimizations are classified according to their importance and impact on obtained performance. We selected to classify them into **high-priority** and **medium-priority** optimizations as follow:

1. **High-priority**

    - *Maximize parallel execution* At device level, we should assign enough threads per block and enough blocks per grid such that the SMs are maximally utilized. At SM level, the number of threads per block should be a multiple of warp size(32) to avoid warps with inactive threads. Also, Depending on the register usage and the shared memory usage, block size must be selected to give the best occupancy.

- *Minimize global memory use* global memory is the slowest type of memory to avoid whenever it is possible. The solution here is straightforward: wherever possible, allocate memory once at the beginning of an application and then reuse that memory in each kernel invocation.
- *Minimize data transfer between the host and the device memories* To reduce the severity of data transfers between the host and the device, developers should attempt to perform as much computation on the GPU as possible.
- *Ensure coalesced accessing to global memory whenever possible* The global memory of the GPU is accessed in blocks of 32, 64 or 128 bytes, so the number of accesses to satisfy a warp depends on how data are grouped.
- *Use shared memory to avoid redundant transfers from global/device memory* Shared memory accesses are faster than global/device memory ones. Hence, we should use shared memory whenever a set of data is shared by threads within the same block. However, the size of shared memory is limited per block and can affect the number of executed warps.

2. **Medium-priority**

   - *Ensure access to shared memory without bank conflicts* Shared memory is divided into several banks with equal sized width (32/64 bits depending on the compute capability). One memory bank can serve one thread once. If more than one thread access the same bank simultaneously, then there will be **bank conflicts** and threads in a warp are **serialized**.
   - *Avoid different execution paths with the same warp (branch divergence)* GPU threads are grouped into sets of 32 named warps in CUDA language. When a task is being executed over a warp, the 32 threads carry out this task simultaneously. However, due to conditional flow instructions in the code, not all the threads will perform the same operation, so the different tasks are executed sequentially, giving rise to a significant loss of efficiency.

5.4 Nvprof Metrics

In the following experiments, NVIDIA `nvprof` profiler is used to profile our applications with `CUDA` and `OpenACC`. This profiler provides a set of metrics and events [115] to analyze the application for better understanding the obtained results. In this work, here are the employed metrics and events in the next experiments:

- **sm_efficiency** The percentage of time at least one warp is active on a multiprocessor averaged over all multiprocessors on the GPU.
- **achieved_occupancy** Ratio of the average active warps per active cycle to the maximum number of warps supported on a multiprocessor
- **gld_efficiency** Ratio of requested global memory load throughput to required global memory load throughput expressed as percentage
- **gst_efficiency** Ratio of requested global memory store throughput to required global memory store throughput expressed as percentage
- **gld_throughput** Global memory load throughput
- **gst_throughput** Global memory store throughput
- **L2_read_throughput** Memory read throughput seen at L2 cache for all read requests
- **L2_write_throughput** Memory write throughput seen at L2 cache for all write requests
- **dram_read_sectors** Number of read requests sent to all sub-partitions of all the DRAM units. This event is not directly available with `nvprof`. The later provides a metric for each sub-partition such as *fb_subp0_read_sectors* for sub-partition 0 and *fb_subp1_read_sectors* for sub-partition 1. Hence, dram_read_sectors is no more than the sum of reads on all partitions in the DRAM.

5.5 Experimental Design

Different NVIDIA GPUs are used with different specifications such as compute capability and number of cores. It is worth noting that users are not allowed to use a specific number of cores in one GPU since it is not provided to do so and all of the `CUDA` cores in one graphic card always run together. Hence, to decrease or increase the number of cores, we have to change and use a different GPU. Table 5.1 shows the employed GPUs with their most important specifications. We use two fixed platforms; the GeForce GTX780M and the Quadro M2000M, and two embedded platforms : NVIDIA `Jetson`

Table 5.1: GPUs Specifications

GPU	Generation	C.C[1]	#SM	#cores	CUDA
GeForce GTX780M	Kepler	3.0	8	1536 (192/SM)	8.0 / 7.5
Quadro M2000M	Maxwell	5.0	5	640 (128/SM)	8.0 / 7.5
Tegra X1	Maxwell	5.3	2	256 (192/SM)	8.0
GK20A(k1)	Kepler	3.2	1	192 (192/SM)	6.5

[1] Compute Capability

`TK1` and `Tegra TX1`. In the following experiments and results, we refer to Jetson TK1 as GK20A corresponding to the name of its GPU.

It is worth noting that with `CUDA`, the four platforms have been employed since all required softwares such `CUDA` compiler (`nvcc`) and profilers (`nvprof, nvvp`) are available on both architectures; Intel and ARM. However, with `OpenACC`, for our best knowledge, there is no available compiler for ARM architectures. There are appreciated open source projects such as accULL [116], Omni [117] and IPMACC [118] compilers. These projects aim to provide a suitable compiler for `OpenACC` on ARM architectures. Unfortunately, we tested all these compilers and we did not manage to launch any `OpenACC` sample on ARM, some of them were not even installed completely.

In our experiments, we use KITTI [101] raw stereo data base each time an optimization is applied to the algorithm for discussion and analysis, then, we apply the obtained optimized algorithm on 9 pairs of image with different resolution from Middlebury Stereo Datasets [1].

5.6 OpenACC and CUDA Optimization of CDD Function

This section describes the implementation of `CDDL` and `CDDR` functions on GPUs. Since these two functions have almost the same execution time, we choose to present the results of only one function referred as `CDD`. Different optimizations are proposed targeting several aspects of parallelization in `GPUs` such as global memory accesses and shared memory usage by following the degree of priority as discussed previously (section 5.2).

Algorithm 6 First Parallel Stereo Matching of A-ELAS (CDDL) with CUDA

```
Require: LD[H×W×16], RD[H×W×16], PRIOR[256], DMIN[H], DMAX[H], PD[H×W]
 1: for row=2 to H-2 parallel do
 2:     for col=2 to W-2 parallel do
 3:         for d=DMIN[row] to DMAX[row] do            ▷ disparity reasearch distance
 4:             SAD = PRIOR[|d-PD[row][col]|]
 5:             for i=0 to 15 do                  ▷ go through pixel's descriptor elements
 6:                 SAD += abs(LD[row][col][i] - RD[row][col-d][i])
 7:             end for
 8:             if SAD<$SAD_{min}$ then
 9:                 $SAD_{min}$ = SAD
10:                 disparity = d
11:             end if
12:         end for
13:         D[row][col] = disparity
14:     end for
15: end for
```

While CUDA provides different optimizations to get the required performances, it is not evident to reach those performances from the first try. Actually, developers usually write different versions and test a variety of approaches and optimization to reach the final desired performance.

5.6.1 First Naive Implementation

5.6.1.1 CUDA Kernel

The first implementation consists on taking the scalar version (C++) and writing a CUDA kernel at pixel granularity, i.e., each thread computes the disparity of one pixel. The kernel takes pixels' descriptors of left and right images as inputs in addition to the prior data and image research boundaries (DMIN, DMAX) as shown in Algorithm 6. Each descriptor is of size $H \times W \times 16$, where W and H are the width and height of the input stereo images respectively. The size is multiplied by 16 referring to the 16 gradients used to describe each pixel.

Algorithm 6 illustrates the approach employed to parallelize CDDL function with CUDA. For more details concerning the employed variables, refer to section 4.1.2 in chapter 4. The same algorithm is employed for CDDR function except at line 6, where the

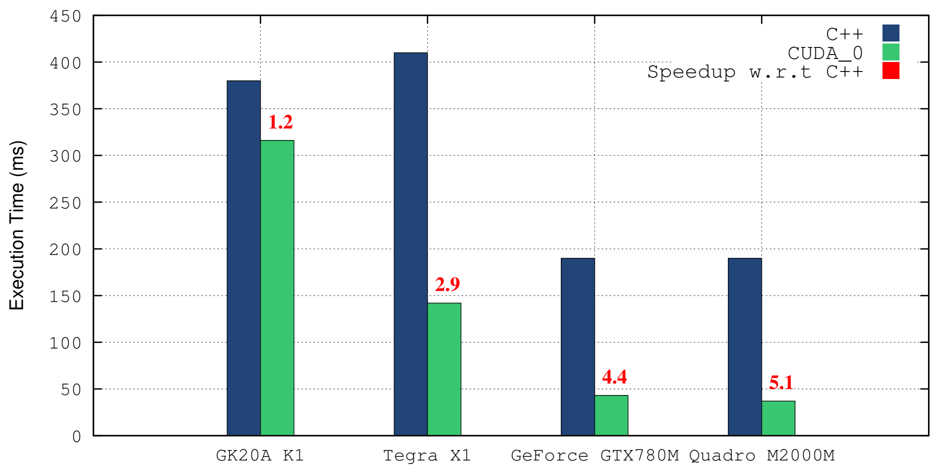


Figure 5.1: Execution Time of First `CDD CUDA` Kernel on KITTI(1242x375)

left and the right descriptors (LD, RD) are permuted since we compute the backward disparity in this case. Parallelization is done at two levels corresponding to the first and second loops. Hence, a 2D `CUDA` grid is created. Concerning the `CUDA` grid size, a block of 32×4 threads is employed in this experiment. The impact of block size will be discussed in further experiments.

Figure 5.1 depicts the obtained results on KIITI dataset. We notice different results and speedup among the four architectures. The best performance is obtained on fixed platforms with a speedup of **5** on Quadro M2000M and more than **4** on GeForce GTX780M. On Tegra X1, a speedup of almost **3** is achieved while on GK20AK1, the speedup is too low (`1.2`).

In all architectures, the obtained speedup is far away from the required one and far from the available hardware performance. To determine the bottlenecks of this kernel, we gathered some metrics from **`nvprof`** profiler as shown in Table 5.2. These metrics have been presented previously in section 5.4. All **`nvprof`** metrics and events are presented in `CUDA` guide [115] on how using `nvprof` profiler.

Global memory load and store throughputs are respectively represented by *gld_ throughput* and *gst_ throughput* metrics. Equations 5.4 and 5.5 give the formula to compute these metrics. Equations 5.6 and 5.7 show respectively how to compute the total number of read and write requests to L2 cache. We notice that in those equations the memory is partitioned to 4 sectors (subp0, subp1, subp2, subp3) but this may not

Table 5.2: CDD CUDA First kernel results on KITTI(1242x375): GPU execution time, `Nvprof` Metrics and Events

Nvprof Metric/Event	GK20A(K1)	Tegra X1	GeForce GTX780M	Quadro M2000M
sm_efficiency	100%	99.68%	99.70%	99.88%
achieved_occupancy	0.94	0.93	0.88	0.94
gld_throughput (measured)	**22.49 GB/s**	**57.81 GB/s**	**200.39 GB/s**	**213.57 GB/s**
gst_throughput	**5.62 MB/s**	**15.51 MB/s**	**53.85 MB/s**	**57.68 MB/s**
gld_efficiency	6.17%	6.17%	6.16%	6.16%
gst_efficiency	84.21%	84.21%	84.21%	84.21%
L2_read_throughput	**22.5 GB/s**	**57.83 GB/s**	**200.39 GB/s**	**215.48 GB/s**
L2_write_throughput	**6.01 MB/s**	**15.51 MB/s**	**53.86 MB/s**	**57.68 MB/s**
L2_total_read_sector_queries	265517048	265771809	265378107	267789110
gpu_time	316 ms	142 ms	43 ms	37 ms
gld_throughut_computed (5.4)	26.88 GB/s	59.89 GB/s	197.49 GB/s	231.6 GB/s

be always the case, it depends on the hardware employed.

$$gld\_throughput = ((128 \times L1\_global\_load\_hit) + (L2\_total\_read\_sector\_queries) \times 32 - (L1\_local\_ld\_miss \times 128))/gputime \quad (5.4)$$

$$gst\_throughput = (L2\_total\_write\_sector\_queries) \times 32 - (L1\_local\_ld\_miss \times 128))/gputime \quad (5.5)$$

$$L2\_total\_read\_sector\_queries = L2\_subp0\_read\_requests + L2\_subp1\_read\_requests + L2\_subp2\_read\_requests + L2\_subp3\_read\_requests \quad (5.6)$$

$$L2\_total\_write\_sector\_queries = L2\_subp0\_write\_requests + L2\_subp1\_write\_requests + L2\_subp2\_write\_requests + L2\_subp3\_write\_requests \quad (5.7)$$

We used Equation 5.4 to compute the global load throughput and compared it to the measured results with `nvprof`. The result is shown in table 5.2. We got close results with a small difference between the measured (`gld_throughput_measured`) and the computed (`gld_throughput`) global load throughputs.

We notice also that in Equations 5.4 and 5.5 `L1` loads are employed to compute those metrics. However, from table 5.2, we notice that `gld_throughput` and `L2_read_throughput`

Algorithm 7 First Parallel Stereo Matching of A-ELAS (CDDL) with OpenACC

```
Require: LD[H×W×16], RD[H×W×16], PRIOR[256], DMIN[H], DMAX[H], PD[H×W]
1: #pragma acc kernels copyin(...)  copyout(...)  ▷ data transfer host↔device
2: #pragma acc loop independent
3: for row=2 to H-2 do
4:     #pragma acc loop independent
5:     for col=2 to W-2 do
6:         compute_disparity[row][col]           ▷ Algorithm 6 line 3→ line 13
7:     end for
8: end for
```

metrics give the same values, the same observation concerns `gst_throughput` and `L2_write_throughput` metrics, they are both equal. In other words, the cache L1 **is not used by default**.

5.6.1.2 OpenACC kernel

We implemented the same kernel with `OpenACC` directives. While multi-level parallelization is not possible with `OpenACC` on multi-core (release 16.10), fortunately, it is supported on GPU-based architectures. As first version, we employed `kernels` construct and hence let freedom to the compiler to configure `CUDA` grid (block size and number). We also employed `loop` clause to guide the compiler and explicitly mention that we want to parallelize at two levels as illustrated on Algorithm 7. In contrast to `OpenACC` on multi-core machines where data transfers between the host and the device memory are hidden, on GPU, they are managed explicitly by the programmer. Clause `copyin` is used to copy input data from the host to the device, and clause `copyout` is used to transfer data to the host.

It is important to notice that each time we compare `OpenACC` kernel to its corresponding `CUDA` kernel, both kernels are supposed to perform the same instructions and if any optimization is presented, it is applied also to both kernels. If there is any difference, it is presented and details are given. For this `OpenACC` first kernel, the only difference is the `CUDA` grid size. In `CUDA` kernel, we used a block of $[32 \times 4]$ threads. With `OpenACC`, we choose at the beginning to let the compiler configure `CUDA` grid through the `kernel` clause. The purpose behind this is to understand how the compiler sets the size of blocks.

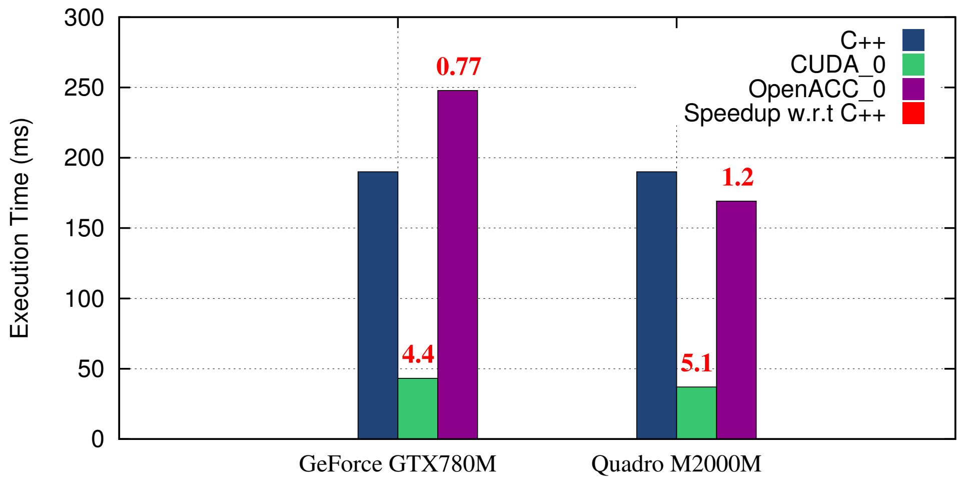


Figure 5.2: Execution Time of First `CDD OpenACC` Kernel vs First `CDD CUDA` kernel on KITTI(1242x375)

Table 5.3: OpenACC Configuration with Kernel Clause

GPU	Grid_Size	Block_Size	Reg[1]	SSMem[2]	DSMem[3]
GeForce GTX780M	(1242 375 1)	`(32 1 1)`	37	16B	128B
Quadro M2000M	(1242 375 1)	`(32 1 1)`	37	16B	128B

[1] Number of registers used by one thread
[2] Static Shared Memory
[3] Dynamic Shared Memory

Figure 5.2 depicts the execution time of this first `OpenACC` kernel versus the first naive `CUDA` kernel of `CDD`. We notice clearly that `OpenACC` kernel is too slow compared to `CUDA` one on both architectures. Actually `OpenACC` kernel is even slower than the scalar version on GeForce GTX780M architecture since the speedup is less than 1. To understand this important difference, we collected some data from `PGI` compiler and `nvprof` profiler as shown in Table 5.3. The most important information concerns the grid configuration and particularly the block size. The later is configured along colons only (1D) of $[32 \times 1]$ threads. In `CUDA` kernel, we rather used a 2D block of $[32 \times 4]$ threads. Another observation concerns the use of static (SSMem) and dynamic (DSMem). It seems that the `PGI` compiler tries to optimize the kernel through shared

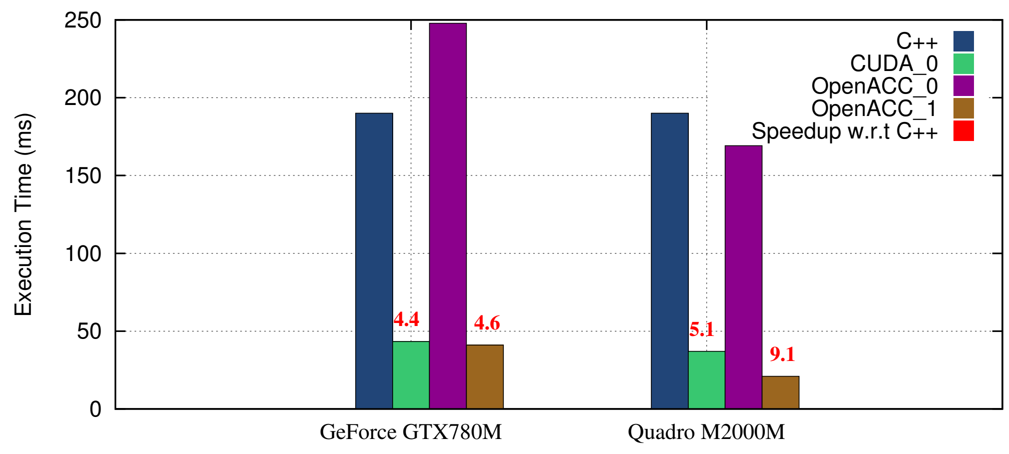


Figure 5.3: Execution Time of Optimized First `CDD` `OpenACC` Kernel vs First `CDD` `CUDA` on KITTI(1242x375)

memory. With this version, we can conclude that `PGI` compiler could configure a 2D `CUDA` grid via `kernel` clause.

Fortunately, `OpenACC` allows users to manually set the size of `CUDA` block. To do so, we need to add `gang()` and `vector()` clauses to both for loops. Listing 5.1 shows how we use `gang()` and `vector()` clauses. Figure 5.3 depicts the obtained execution of `OpenACC` kernel (`OpenACC_1`) by setting `CUDA` block size to [32×4]. In GeForce GTX780M GPU, we get the same execution time as `CUDA_0` kernel. However, in Quadro M2000M, `OpenACC_1` kernel is faster than `CUDA_0` one. The next section explains this difference.

```
#pragma acc kernels copyin(inputs) copy(output){
  #pragma acc loop independent gang(94) vector(4)
  for(int32_t row=2; row<H; row++)
    #pragma acc loop independent gang(39) vector(32)
    for( int32_t col=0; col<W; col++){
        // compute disparity at pixel (row, col)
    }
}
```

Listing 5.1: CUDA Grid Set with gang and vector Clauses

While `gang()` clause is used to set the number of blocks, `vector()` clause sets the number of threads within a block. If we want to use a block of [32 × 4] threads, then, as shown in listing 5.1, `vector(32)` is used in the first `for` loop to set the size of the block along the image's width (x direction), and similarly, `vector(4)` is used in the

Table 5.4: Some nvprof Metrics and Events of OpenACC_0 and OpenACC_1 Kernels

Nvprof Metric/Event	GTX780M V0	GTX780M V1	M2000M V0	M2000M V1
sm_efficiency	100.00%	99.10%	100.00%	99.76%
achieved_occupancy	0.25	0.68	0.49	0.47
gld_throughput	`6.21 GB/s`	`199.09 GB/s`	`15.64 GB/s`	`414.80 GB/s`
gst_throughput	57.44 MB/s	51.98 MB/s	84.27 MB/s	99.35 MB/s
L2_read_throughput	`6.21 GB/s`	`199.09 GB/s`	`6.22 GB/s`	`40.38 GB/s`
L2_write_throughput	57.44 MB/s	51.98 MB/s	84.27 MB/s	99.36 MB/s
gld_efficiency	36.57%	6.20%	21.13%	5.57%
gst_efficiency	12.50%	83.81%	12.50%	83.81%
dram_read_sectors	4.33×10^6	1.17×10^6	11.2×10^6	1.86×10^6

second `for` loop to set the size of the block along the image's height (y direction). In this experience, we use KITTI dataset with $[1242 \times 375]$ resolution. Hence, with a block size of $[32 \times 4]$, we need 39 blocks along the x direction $(1242 \div 32)$, and 94 blocks along the y direction $(375 \div 4)$. Listing 5.1 gives a code snippet of this configuration.

Table 5.4 gives some metrics and events returned by `nvprof` on both architectures. kernels `OpenACC_0` and `OpenACC_1` are referred as V0 (version 0) and V1 (version 1) respectively. The most important result concerns the throughput of loads from global memory (`gld_throughput`). The latter has been increased significantly in `OpenACC_1` on both architectures. The same observation concerning `L2_read_throughput`. The number of reads from dram are lower compared to kernel`OpenACC_0`. Finally, we notice that on Quadro M2000M GPU, `gld_throughput` is greater than `L2_read_throughput` in both kernels, where as, in GeFore GTX780M they are both equal. This difference is related to L1 cache usage which will be discussed in the next section.

5.6.2 Optimization 1 : L1 Caching of Global Loads

The first optimization of CUDA developers targets global memory accesses. This may happen through *explicit caching* of global data in shared memory. However, sometimes algorithms have memory access patterns that cannot be coalesced, and that do not fit shared memory. Fermi GPUs have an automatic L1 cache on each streaming multiprocessor (SM) that can be used in this case (irregular access pattern). First-generation

Table 5.5: L1 Cache/Shared Memory Configuration Options on Compute 3.x Devices

parameter	L1 Size	CC 3.0-3.5 Shared Size	CC 3.7 Shared Size
`cudaFuncCachePreferShared`	16	48	112
`cudaFuncCachePreferL1`	48	16	80
`cudaFuncCachePreferEqual`	32	32	96

Table 5.6: GPUs Memory Specifications

Specification	GK20A(k1)	Tegra X1	GeForce GTX780M	Quadro M2000M
Memory Clock Rate	924 MHz	13 MHz	2500 MHz	2505 MHz
Memory Bus Width	64 bits	64 bits	256 bits	128 bits
globalL1CachSupported	1	1	0	1
Peak Memory bandwidth	14.78 GB/s	0.2 GB/s	160 GB/s	80 GB/s
L2 cache size	128 KB	256 KB	512 KB	2048 KB
L1 cache size	48 KB(L1/shared)	48 KB(L1/shared)	48 KB(L1/shared)	48 KB(L1/shared)

Kepler GPUs have an automatic L1 cache on each SM, but it only caches local memory accesses. In these GPUs, the absence of automatic L1 cache for global memory is replaced by a separate 48 KB read-only (texture) cache per SM. Developers can configure the split between L1 and shared memory via the `cudaDeviceSetCacheConfig` and `cudaFuncSetCacheConfig` API calls (Table 5.5).

NVIDIA has enabled L1 caching of global memory on some GPUs such as GK110B, GK20A and GK210 chips. The Tesla K40 (GK110B), Tesla K80 (GK210) and Tegra K1 (GK20A) all support by default this feature. To query whether a GPU supports caching global memory operations or not, developers can use `cudaGetDeviceProperties` and check the `globalL1CacheSupported` property. Actually, examining only the Compute Capability is not enough; Tesla K20/K20x and Tesla K40 both support Compute Capability 3.5, but only the K40 supports caching global memory in L1. On Kepler GPUs, even if they do support caching global memory in L1, by default, it is disabled. Programmers have to enable caching by passing `-Xptxas="-dlcm=ca"` compiling option to `NVCC`.

In Maxwell GPUs of Compute Capability 5.0, the L1 cache and texture/read-only cache are combined into a single unit, with a separate dedicated 64KB shared memory

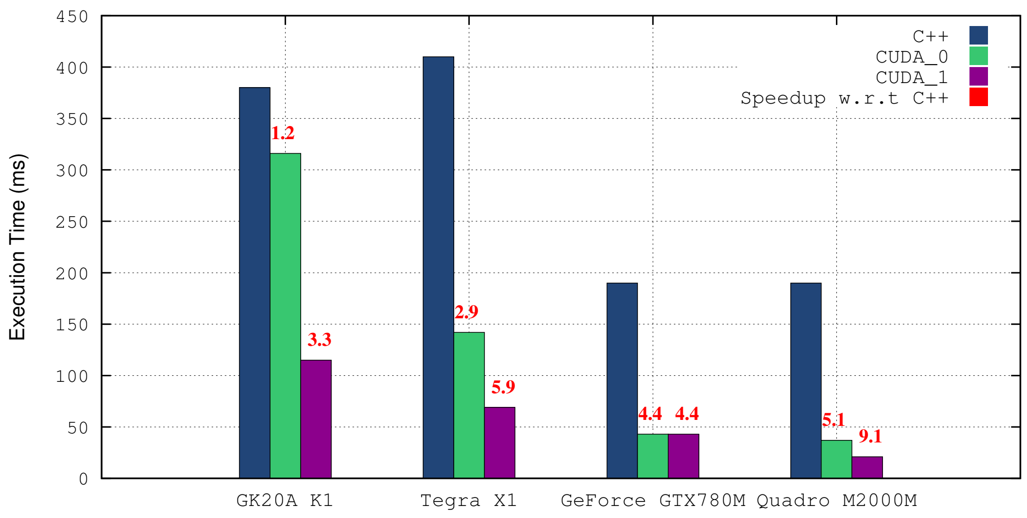


Figure 5.4: Execution Time of `CDD CUDA` Kernel with L1 Caching on KITTI(1242x375)

unit. Read-only global memory accesses can be cached with const `__restrict__` pointer qualifiers, or via the `__ldg()` intrinsic. Local memory operations are only cached in L2. In second-generation Maxwell GPUs with Compute Capability 5.2, global memory caching can be done through the `-Xptxas="-dlcm=ca"` option to NVCC.

Table 5.6 gives information concerning global and cache memories in the employed platforms in these experiments. Other information such as Compute Capability have been presented previously in table 5.1. We enabled global memory caching on L1 according to each architecture, its corresponding Compute Capability and GPU's generation.

5.6.2.1 CUDA Kernel

Based on the aforementioned details, we enabled the L1 on the employed architectures. Figure 5.4 depicts the obtained results. We observe better performance when global memory accesses are cached through L1 in all architectures except in NVIDIA GeForce GTX780M. Actually, the latter is of Compute Capability 3.0 belonging to the first generation of Kepler GPUs where L1 cache is not enabled. As explained previously, NVIDIA enabled L1 cache only on specific first generation Kepler GPUs such as GK20A which is integrated in Jetson TK1. In Tegra X1, a speedup of **2** is obtained compared to the first kernel without caching. In Jetson TK1, a speedup of **3** is achieved. Finally, in Quadro M2000M, we notice a speedup of less than **2**.

Table 5.7: Nvprof Metrics of CDD CUDA Second Kernel with L1 Caching (1242x375)

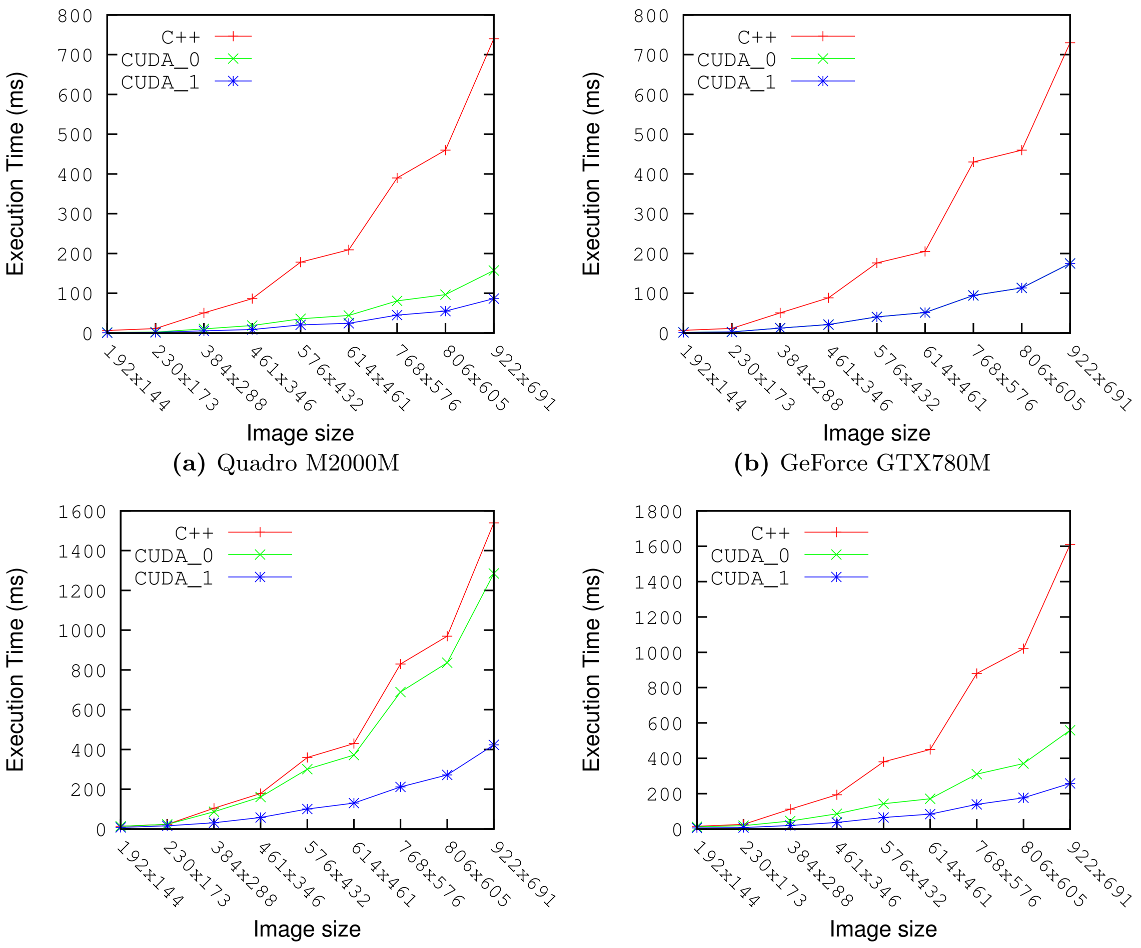
Nvprof Metric	GK20A(k1)	Tegra X1	GeForce GTX780M	Quadro M2000M
gld_throughput	22.49 GB/s	57.81 GB/s	200.39 GB/s	213.57 GB/s
gld_throughput_ca	91.46 GB/s	137.13 GB/s	200.39 GB/s	412.07 GB/s
gst_throughput	5.62 MB/s	15.51 MB/s	53.85 MB/s	57.68 MB/s
gst_throughput_ca	20.22 MB/s	33.24 MB/s	53.85 MB/s	102.12 MB/s
L2_read_throughput	22.5 GB/s	57.83 GB/s	200.39 GB/s	215.48 GB/s
L2_read_throughput_ca	962.12 MB/s	53.52 GB/s	200.39 GB/s	125.94 GB/s
L2_write_throughput	6.01 MB/s	15.51 MB/s	53.86 MB/s	57.68 MB/s
L2_write_throughput_ca	20.23 MB/s	32.42 MB/s	53.86 MB/s	102.13 MB/s

Table 5.7 presents some `nvprof` metrics with and without L1 caching. Metrics suffixed with **_ca** are collected with L1 caching enabled. In GeForce GTX780M, since L1 cache is disabled, we notice no difference between metrics with and without caching through L1. In the other systems, global memory loads (`gld_throughput_ca`) and stores (`gst_throughput_ca`) throughputs with caching through L1 are higher. We also observe that `L2_read_throughput_ca` is less than `L2_read_throughput` and `gld_throughput_ca`. The difference represents the use of L1 cache. In Kepler GPUs with Compute Capability of 3.x, we can check the use of L1 cache through `nvprof` events; `l1_global_load_hit` and `l1_global_load_miss`. In GeForce GTX780M, these events are always **null**. In GK20A GPU, they are not when L1 caching is enabled. Unfortunately, `nvprof` does not provide these events in Maxwell GPUs. Finally, the throughput of writing in L2 memory (`L2_write_throughput_ca`) has been improved when L1 caching is enabled.

Since `CDDL` and `CDDR` give almost the same performance, we chose to represent only one function (`CDDL`) for referred as `CDD`. Figure 5.5 depicts the obtained performance of `CDD` on Middlebury dataset of CUDA kernel with L1 caching versus CUDA kernel without caching. As expected, in GeForce GTX780M, `CUDA` and `CUDA_CACHED` are presented with the same curve since L1 cache is not enabled in this GPU.

5.6.2.2 OpenACC Kernel

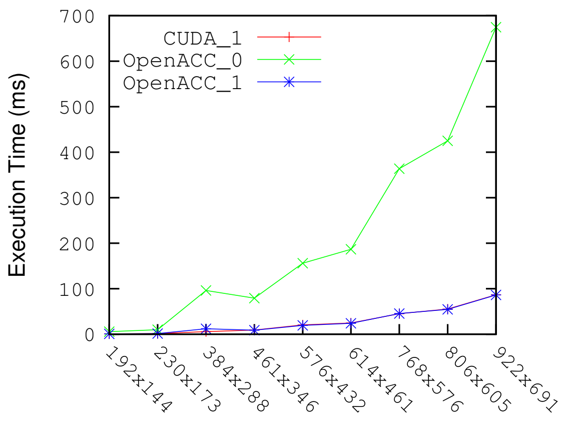
PGI compiler provides `-ta=loadcache:L1` option o enable L1 cache. We applied this option with `OpenACC_1`. Actually, we did not get any performance, in other words we



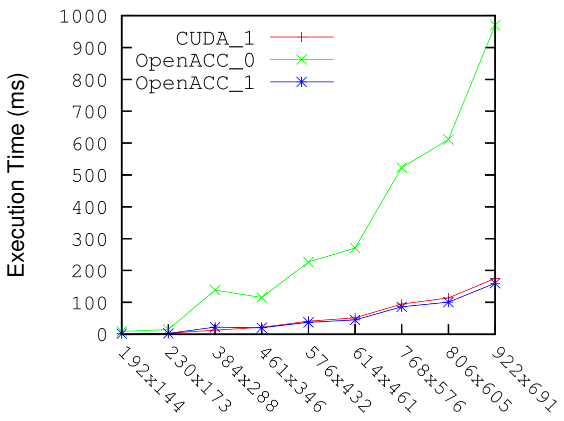
(a) Quadro M2000M **(b)** GeForce GTX780M **(c)** GK20A (K1) **(d)** Tegra X1

Figure 5.5: Execution Time of `CDD` `CUDA` Kernel with L1 Caching on Middlebury Dataset

got the same performances as with `OpenACC_1` kernel without adding this compiling option. To explain the reason, let's refer again to table 5.4. We noticed previously, `gld_throughput` is greater than `l2_read_throughput` Quadro M2000M GPU, while they are both equal in GeForce GTX GPU. As we explained with `CUDA`, this is related to L1 cache which is enabled on Quadro M2000M GPU an not on GTX780M. Hence, `pgcc` compiler uses automatically L1 cache whenever it is possible even without adding any compiling option. Figure 5.6 depicts the obtained execution time on Middlebury dataset with different resolutions. We notice that the blue curve (`OpenACC_1`) and the red one (`CUDA_1`) corresponding to `CUDA` kernel with L1 cache enabled (when it is possible), overlap. This confirms that `pgcc` compiler caches global memory access in L1 cache by default whenever it is possible.



(a) Quadro M2000M



(b) GeForce GTX780M

Figure 5.6: Execution Time of OpenACC kernels on Middlebury Dataset

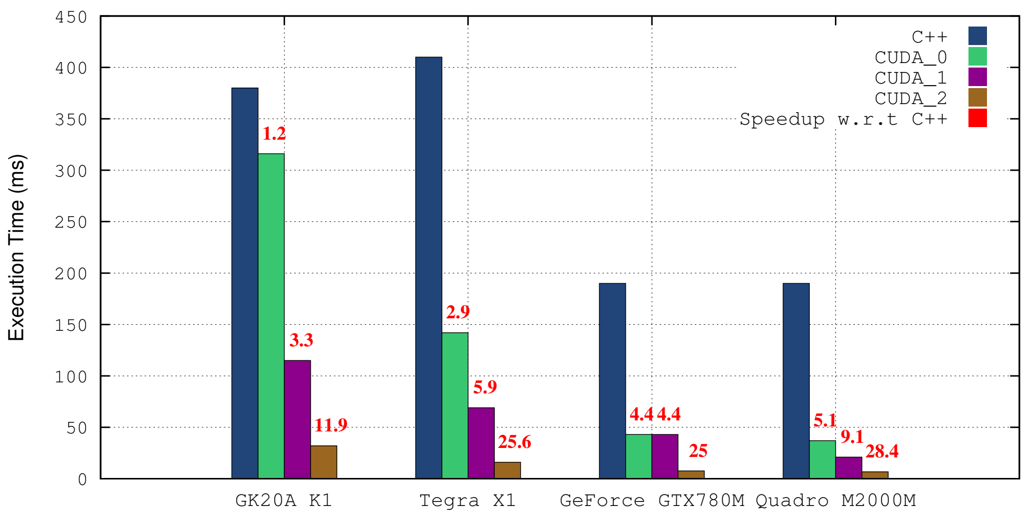
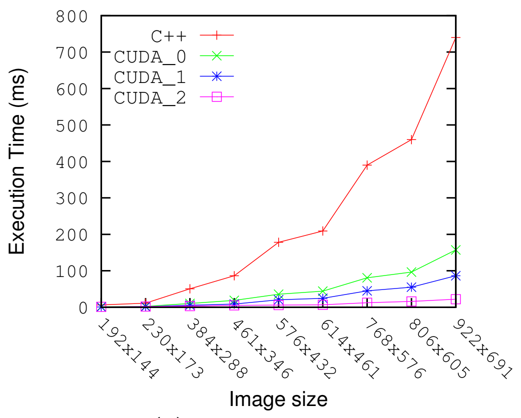


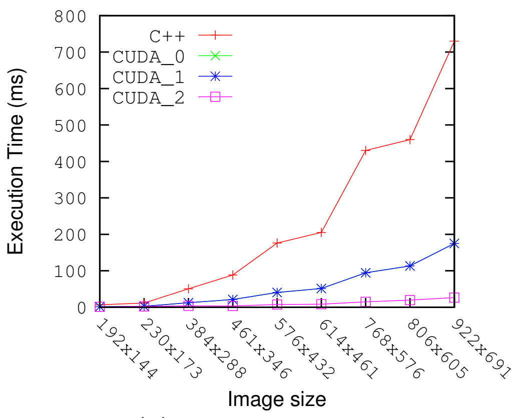
Figure 5.7: Execution Time of Third `CDD CUDA` Kernel on KITTI(1242x375)

5.6.3 Optimization 2 : Minimize Global Memory Use/Accesses

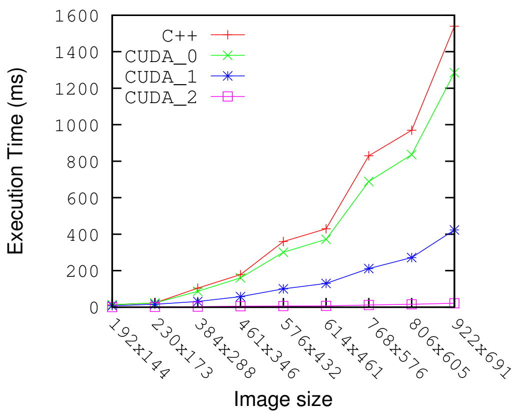
Among the optimizations with high priorities already mentioned; minimizing global memory transfers to device memory. ELAS algorithm relies on **2** descriptors' images



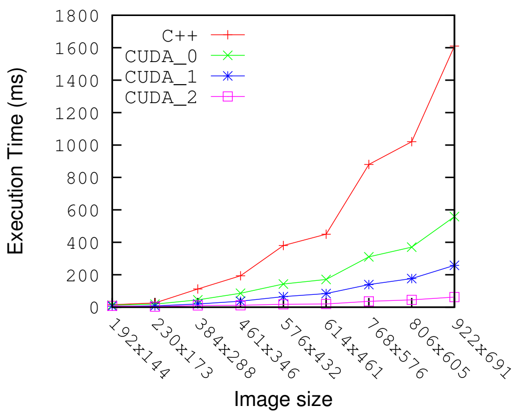
(a) Quadro M2000M



(b) GeForce GTX780M



(c) GK20A (K1)



(d) Tegra X1

Figure 5.8: Execution Time of Third `CDD` `CUDA` Kernel on Middlebury Dataset

of $Hight \times Width \times 16$ size. in this image, each pixel is described with 16 elements from Sobel horizontal and vertical gradients. We propose to work directly on Sobel images. In other words, instead of using $2 \times [Hight \times Width \times 16]$ images, we only use $4 \times [Hight \times Width]$ images; we reduce the required memory size by a factor of **8**. Also, descriptors images are first transfered to device memory before calling `CDD` or `CSM` CUDA kernels. By working directly on Sobel images, we do **zero** transfer since Sobel images are already in device memory computed from Sobel CUDA kernel.

5.6.3.1 CUDA Kernel

Figure 5.7 depicts the execution time of `CDD` CUDA kernel on KITTI dataset with the aforementioned optimization. We observe clearly that CUDA_2 kernel gives better

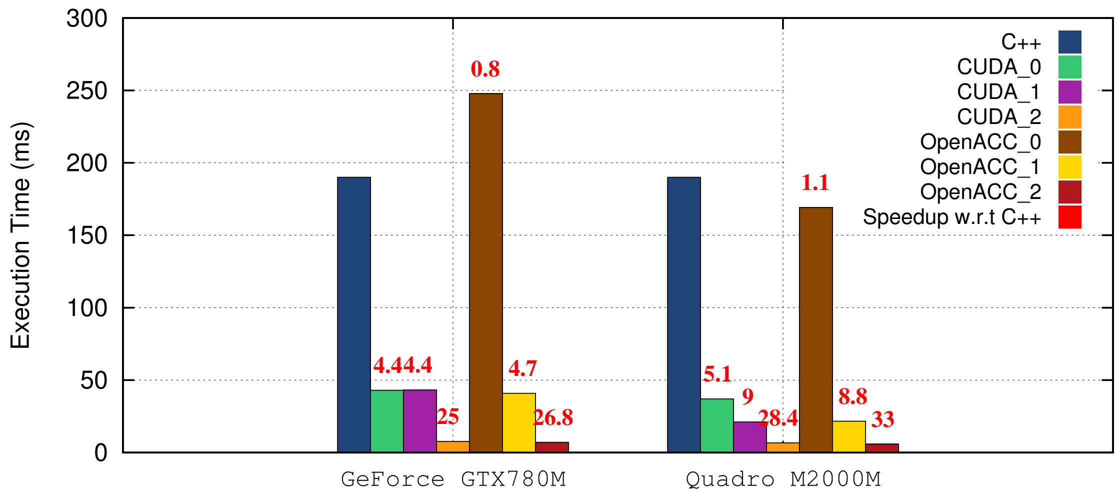


Figure 5.9: Execution Time of CDD CUDA and OpenACC kernels on KITTI(1242x375)

performance by reducing the execution time on all architectures. In Jetson K1 GPU (GK20A), we reach a speedup of almost **12**. In both Tegra X1 and GeForce GTX780M, a sppedup of **25** is achieved. Finally, the best performance is obtained in Quadro M2000M GPU with a speedup of more than **28**. These results show clearly the importance of reducing global memory transfers.

Figure 5.8 depicts the obtained performance of this kernel (CUDA_2) compared to the previous ones on Middlebury dataset with different image resolution. We observe clearly the high importance and priority of minimizing global memory accesses and transfers in order to obtain better performance of CUDA kernels.

5.6.3.2 OpenACC Kernel

We applied the same optimization to the last OpenACC kernel (OpenACC_1). Figure 5.9 depicts the obtained results with both CUDA and OpenACC. We clearly observe the importance of minimizing global memory transfers and accesses on both architectures and with both tools (CUDA, OpenACC). Also, we notice that we obtain the same performance in (OpenACC_2) kernel compared to CUDA_2 one on both architectures. These results show that OpenACC can compete CUDA in computing performance while it requires less programming efforts and time compared to CUDA.

5.6.4 Optimization 3 : Use Shared Memory

From section 5.2, the use of shared memory belongs to medium-priority optimizations. Shared memory is a very fast read and write memory. It is used whenever data is accessed by more than one thread in the same block to reduce global memory transfers and device memory accesses. In `CDD` function, the research boundaries (minimum and maximum) of the disparity in the right image (the left image is the reference) is the same for all pixels in the same image's line. Hence, all pixels in the left image access the same pixels in the right image. There is an obvious data access redundancy in the right image. Here comes the importance of shared memory use to minimize data accesses to device memory.

5.6.4.1 First Naive Implementation

We propose a first `CDD CUDA` kernel using shared memory with the following features:

- Load right horizontal and vertical Sobel gradients on shared memory.
- Use **1 thread/ 4 pixels** granularity to minimize shared memory size per block and maximize parallelism.
- Use pixel's descriptors of size 16.
- Use `CUDA` block of [32x4] threads.
- load **4KB** data on shared memory corresponding to 2 blocks of $[256 \times 8]$ each (vertical and horizontal right gradients).
- In Jetson k1, try with different configurations of L1/shared memories since they share the same physical memory block (see table 5.5).

Figure 5.10 depicts the obtained results of `CUDA_3V0` kernel which uses shared memory. The results of the previous kernel (`CUDA_2`) which uses L1 cache are also depicted. The results on Jeston K1 are depicted and discussed in the following paragraphs since we have different possible configurations of the shared memory. We observe that on GTX780M (5.10), almost no performance has been achieved compared to `CUDA_2` kernel. In Maxwell GPUs (Tegra X1, Quadro M2000M) having separate shared and L1 cache memory blocks, we notice a better performance. From `nvvp` profiler, the bottleneck of this kernel is the pattern access employed to load data from device memory to

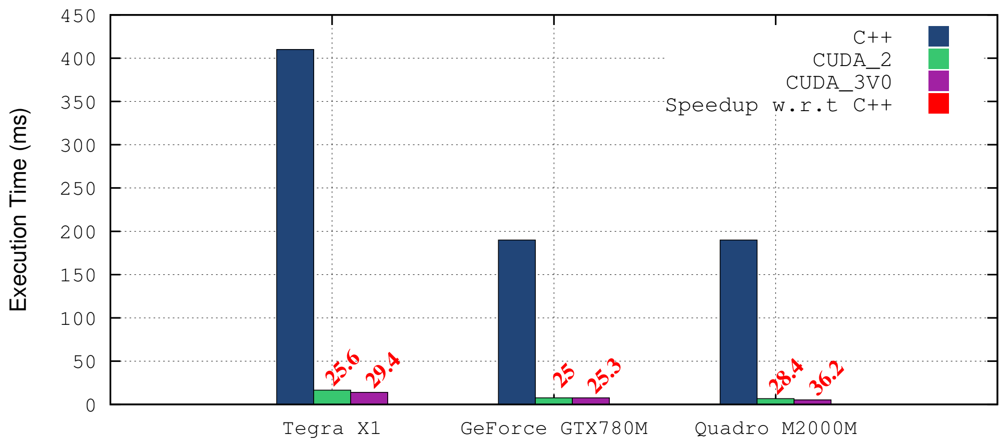


Figure 5.10: Execution Time of First Version of Third `CDD CUDA` (CUDA3_V0) Kernel with Shared Memory on KITTI Dataset

shared memory. The pattern employed to get each pixel descriptor is *not regular enough for memory coalescing.*

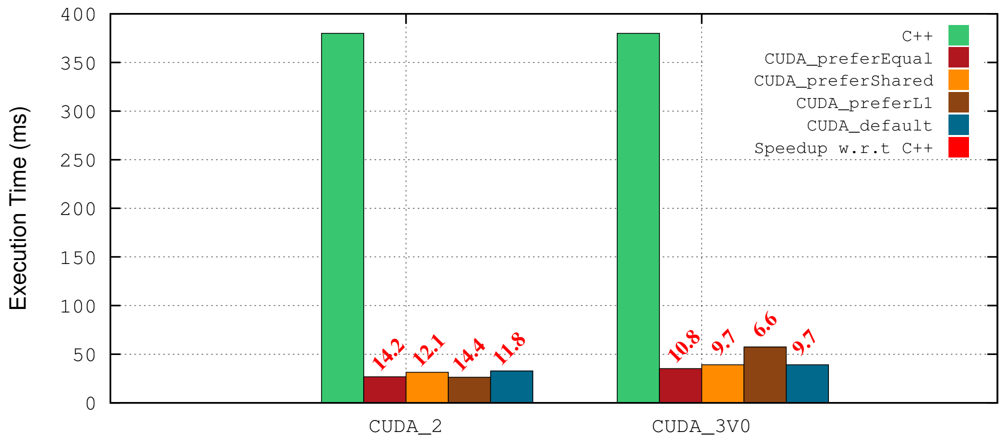


Figure 5.11: Execution Time of First Version of Third `CDD CUDA` (CUDA3_V0) Kernel with Shared Memory on KITTI Dataset, in Jetson K1

The obtained results of Jetson K1 (GK20A) are depicted in Figure 5.11. We clearly notice that this first implementation with shared memory is not optimized to benefit from the fast memory access of the shared memory. We notice that the execution time is more important with all possible configurations compared to the previous version (`CUDA_2`) without shared memory.

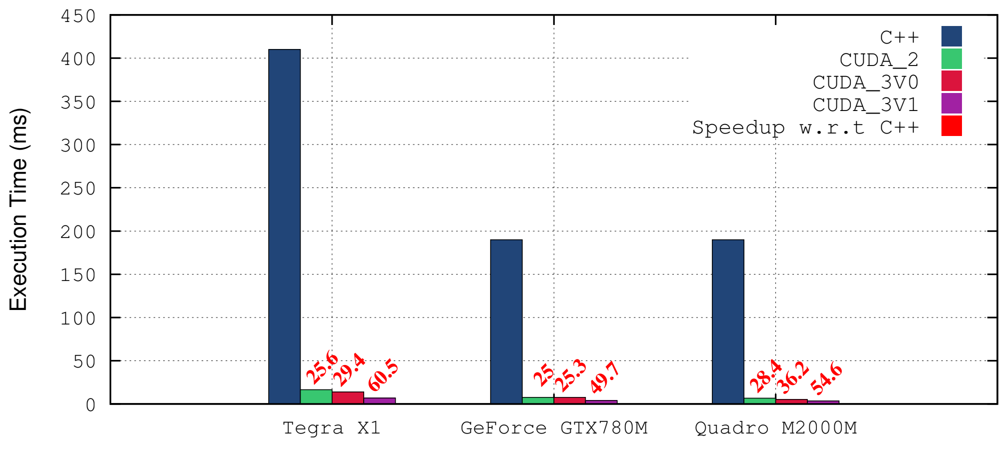


Figure 5.12: Execution Time of Second Version of Third `CDD CUDA` (CUDA_3V1) Kernel with Shared Memory on KITTI Dataset

5.6.4.2 Optimize Regular Access Pattern

To improve the previous kernel, we propose to change the pattern employed to get each pixel descriptor. The idea consists on using a descriptor of **18** elements. Each **9** elements are obtained from horizontal and vertical gradients respectively by using a patch of 3×3 size around the considered pixel. This optimization gives more regular access pattern which allows also loop unrolling since a loop can be used to load the data in this case which is not possible in the previous kernel. In this version (`CUDA_3V1`), we keep the same configuration (except this optimization) as in the previous kernel `CUDA_3V0` such as block size, thread/pixel granularity ...

Figure 5.12 shows the obtained execution time on KITTI dataset of this `CUDA` kernel in the three architectures: Tegra X1, GeForce GTX780M and Quadro M2000M . We observe better performance with lower execution time in all architectures. If compare `CUDA_3V0` to `CUDA_3V1`, we obtain a speedup of 2. With respect to the scalar version (`C++`), the speedup is more than 50 in all architectures.

Figure 5.13 depicts the obtained results on Jetson K1 GPU. We notice better performance with `CUDA_3V1` compared with `CUDA_3V0`. We managed to double the speedup by modifying the memory access pattern to the shared memory. These results show the importance of working on regular memory accesses when fast memories such as caches or shared memory are employed to accelerate the algorithm.

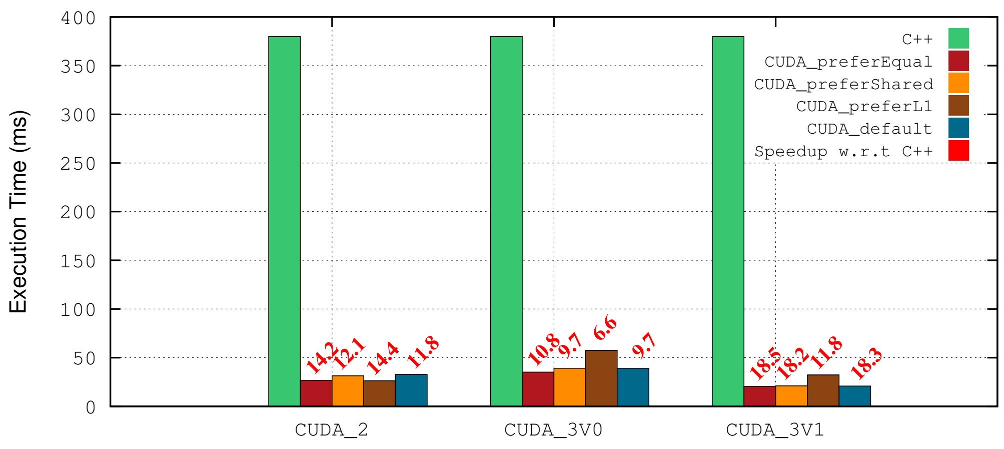


Figure 5.13: Execution Time of Second Version of Third `CDD CUDA` (CUDA_3V1) Kernel with Shared Memory on KITTI Dataset in Jetson K1

5.6.4.3 Maximize Number of Warps

In this version (`CUDA_3V2`), we propose to reduce thread/pixel granularity to 2 pixels per thread. The GPUs employed in these experiments all are configured to have a maximum of *48KB* of shared memory per SM. Because each kernel in `CUDA_3V1` version is configured to use *4KB* of shared memory for each block, each SM is limited to simultaneously executing *12 bloks(48 warps)* instead of *16 bloks(64 warps)*. By reducing the amount of shared memory per block by *2*, we get *maximum number of warps (64)* executed in each SM. We keep the same features as in kernel `CUDA_3V1` except the proposed optimization.

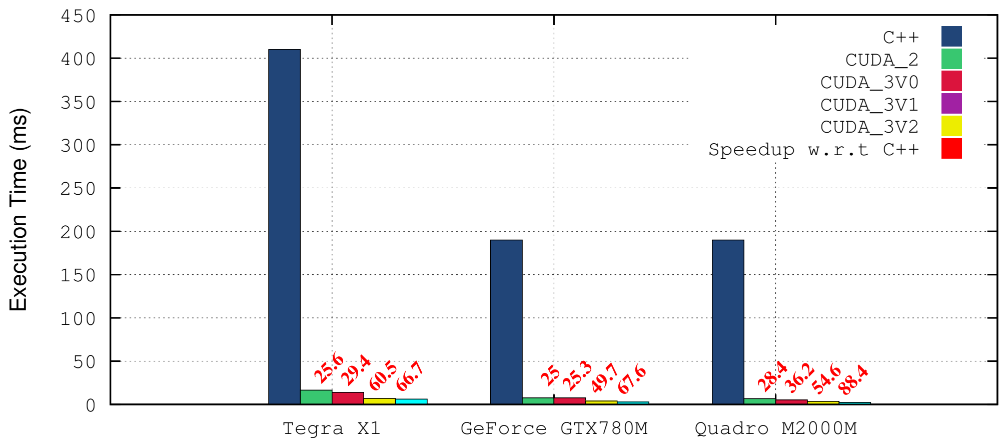


Figure 5.14: Execution Time of Third Version of Third `CDD` `CUDA` (CUDA_3V2) Kernel with Shared Memory on KITTI Dataset

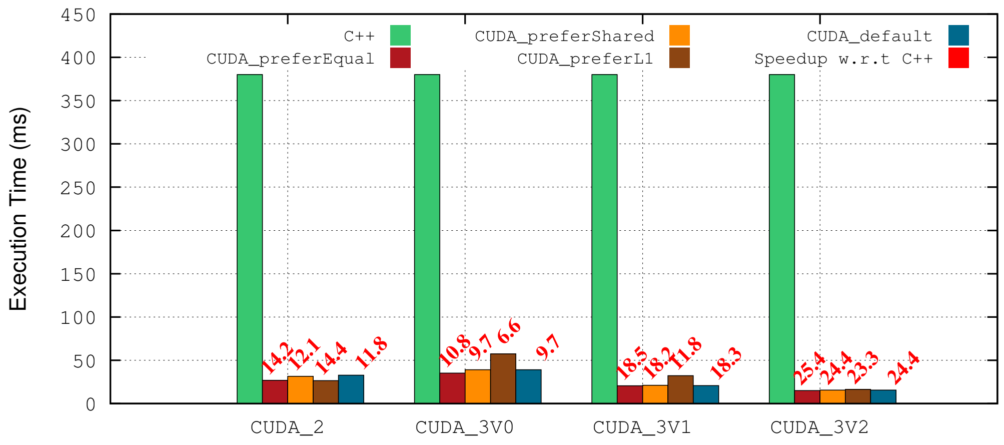


Figure 5.15: Execution Time of Third Version of Third `CDD` `CUDA` (CUDA_3V2) Kernel with Shared Memory on KITTI Dataset in Jetson K1

Figure 5.14 shows the obtained execution time with `CUDA_3V2` kernel of `CDD` function on KITTI dataset in all architectures except in Jetson K1 (see the next paragraph). We managed to accelerate the `CDD` function by maximizing the number of launched warps per SM. This is an important feature to achieve better performance. We notice an important speedup which is more than **88** for instance in Quadro M2000M.

Figure 5.15 depicts the obtained results on Jetson K1. We notice better performance in this version (`CUDA_3V2`). The results illustrates also small difference between the

Table 5.8: CSM and CDD Specifications

Feature	CDDL	CDDR	CSM
N[1]	$H \times W$	$H \times W$	$(H \times W) \div 25$
disp_max_range	40	40	250
left_disp	yes(1)	no(0)	yes(1)
right_right	no(0)	yes(1)	yes(1)
M[2]	1	1	4
total_num_SAD	$H \times W \times 40 \times (1+0) \times 1$	$H \times W \times 40 \times (0+1) \times 1$	$((H \times W) \div 25) \times 250 \times (1+1) \times 4$

[1] Number of treaded pixels for which the disparity is computed.

[2] Number of neighboring pixels used to compute the cost matching function (SAD).

different configuration of shared/L1 memories compared to the previous versions such as `CUDA_3V0`.

5.7 OpenACC and CUDA Optimizations of `CSM`

In this section, we present and discuss the optimizations applied to accelerate the `CSM` function. The latter computes the disparity of a set of support points candidates. The SAD cost function is employed. To find the disparity of each support point we compute the SAD between the 4 corners surrounding the considered support points. The corners are taken from a patch of size 5×5 pixels. This results in irregular memory accesses. Also, at this level we do not have any prior data for the disparity range. Hence, for each pixel, we look for its corresponding pixel in a maximum range of 255. For more details of this function presently presented in chapter 4, refer to section 4.6.1.2.

Before optimizing `CSM` function, we analyzed it in terms of arithmetic intensity to compare it to `CDDL` and `CDDR` functions. Table 5.8 gives some data required to compute the total number of `SAD` computed in each function. While `CDDL` and `CDDR` functions compute the disparity for the whole image ($H \times W$), `CSM` function treats only some support points candidates. These points are selected in such a way that each points are separated horizontally and vertically by 5 pixels. Hence, we only compute disparity for $(H \times W) \div 25$ pixels. The maximum disparity range is smaller in `CDDL` and `CDDR` functions, since at this level, we already got some prior data from support points to limit the disparity boundaries. However, with `CSM` function, there is no prior data, hence we take the maximum disparity range ($[0:250]$). In KITTI dataset, we get a an average

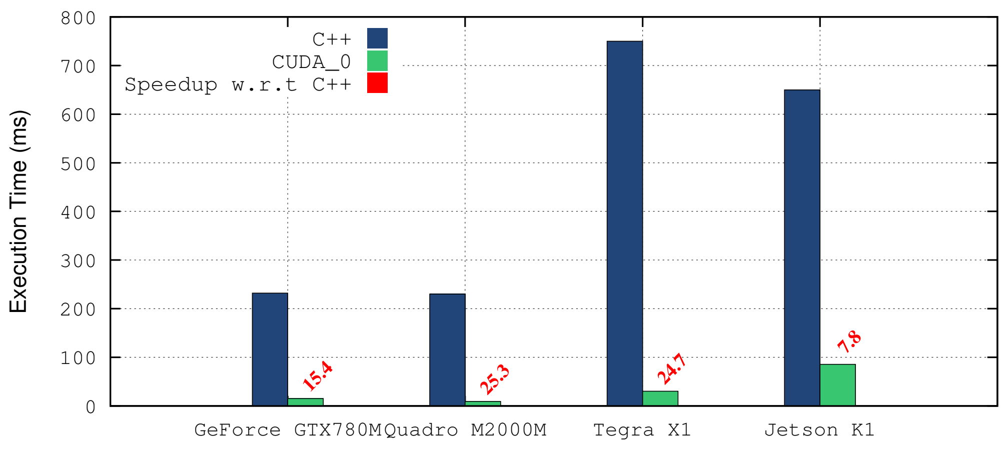


Figure 5.16: Execution Time of CSM CUDA First Kernel on KITTI(1242x375) with CUDA

disparity range of 40, which can varies in other datasets. Finally, in CDDL and CDDR function, we compute only one disparity, left and right respectively. In CSM function, we rather compute both disparities and we perform some post processing to check for consistency between the two disparities. From table 5.8, CSM function computes twice number of SAD compared to CDDL and CDR function. Hence, we expect this function to be slower and requires different optimization approaches.

5.7.1 First Naive Implementation

As first implementation, we selected to use directly Sobel images as input data instead of descriptor images. Since choice is based on the previous performance obtained on CDD function. As discussed previously (section 5.6.3). This optimization allows to minimize accesses to global memory which a crucial point with a high priority. The CUDA block size is set to 32×4. Each thread computes the forward (left) and backward (right) disparity of each support point.

5.7.1.1 CUDA Kernel

We then implemented the CSM function by using Sobel images as input data. Figure 5.16 depicts the execution time of this kernel (CUDA_0) compared to the scalar version (C++). We clearly notice that CUDA kernel is faster than the scalar version in all GPUs. The speedup in Maxwell's GPUs (Tegra X1, Quadro M2000M) is around **25**. In GeForce

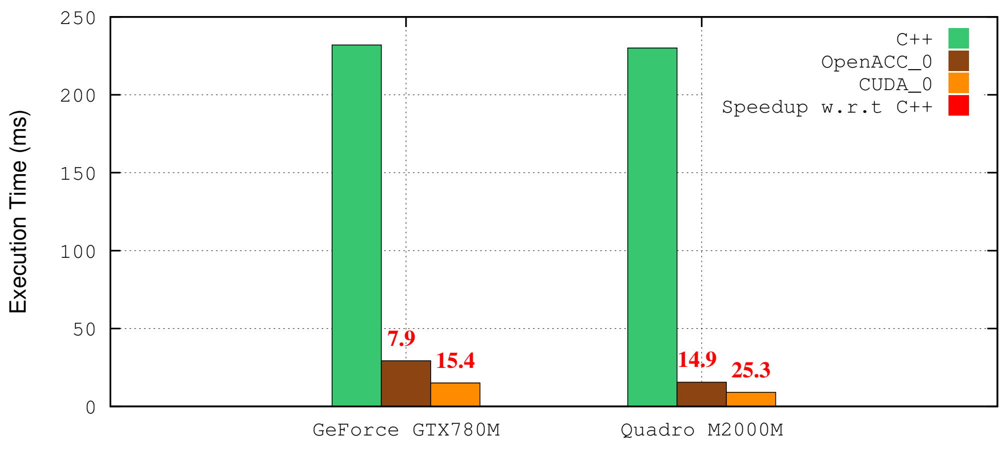


Figure 5.17: Execution Time of `CSM` `OpenACC` First Kernel on KITTI(1242x375)

GTX GPU, a speedup of around **16** is achieved while in Jetson K1 the speedup obtained is only **7.6**.

5.7.1.2 OpenACC Kernel

We implemented the same kernel with `OpenACC` by setting the same `CUDA` block size by using `gang` and `vector` clauses. The results are depicted in Figure 5.17. Despite the fact that we obtained an important speedup compared to the scalar version (`C++`), this speedup is lower compared to `CUDA`'s kernel speedup. To accelerate `OpenACC` kernel, we propose two optimizations discussed in the next sections

5.7.2 First Optimization with OpenACC: avoid warp divergence

The first optimization consist on minimizing the warp divergence. Warp divergence occurs with control conditions and branches such as with `if`$\rightarrow$`then`$\rightarrow$`else` statements. When this happens, threads within the same warp may not follow the same branch, hence, they wont do the same instructions. This yields to the execution of the different branches **sequentially**. To avoid this divergence, we propose to perform a test on the thread index to make sure that all threads within the same warp follow the same direction. The idea consist on replacing each test applied to the thread index with %32. In other words, let's take a thread at location (u,v). If in the original code we have a test `if u<2 and v<2 then ...`, then we replace this statement with the following `if u%32<2 and v%32<2 then ...`

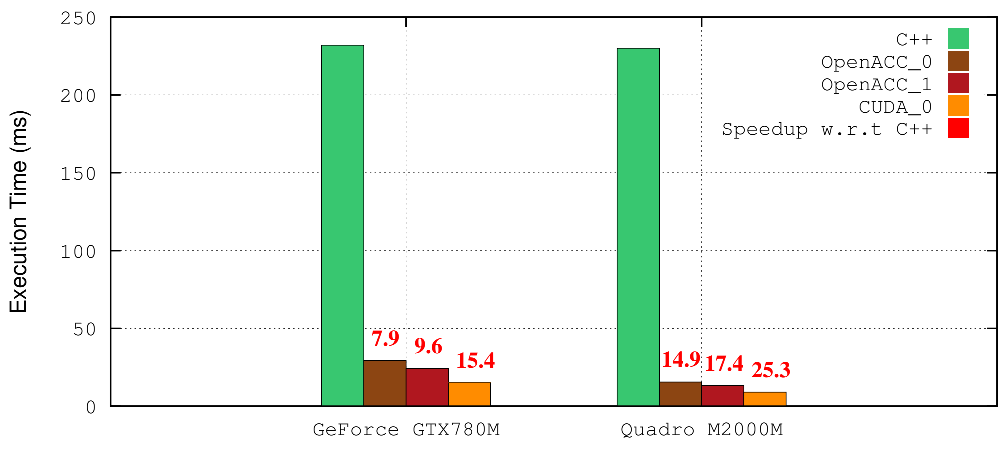


Figure 5.18: Execution Time of CSM OpenACC Second Kernel on KITTI(1242x375)

Figure 5.18 depicts the obtained results with this optimization (OpenACC_1). This optimization allows to decrease the execution time and consequently increase the speedup. However, we notice that we did not yet reach the same performance as with CUDA_0 kernel. We propose the next optimization to accelerate more the OpenACC kernel of CSM function.

5.7.3 Second Optimization with OpenACC: Explicit Loop Unrolling

When working with directive-based approaches, it is evident to understand and debug when the obtained performances are far from the expected ones. With CSM function, we faced a problem of understanding why OpenACC kernel is slower than CUDA one with the same parallelization approach. The approach we followed to figure out the issue is a progressive one. The idea consist on comparing the performance between CUDA_0 and OpenACC_2 kernels by omitting and putting back parts of the code step by step. Each time we remove part of the CSM code, we compare the execution time of the two kernels. By doing so, we manages to find where the PGI compiler fails to optimize the code.

Actually, we found that PGI compiler fails to perform `loop unrolling` on the `for` loop which computes the SAD between the four corners. The solution then is to do **`loop unrolling` explicitly** by hand. Figure 5.19 shows the obtained results. We notice that with this optimization (OpenACC_2), we managed to improve the obtained speedup and even get better performance compared to CUDA_0 kernel on both architectures.

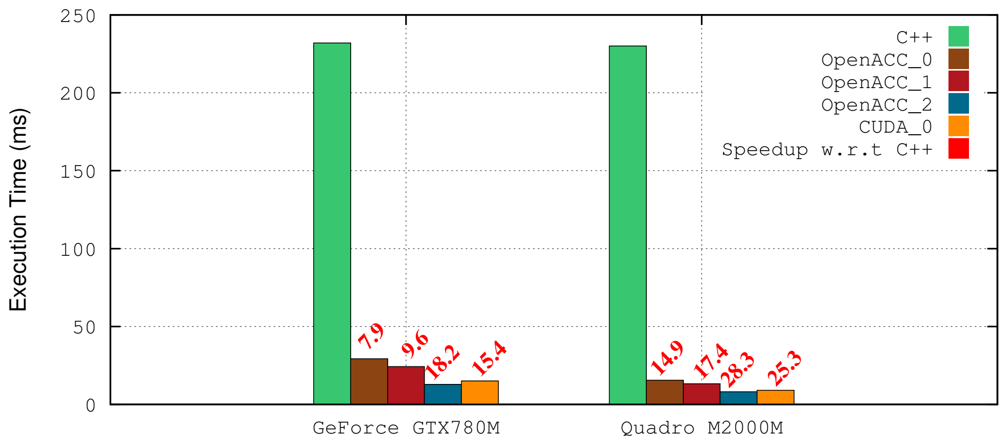


Figure 5.19: Execution Time of `CSM OpenACC` Third Kernel on KITTI(1242x375)

5.8 Evaluation : CUDA vs OpenACC

5.8.1 Productivity

`CUDA` is the most widely used programming platforms for high performance scientific computing on `GPUs`. However, it is a low-level and explicit programming language which cannot be handled easily by non-experts in the field. This results in relatively low programming productivity. Actually, porting existing CPU-based (scalar) code to `CUDA` is a whole process. First, code is profiled and analyzed to determine the most time consuming regions referred to *bottlenecks*. These regions are then *rewritten* into `CUDA` kernels, with less or more significant changes in the original code. Then, the programmer has to analyze the code to optimize it according to the available hardware computing capabilities and software specifications. In the other hand, `OpenACC` also requires bottlenecks identification. However, compared to `CUDA`, *no rewriting* is required anymore, in other words, the original CPU-based code can be reused. The programmer needs just to add some annotations in the regions to parallelize through `OpenACC` directives to give hints to the compiler. This concept reduces significantly programming efforts when porting large applications.

Porting Cycle Programmers should take an incremental approach to accelerate applications using `OpenACC` to ensure correctness. This approach starts by assessing application performance, then using `OpenACC` to parallelize important loops in the code, after that, optimizing data locality to remove unnecessary data movements between the host and accelerator, and finally optimizing loops within the code to maximize performance on a given architecture. There are two important things to take into consideration. First, during the process of optimization, application performance may actually slow down. Developers should not become frustrated if their initial efforts result in a loss of performance. This is generally the result of implicit data movement between the host and accelerator, which will be optimized as a part of the porting cycle. Second, it is **critical** that developers check the program results for correctness after each change. Frequent correctness checks will save a lot of debugging effort, since errors can be found and fixed immediately. Some developers may find it beneficial to use a source version control tool such as `Git` to track the code after each successful change so that any breaking changes can be quickly thrown away and the code returned to a known good state.

5.8.2 Portability

`OpenACC` has been designed in such a way that with a single programming model, programmers can write a single parallel program which can run with high performance while targeting a wide range of architectures used today: multi-core CPUs with `SIMD` instructions, many-core processors such as Kalary `MPPA-256`, massively parallel `GPUs`, and heterogeneous design where a host CPU is coupled to a co-processor or GPU accelerator. `PGI` has developed `OpenACC` compilers targeting NVIDIA Tesla and AMD Radeon GPUs, multicore and manycore CPUs. Several works in the literature provide benchmarks and applications which show performance portability of `OpenACC` across different systems [119], [120], [121].

`OpenACC` targets portability between `CPU-based` platforms such as multicore and manycore `CPUs`, and GPU-based systems. This allows developers to develop codes at once, with no need to include compile-time conditionals (`ifdefs`) or any special modules for each target which reduces significantly development and maintenance cost. If GPU-accelerated systems are targeted, programmers can develop parallel `OpenACC` code, test on multicore laptop or workstation without a GPU. This simplification separates algorithm development from GPU performance tuning. Also, usually, developers find debugging easier on the host than with both host and GPU code.

To summarize, while CUDA is only used on NVIDIA GPUs, in contrast, OpenACC has been designed to be portable on different architectures (CPU, GPU) from multiple vendors.

5.8.3 Performance

OpenACC portability across different architectures results in limiting the use of vendor-specific architectural specifications. OpenACC does not support software-addressable on-chip memory referred to *shared memory* in CUDA. OpenACC provides cache directive to fetch data from shared memory. However, it is limited to *read-only* data due to the lack of synchronization related to shared memory.

5.8.3.1 OpenACC Benefits and Limitations

The simplicity and portability that OpenACC programming model sometimes comes at a cost to performance. The OpenACC abstract accelerator model cannot represent architectural specifications of any device without making the language less portable. There will always be some optimizations that are possible in a lower-level programming model, such as CUDA or OpenCL, that cannot be represented at a high level. For instance, although OpenACC has the cache directive, shared memory on NVIDIA GPUs is more easily represented using CUDA. It is up to the developers to determine the cost and benefit of selectively using a lower level programming language for performance critical sections of code. In cases where performance is too *critical* to take a high-level approach, it is still possible to use OpenACC for much of the application.

5.9 Discussion

In this chapter, we investigated and evaluated two different parallel approaches on GPUs. As a directive-based approach, we selected OpenACC mainly for its portability on different architectures (CPUs, GPUs). As a second approach, we worked on CUDA. The latter is widely employed on programming GPUs (NVIDIA GPUs). The obtained results allow us to evaluate both approaches from different aspects; performance, portability and programming effort. To conclude, we can say that actually no approach is perfect for a particular application. In other words, one approach may be easy and fast to use such as OpenACC but may not give easily the expected performances as we have seen with CSM function 5.7. In the other hand, CUDA gives good performances, but the process of optimization is a progressive one which implies testing different optimizations before getting the required performance.

6

Kernel-System-Level Optimizations

Accelerating image processing algorithms is not a trivial task. Actually, optimization efforts fall into two axes: *kernel-level* and *system-level*. Kernel-level optimizations aim to optimize parts of the algorithm such as functions. In other words, these optimizations are applied only at low level referred as *kernel-level*, not on the complete algorithm. System-level optimizations in the other hand aim to optimize the algorithm as a whole system such as optimizing the inter-communications, data transfers and memory bandwidth.

In the previous chapters, we presented different kernel-level optimizations while targeting CPU-based systems as well GPUs. In chapter 4, we discussed shared memory parallelization and vectorization on CPU-based systems with various techniques (`OpenMP, SIMD, OpenACC, NT2`). In chapter 5, the algorithm is parallelized on GPUs with different approaches (`CUDA, OpenACC`).

In this chapter, we investigate one system-level technique proposed by Khronos group. It is a graph-based framework referred as `OpenVX`. It is a cross platform `API` standard where image processing applications are presented with a set of basic functions interacting with some data dependencies. The literature states several works on optimizing `OpenVX`. The majority of the proposed approaches describe techniques to accelerate the execution of graph-based image processing applications with `OpenVX` on many-core accelerators [122]. However, to the best of our knowledge, no work has investigated `OpenVX` optimizations presented in the first release of `OpenVX` [55].

In the first part of this chapter, we study the impact of `OpenVX` on computer vision applications [18]. Three important `OpenVX` optimizations are investigated; *kernels* *merge*, *data Tiling* and *parallelization* via OpenMP. We also show the important impact of data access pattern and how OpenVX responds to each pattern access class.

In the second section of this chapter, we propose an approach to target both system-level and kernel-level optimizations on different hardware architectures [19]. Our approach consists in merging OpenVX framework and Numerical Template Toolbox NT2 Library. OpenVX gives a close attention to system-level optimizations and enables hardware vendors to implement customized accelerated image and vision algorithms. NT2 library (see section 2.3.4.2 from chapter 2) accelerates kernels on different architectures with a minimal cost of rewriting, based on generative programming model. We test our approach on different computer vision kernels employed in ADAS. We target different acceleration techniques such as vectorization and shared memory parallelization. We perform our experiments on x86 architecture as well as on NVIDIA Tegra X1 ARM cores. We manage to execute OpenVX hardware customized kernels in both architectures with **zero** rewriting cost thanks to NT2. Also, it is worth noting that we get the same performances on both architectures.

6.1 OpenVX

6.1.1 Background

OpenVX [55] is a cross-platform C-based API standard. It allows enabling hardware vendors to implement and optimize image processing and computer vision applications. It is strongly supported by many industrial actors such as NVIDIA. OpenVX framework is fully transparent to architectural details. Indeed, all details concerning hardware platform are hidden in the underlying Run-Time Environment (RTE). This features enables the portability of computer vision applications across different heterogeneous platforms. This approach allows the hardware vendors to delegate the performance tuning according to their requirements and hardware specifications.

OpenVX API uses a graph-based software architecture to enable efficient computation on heterogeneous computing platforms, including those with accelerators such as GPUs. OpenVX provides a set of primitives (kernels) which are commonly used in computer vision algorithms. It also provides a set of data objects that may be simple data structures as scalars, arrays, matrices and images. In addition, it supplies high level data objects as histograms, image pyramids and look-up tables.

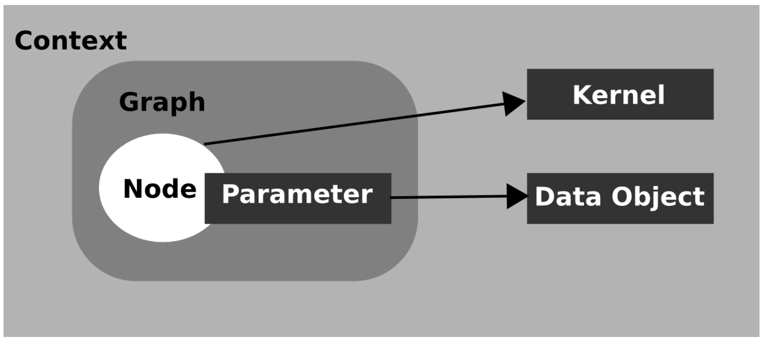


Figure 6.1: OpenVX Framework Objects.

6.1.2 Programming Model : Front-End

OpenVX programming model is inspired from the fact that most image processing algorithms can be structured as a set of basic functions. Indeed, OpenVX relies on graph-oriented execution model based on Directed Acyclic Graphs (DAG) to describe data flow and processing. OpenVX also defines the *vxu* utility library referred also as the *immediate mode*. It enables the programmer to use OpenVX predefined functions as a directly callable C function, **without** creating a graph. However, this mode do not benefit from the optimizations enabled by graphs. The vxu library is the simplest way to use OpenVX and as the first step in porting existing vision applications.
Figure 6.1 illustrates OpenVX framework objects and their relationships.

- OpenVX *context* is a container of all OpenVX objects. A graph is also created in reference to the context. Hence, before creating a graph and using any data object, a framework *context* is first created.

- OpenVX *graph* gathers nodes with their connections. The graph must be directed (only goes one-way) and acyclic (does not loop back). The order of nodes' execution inside a single graph is defined by the data-flow which is guaranteed.

- OpenVX *data objects* are used by an OpenVX program such as images. They are declared within the context. Data objects may be declared as *virtual*. Virtual data define only a dependency between adjacent kernel nodes, however, they are not associated to any memory area accessible through API functions

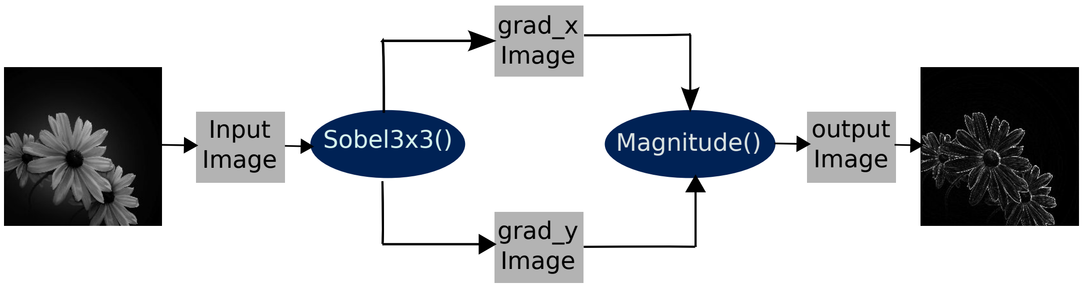


Figure 6.2: OpenVX Sample Graph Diagram of Sobel Filter.

- `OpenVX` *kernel* is the abstract representation of a computer vision function, such as a "Sobel Gradient". `OpenVX` standard provides a library of predefined vision kernels. It also supports customized user defined kernels with new features.
- `OpenVX` *node* is an instance of a kernel. The node encapsulates the kernel functionality. Every node that is created has to belong to a graph. It is important to understand the difference between a kernel and a node. By *kernel*, we mean the implementation of the processing function or algorithm. Where as, a *node* is an instance of the kernel within the graph.

```c
vx_status status = VX_SUCCESS;
vx_context context = vxCreateContext(); // create context
vx_graph graph = vxCreateGraph(context); // create graph
// create images
vx_image images[] = {
 vxCreateImage(graph, width, height, VX_DF_IMAGE_U8),
 vxCreateImage(graph, width, height, VX_DF_IMAGE_S16),
 vxCreateImage(graph, width, height, VX_DF_IMAGE_S16),
 vxCreateImage(graph, width, height, VX_DF_IMAGE_S16),
};
vxuFReadImage(context, "lenna.pgm", images[0]); // read input image
// construct graph nodes
vx_node nodes[] = {
 vxSobel3x3Node(graph, images[0], images[1], images[2]),
 vxMagnitudeNode(graph, images[1], images[2], images[3]),
};
status = vxVerifyGraph(graph); // verify graph
if (status == VX_SUCCESS)
   status = vxProcessGraph(graph); // execute graph
```

Listing 6.1: C Code Snippet of OpenVX Sobel Filter.

Listing 6.1 shows a typical `C` code of an `OpenVX` Sobel filter. Its corresponding diagram is depicted in Figure 6.2. The employed kernels are all predefined primitives available in `OpenVX` standard. The programming flow starts by creating a context for the `OpenVX` to manage references to all used objects (line 2, Listing 6.1). Based on this context, we construct a graph (line 3). At this level, we create all required data objects (lines 6 to 9). We read the input data (line 11). We notice here that we used the vxu function to read the input outside the graph, we could also use the vx function inside the graph. Then, we make connections between nodes to build the graph nodes (lines 13 $\rightarrow$ 15). Then, the graph is verified (line 17) to guarantee some mandatory properties such as:

- Input and output specifications (data direction, data type . . .) must be compliant to the node interface.
- Cycles are not allowed in the graph.
- It is not allowed to associate more than a single writer node to any data object.

Finally, the graph is processed by the `OpenVX` framework (line 19). At the end of the code execution, we destroy all created data objects and the graph, and finally we release the context.

6.1.3 System-Level Optimizations

The most important optimizations of `OpenVX` [55] are :

- **Graph Scheduling** consists in splitting the graph execution across the whole system which may be heterogeneous. Hence, we can execute a set of kernels in different processing units like CPU, GPU and DSP. The scheduling is performed according to the performance requirements of each kernel. These requirements refer to faster execution and lower power consumption.
- **Kernels Merge** replaces a subgraph which consists of different kernels with one single kernel for better memory locality and less kernel launch overhead.
- **Data Tiling** executes a subgraph at tile granularity rather than at image granularity. Tilable kernels are those where the output depends only on a subset of the input, not the entire input. Data is *automatically* broken into tiles. Tiling is

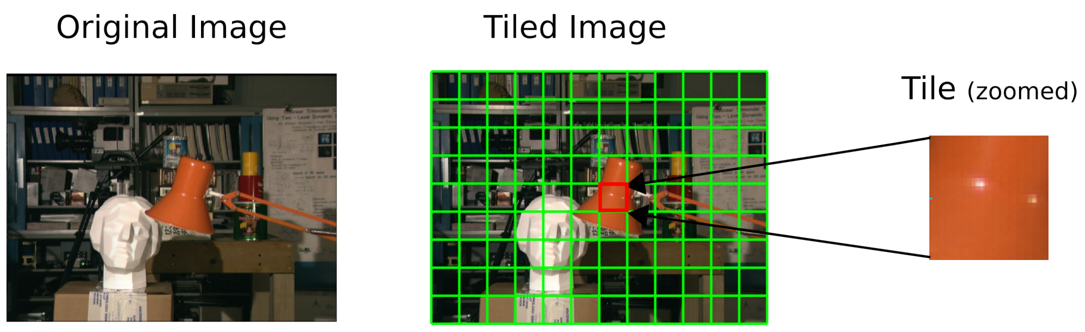


Figure 6.3: Tiling Principle in Image Processing

important and useful for several reasons. First, intermediate data tiles are stored entirely in on-chip local memory or caches. As a consequence, external memory usage and bandwidth are reduced. In addition, memory access coherence is better managed.

- **Memory Management** relies on virtual data to better use memory by reducing allocation overhead. Virtual data are not allocated or constructed, they are not guaranteed to actually reside in main memory. Basically, they just define a dependency between adjacent kernel nodes.

6.1.4 OpenVX User Kernel Tiling Extension

The idea behind tiling is related to the processors architecture in one hand, and to the image processing algorithms requirements in the other hand. Modern processors have different memory hierarchy ranging from small, fast and expensive memory to relatively large, slow and inexpensive memory. Image data is generally too large to fit in small and fast memory. So, the solution is to break the image into small blocks called tiles which fit into small and fast memory. This tiling optimizes memory access and allows parallel execution of different tiles. Figure 6.3 illustrates the principle of tiling in image processing. `OpenVX` proposes a user tiling `API` to better optimize the user nodes. To use the `OpenVX` tiling extension, the developer must provide the kernel function. There are two types of function to provide:

- **Flexible function** : it accepts irregular tile sizes and it may be called near the edge of the tile's parent image, so it has to take into consideration these issues.

- **Fast function** it is a specific case of the flexible function, it accepts only regular tiles size and it doesn't care about the edges and the neighborhood region of the parent image. It is only called with tile size and pixel alignment that are multiple of the user-specified "tile block size"

If the developer gives only the fast function, output near to the edges will not be computed. If the flexible function is given, then output is computed in the whole images including the tiles edges and the image neighborhood region. Even though irregular tile sizes are possible with flexible function, OpenVX attempts always to use regular sizes and alignments where possible. If both functions are provides then fast function is called as much as possible with its restriction and flexible function is called elsewhere where it is necessary (edges).

Tiling is managed by the `OpenVX` graph manager. However, the data access pattern is not taken into account. In an unoptimized `OpenVX`, images are by default allocated in L3 memory. Meanwhile, with tiling, the `OpenVX` run-time manager allocates the tiles in L1 buffers.

The performed experiments as well as the obtained results on investigating `OpenVX` optimizations are presented on section 6.3

6.2 Boost.SIMD

For the second part of this chapter, we selected to use one important optimization of `NT2` referred as `boost.SIMD` [123]. It is a `C++` template library which provides a simple approach to benefit from the `SIMD` hardware based on a `C++` programming model. `Boost.SIMD` allows vectorization on different hardware in a more portable way with different technologies of `SIMD` such as `Altivec`, `SSE` or `AVX` while providing a generic way to extend the set of supported functions and hardwares. Recently, a support for ARM-based architecture is available which allows then to generate a `NEON`-like code.

6.2.1 Why boost.SIMD

When `SIMD` units were introduced, processors can exploit the data parallelism available in applications by executing a single instruction on multiple data simultaneously in one single register.

In 2009, ARM designers introduced a new feature in ARM-based embedded processors by providing an enhanced floating point capabilities together with `NEON SIMD` technology into Cortex-A series microprocessors [124]. These processors were initially designed for smartphone market. These new features allow more effective data movement with a smaller instruction stream. The fact of adding improved floating point and `SIMD` instructions comes from the rapidly increasing demands of modern multimedia applications [125]. The current range of Cortex-A processors are capable of performing single precision floating point operations with SIMD. Recently, 64-bit ARM v8-A series support double precision operations such as in Tegra X1 ARM CPU [10].

With a constantly increasing need for performance in applications, today's processors' designers provide richer `SIMD` instructions set while using larger and larger `SIMD` registers. For instance, Intel introduced the `AVX` extension in 2011 with registers of size 256-bits to enhance the x86 instruction set. However, programming applications with `SIMD` instructions on the current targets is not a trivial task. Actually, to take advantage of the `SIMD` extension, programmers usually use low-level intrinsics and need to write a new code of the initial algorithm in adequacy with the architecture specific details. In addition, all these efforts are repeated for every different `SIMD` extension and on every architecture. This yields to a low portable design and time consuming programming approach.

To cope with the aforementioned issues–portability, time-consuming development–different solutions have been proposed in the literature. First, automatic vectorization, or autovectorization [126], [127], [128], performed by compilers. A code is analyzed to identify profitable instructions for conversion from scalar to vector, then the code is transformed appropriately. However, compilers fail when a code is not presenting a clear vectorizable pattern, for instance, with complex data dependencies and non-contiguous memory accesses. Another solution based on compilers also consists on using *explicitly* directives to give hints to the compiler where possible vectorization may be performed. As an example of this approach, we may cite the `ICC` and `GCC` `#pragma simd` directives. It is possible to autovectorize via some libraries such as `Intel MKL` [129]. These libraries provides a set of linear algebra and/or signal processing kernels optimized for a specif architecture. Hence, for some applications, we may not find the required routines to autovectorize the code.

6.2.2 What is boost.SIMD

`Boost.SIMD` is high-level `C++` library designed to program `SIMD` architectures. It has been designed as an Embedded Domain Specific Language (`EDSL`). `Boost.SIMD` uses a generic programming model to handle vectorization in a portable way. This portability is guaranteed through an abstraction of `SIMD` registers. This abstraction is handled through `pack` class. For a given type `T` and a given static integral value `X` (`X` being a power of 2), a `pack` encapsulates the best type able to store a sequence of `X` elements of type `T`. For arbitrary `T` and `X`, this is simply `std::array<T,X>`, but when `T` and `X` matches the type and width of a `SIMD` register, the architecture-specific type used to represent this register is used instead. The class `pack`, supports an important number of high-level functions ranging from `C++` operators to arithmetic and reduction functions.

In the following section, we show the results of investigating some `OpenVX` optimizations. In section 6.4, we show the experiments and the obtained results on integrating kernels on `OpenVX` previously accelerated with `boost.SIMD`.

6.3 System-level Optimizations

It is compulsory to deal with pattern access when we work on stencils and image processing optimizations. Each pattern type implies different memory management, optimization and parallelization approaches. `OpenVX` standard does not provide anyway to specify the supported patterns' types and how it manages the run-time optimizations for each type. Hence, to assess the real impact of `OpenVX` and its optimizations, we have employed different image processing algorithms with various data access patterns (Figure 6.4).

We use for our experiments kernels from our previous work on a stereo vision vehicles detection algorithm based on disparity map segmentation and objects classification [17]. The selected kernels and their corresponding data access pattern (Figure 6.4) are as follows:

- **Point operator** the value of each output point or pixel is computed from its corresponding input point like threshold kernels. We have selected the sky-subtraction kernel [17]. This kernel removes the sky from the disparity map.

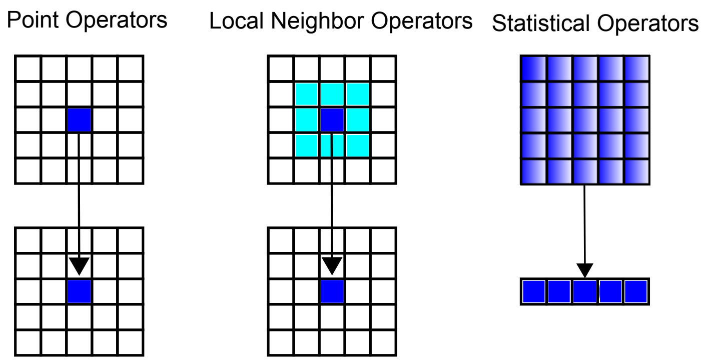


Figure 6.4: Some Classes of Data Access Pattern.

- **Local neighbor operator** the value of each output point depends on an input tile like morphological corrections. Sobel edge detector has been selected for this category.
- **Statistical operator** the value of each output point is computed with a statistical function like in histograms. For this category, we have used the V-disparity map [17]. It is a kind of histogram constructed from the disparity map to detect the road.

The first experiment consists in comparing the execution time of `OpenVX` kernels with `C++` equivalent programs. Then, we test kernel merge optimization to understand its real contribution on execution time,i.e. we compare the time performance of two kernels before and after their merging. We then investigate data tiling. Finally, we parallelize `OpenVX` kernels via `OpenMP` with and without `OpenVX` tiling.
We have done our experiments on a standard PC with an Intel CPU (i5) of 1.8 GHz. The operating system is Ubuntu 14.04. We installed the `OpenVX 1.1` sample implementation got from Khronos group [130].

6.3.1 Obtained Results

6.3.1.1 First Experience : OpenVX vs C++

In this section, we discuss whether the `OpenVX` framework and its graph management add an overhead compared to `C++` language. To do so, we implemented the three aforementioned kernels in `C++` and in `OpenVX`. For `C++`, we used `OpenCV API` to read

(a) Lenna (512x512) **(b)** Flower (1024x1024) **(c)** Wall (2048x2048)

Figure 6.5: Employed Images for Sobel Filter

input images and display outputs. For `OpenVX`, we have used the default template provided in the different samples within `OpenVX`.

To add a new kernel to `OpenVX` framework, we followed the steps mentioned in the file `NEWKERNEL` found in `OpenVX` home directory after installation. As a whole, there 10 major steps with a set of sub-steps on each one. Each step aim to update particular files in `OpenVX` to add some information concerning the new kernel. The steps can be summarized as follow:

1. Add kernel node and describe it in details such specifying data transfer direction (in, out) of all the parameters.
2. Add node prototype.
3. Add any external types needed.
4. Create a new file and write new kernel description. Describe in details all the specification of the input(s) and the outputs (s) such as the data types and their corresponding transfer direction.
5. Add vxu function if desired.
6. Write the `C` code of the kernel in a new file.
7. Compile the `OpenVX` directory to updates all the files and check for possible errors.
8. To test the new kernel, either modify or add new unit test in the provides samples.

By following the aforementioned steps, we added our kernels to `OpenVX` framework by wrapping the same code of the scalar version and by following the template provided in OpenVX. As first results, we focus on one kernel, Sobel filter, the results of the other kernels are discussed later one. Figure 6.5 depicts the employed images on Sobel filter and their corresponding resolutions. For sky-removal and V-disparity kernels, we used KITTI [101] dataset of 1242×375 pixels.

```c
 1 for (y = 1; y < h-1 && (status == VX_SUCCESS); y++) {
 2   for (x = 1; x < w-1; x++) {
 3       vx_uint8 *in_00 = vxFormatImagePatchAddress2d(src_base, x-1, y-1, &src_addr);
 4       vx_uint8 *in_01 = vxFormatImagePatchAddress2d(src_base, x, y-1, &src_addr);
 5       vx_uint8 *in_02 = vxFormatImagePatchAddress2d(src_base, x+1, y-1, &src_addr);
 6       vx_uint8 *in_10 = vxFormatImagePatchAddress2d(src_base, x-1, y, &src_addr);
 7       vx_uint8 *in_12 = vxFormatImagePatchAddress2d(src_base, x+1, y, &src_addr);
 8       vx_uint8 *in_20 = vxFormatImagePatchAddress2d(src_base, x-1, y+1, &src_addr);
 9       vx_uint8 *in_21 = vxFormatImagePatchAddress2d(src_base, x, y+1, &src_addr);
10       vx_uint8 *in_22 = vxFormatImagePatchAddress2d(src_base, x+1, y+1, &src_addr);
11       // compute horizontal and vertical gradients
12         grad_x = (*in_02) + 2*(*in_12) + (*in_22) - (*in_00) - 2*(*in_10) - (*in_20);
13         grad_y = (*in_20) + 2*(*in_21) + (*in_22) - (*in_00) - 2*(*in_01) - (*in_02);
14         vx_uint8 *dst = vxFormatImagePatchAddress2d(dst_base, x, y, &dst_addr);
15         // compute the sum of gradients
16         sum = sqrt(pow(grad_x, 2) + pow(grad_y, 2));
17         // normalize the sum and store the result
18         *dst = (sum) > 255 ? 255:(vx_uint8)(sum);
19   }
20 }
```

Listing 6.2: C Code Snippet of our OpenVX Sobel Filter.

Listing 6.2 illustrates part of Sobel kernel code that we wrapped within OpenVX by following the provided template in the samples. The kernel takes a gray scale image (8-bits) as an input. Some initializations are performed before such as extracting the rectangular patch required from the input and the output images through the function `vxAccessImagePatch()`. In our case the whole images are employed. We start by getting the required non zero neighboring pixels (lines 3 $\rightarrow$ 10). Then, we compute the horizontal and vertical gradients of each pixel (lines 12, 13). The gradients are stored in temporary `short` (16-bits) variables. After that we calculate the sum (line 16). Finally, we normalize the sum to the gray scale range [0:255] (lines 17, 18) and store the final result on the corresponding output pixel.

Figure 6.6 shows the obtained execution time on Sobel filter on different image resolutions. The first column (C++) presents the time obtained with C++ and the second column (OpenVX) gives the results with OpenVX by following the default programming

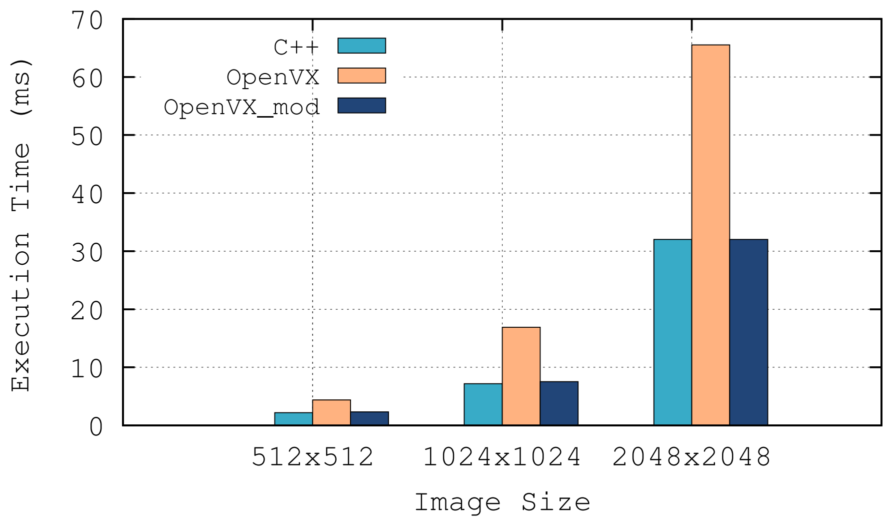


Figure 6.6: Performance Comparison Between C++ Language and OpenVX Framework on Sobel Edge Detector.

model of `OpenVX` samples. We clearly notice that `OpenVX` performs worse than C++ with an average factor of almost **2**.

By analyzing the code of the `OpenVX` provided samples, we noticed that the reason behind this difference in execution time is related to the way `OpenVX` accesses image pixels via the function *vxFormatImagePatchAddress2d()*. The latter extracts a patch with the corresponding horizontal and vertical offsets. Hence, each time a pixel is required, its address is computed with this function. To get rid of this difference, we modified the programming model of our kernels in `OpenVX` to avoid computing pixels' address each time. We did so by fetching a pointer to each image line once and then use it to get pixels according to their vertical position along the columns within the same line. This approach is referred as *linear addressing*. Listing 6.3 illustrates the principle (lines 2, 3, 4).

The results of this optimization are shown in the third column (OpenVX_mod) of Figure 6.6, *mod* stands for modified. We clearly notice that With this approach, we get the same performances compared to `C++`.

From this experiment we can conclude that `OpenVX` provided samples are not optimized in terms of execution time, hence, the developer needs to adapt his program for better performance.

```c
for (y = 1; y < h-1 && (status == VX_SUCCESS); y++) {
  // get image's lines address
  in_0 = vxFormatImagePatchAddress2d(src_base, 0, y-1, &src_addr);
  in_1 = vxFormatImagePatchAddress2d(src_base, 0, y, &src_addr);
  in_2 = vxFormatImagePatchAddress2d(src_base, 0, y+1, &src_addr);
  vx_uint8 *dst = vxFormatImagePatchAddress2d(dst_base, 0, y, &dst_addr);
  for (x = 1; x < src_addr.dim_x -1; x++) {
    // compute gradients
      grad_x = in_0[x+1] + 2*in_1[x+1] + in_2[x+1] - in_0[x-1] - 2*in_1[x-1] - in_2[x-1];
      grad_y = in_2[x-1] + 2*in_2[x] + in_2[x+1] - in_0[x-1] - 2*in_0[x] - in_0[x+1];
    // compute the sum of gradients
      vx_int32 sum = sqrt(pow(grad_x, 2) + pow(grad_y, 2));
      // normalize the sum and store the result
      dst[x] = (sum) > 255 ? 255:(vx_uint8)(sum);
  }
}
```

Listing 6.3: C Code Snippet of our Modified OpenVX Sobel Filter (OpenVX_mod).

6.3.1.2 Impact of Data Access Pattern

We implemented the V-disparity and the sky-subtraction kernels–Sobel filter has been implemented from the previous experiment– to evaluate the impact of data access patterns on `OpenVX` framework and to confirm the previous results. Figure 6.7 shows the obtained performances in Mega pixels per second (Mp/s) for each kernel. Sobel filter is applied to an image of size 1024×1024, kernels sky-subtraction and V-disparity both use KITTI images of size 1242×375.

Concerning the sky-subtraction kernel, the algorithm takes a bidirectional image (i.e. a disparity map) to update or not with respect to some tests applied on another input image (i.e. a saturation channel). Using a bidirectional data object avoids data copying when a pixel value is not updated in case the test is not verified. C++ with `OpenCV API` and `OpenVX` framework both allow using input/output images.
Figure 6.7 confirms that `OpenVX` gives the same execution time with different data access pattern. These are the expected results since we did not apply any optimization at this level, only scalar kernels are used. Data access pattern is crucial when optimizations are applied which may work with one pattern and not with others.

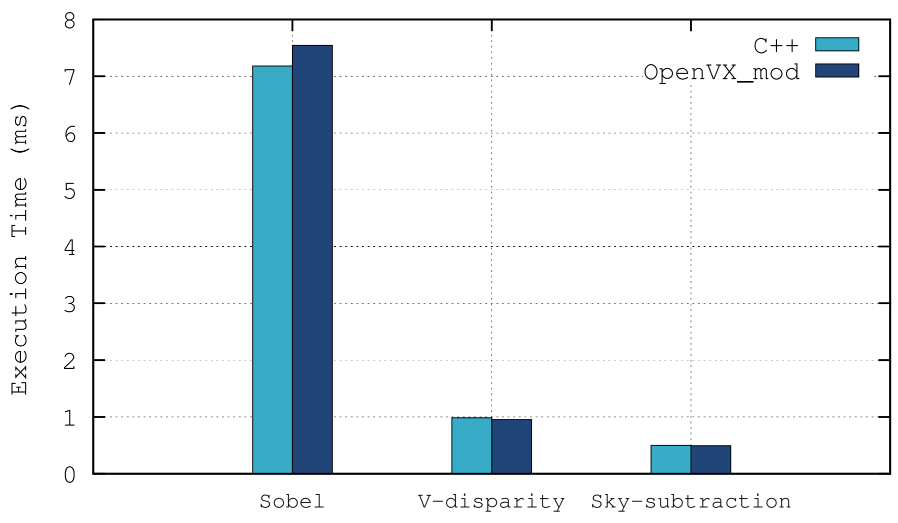


Figure 6.7: Data Access Pattern Impact on OpenVX Execution Time

Table 6.1: Execution Time of Kernel Merge Optimization on Sobel Filter.

Image Size	$\mathbf{t(G_x)}$(ms)	$\mathbf{t(G_y)}$(ms)	$\mathbf{t(G)}$(ms)	$\mathbf{t_s}$(ms)	$\mathbf{t_m}$(ms)
512x512	0.121	0.124	2.515	2,760	2.113
1024x1024	0.453	0.434	6.334	7,221	7.149
2048x2048	1.632	1.638	31.498	34,768	32.143

6.3.1.3 Kernels Merge

We selected Sobel filter to test kernels merge optimization. Sobel filter is a cascade of three elementary functions or kernels (Figure (6.2)): horizontal gradient (G_x), vertical gradient (G_y) and sum or magnitude of gradients ($G = \sqrt{G_x^2 + G_y^2}$). We developed these three kernels in `OpenVX` since as discussed previously (Listing 6.1), horizontal and vertical gradient kernels are merged in one kernel within `OpenVX` (`vxSobel3x3Node()`). Then we developed also the merged version of the three kernels in one which is not also available in `OpenVX` standard.

Table 6.1 presents the execution times of: the horizontal gradient t(G_x), the vertical gradient t(G_y), the sum of the gradients t(G), the total time of the three kernels (sep-

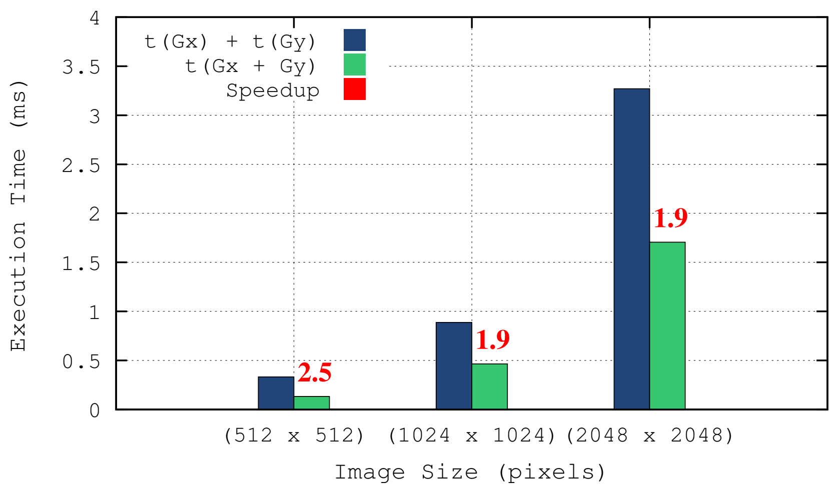


Figure 6.8: Kernels Merge Execution Time with OpenVX on Sobel Filter.

arable version) $t_s = (t(G_x) + t(G_y) + t(G))$, and the time of the whole Sobel filter merged in one kernel $t_m = t(G_x + G_y + G)$.

From Table 6.1, we notice that the execution time of the merged kernels t_m is around 5% less than the non merged ones (t_s). However, we expected the results to be better than this. Indeed, the performance of merging kernels is only obtained if the time required for memory accesses in the original algorithm is much more important than arithmetic operations' one. Then, if by merging the kernels we manage to reduce memory access by keeping the same arithmetic complexity, we then expect to accelerate our algorithm.

We perform the same experiment on the horizontal and vertical gradients. The latter choice is motivated by the ratio of *memory accesses* to *arithmetic operations* **R**. To compute each output pixel, R is equal to 7/7 on horizontal and vertical gradients corresponding to 6 reads to get the local non zero neighbor pixels, 5 additions, 2 multiplications and 1 write to store the output. However, if we merge these two kernels, we end up with R which is less than 1; with 8 reads of the required local neighbor points of both gradients, $7 \times 2 = 14$ arithmetic operations and 2 writes. Then, in this case, R is equal to 10/14. Based on this ratio, we implemented a merged kernel of only horizontal and vertical gradients on `OpenVX` and then compared the obtained performance with the separable kernels one. Figure 6.8 shows the execution time (ms) obtained on different image resolutions. We notice an average acceleration factor of **2** when we merge the two

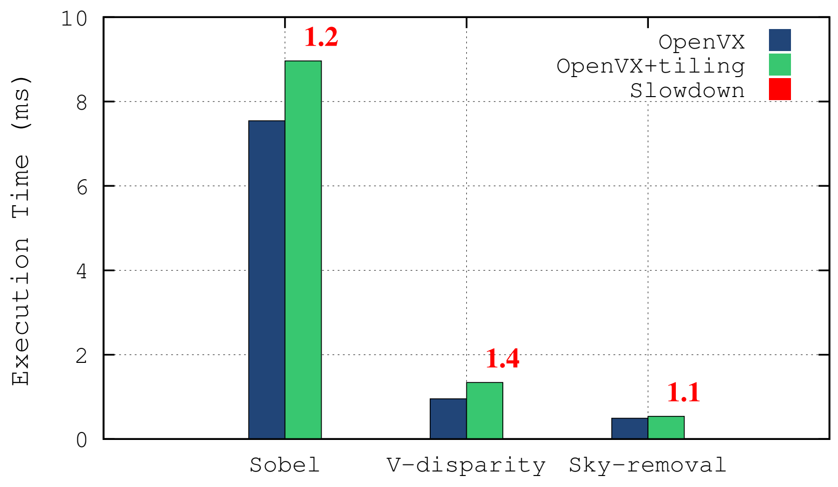


Figure 6.9: Obtained Execution Times with OpenVX Tiling Extension

kernels ($p(G_x+G_y)$) compared to the separable version ($p(G_x)+p(G_y)$. It is related to the reduction of memory accesses and `OpenVX` kernels launch overhead.

From this experiment we can conclude that `Kernels merge` optimization of `OpenVX` framework gives important results with respect to non merged kernels evaluated separately. However, we need first to analyze the kernels we want to merge to check if they respond to some features required to obtain a speedup. These features concern memory accesses and arithmetic operations. Indeed, this optimization does not aim to accelerate the processing such as with kernel-level optimization. It focuses on the memory accesses on the whole algorithm– kernels we want to merge. If by merging the kernels we manage to reduce the number of memory accesses while keeping the same arithmetic complexity, then we can expect to accelerate our kernels. This optimization can give important speedup mainly with memory bound kernels such local neighbor operators (filters).

6.3.1.4 Data Tiling

To test `OpenVX` Tiling , we implemented our three kernels and compared the obtained results with the previous ones without tiling. We modified the image access approach as explained in section 6.3.1.1. Figure 6.9 depicts the obtained execution time in (ms) as well as the obtained slowdown with tiling.

Concerning Sobel Filter (local neighbor operator) and sky-subtraction kernel (point operator), we notice a small difference of `OpenVX` performance compared to `OpenVX` with tiling. A slowdown of `1.2` is obtained with Sobel kernel and `1.1` with sky-removal kernel. When we worked with the V-disparity kernel, we faced the problem of different size between the input and the output, since the width of the V-disparity map is equal to the maximum disparity we can get, namely 255. Actually, `OpenVX` tiling extension considers by default the same size for input and output tiles. So, we executed our code by setting the right input and output tiles size by hand. This results in a slowdown of 1.4 in performance. Globally, the obtained performances with `OpenVX` tiling extension are far from the expected ones [55]. This slowdown can be related to the following features of `OpenVX` tiling extension :

- `OpenVX` tiling extension is implemented in software.
- `OpenVX` tiling configuration is fixed for all kernels and for all image sizes.
- `OpenVX` tiling is applied only at kernel granularity.
- There are not enough computations and memory accesses which may saturate the cache memory in the selected kernels.

Tiling in `OpenVX` is done in **software** by storing the tiled image in main memory. But, software tiling is not the best solution [131]. Indeed, storing untiled images in main memory have important advantages. **First**, the image storing and transferral mechanisms, such as DMA or I/O devices, are independent of the tiling scheme and can add some overhead. **Second**, different tilings mechanisms may be used on the same image. And the **third**, and possibly most important, advantage is that the *address translation* is transparent to user software. These advantages show that a conversion of the address take place when caching, and not in software. Hardware tiling improves performance. System software calls are made by the users to set up the cache for higher performance. For example, to allocate tiled memory a call to a custom allocation routine can set up the memory area to be tiled while cached.

Configuration of tile's size and numbers is performed by software and fixed in all cases. Here are the most important features :

1. The number of tiles by default is fixed to 64.

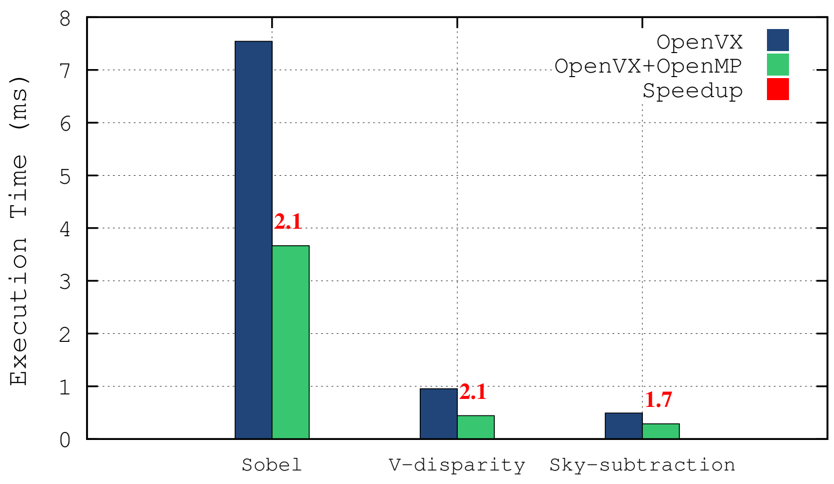


Figure 6.10: Obtained Execution Times on OpenVX with OpenMP

2. The image is tiled only vertically; the width of the tile by default is equal to the image's width.

These details can be found in the file `vx_interfaces.c` from `OpenVX` installation directory. We notice that Khronos developers selected to not tile the image along the width. This choice may be a good one with small images where the tile can fit a cache line, however, with images of high resolutions, this may not be a good solution.

Tiling in `OpenVX` is only applied at *kernel granularity*, in other words, it cannot be applied for instance at graph granularity. At graph granularity, tiling is applied at all images (input, temporary, output) and at all kernels. We expect the results to be better in this case.

The selected kernels for this experiment are not so time consuming. Usually, tiling is supposed to give better performance with kernels of high computing requirements and memory accesses.

6.3.1.5 OpenVX with OpenMP

In this section, we present the results obtained by testing `OpenMP` with `OpenVX`. We performed our tests on a computer with 2 physical cores of 2 threads each. Parallelization is applied at pixel granularity, i.e., each thread computes the output of one pixel in all kernels. Figure 6.15 shows the obtained results in (ms) for the different kernels.

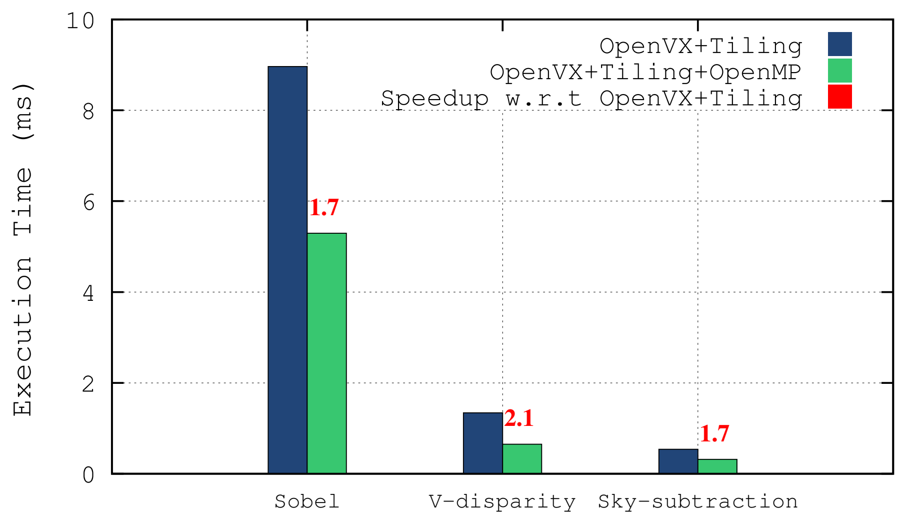


Figure 6.11: Accelerating OpenVX *Tiling Extension* with OpenMP

We notice an acceleration factor of almost 2 with all kernels. This experiment shows one important feature of `OpenVX` which allows users to develop customized kernels for a specific architecture. In this chapter, we selected `OpenMP` shared memory parallelization as an example on multi-core systems. In the next section (6.4), more advanced implementations will be discussed with a different approach.

6.3.1.6 OpenVX Data Tiling with OpenMP

We test `OpenVX` tiling with `OpenMP`. Two approaches are possible. The first one consists in parallelizing the inner loops by associating threads to pixels within the same tile. This technique corresponds to the work presented in section (6.3.1.5) with `OpenMP`. The second approach consists of parallelizing along the outer loops by associating tiles to threads. We followed the second approach.

The obtained results are depicted in Figure 6.11. We notice better results with an average factor of **2**. These results show that tiling extension can be optimized by parallelizing at tile level.

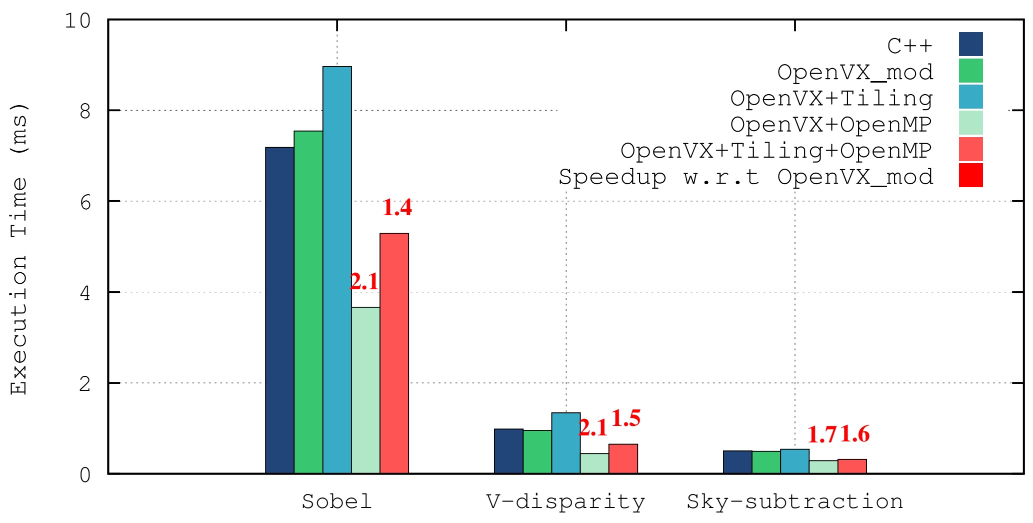


Figure 6.12: Summary of the Obtained Results with the Different Optimizations Tested on OpenVX

6.3.2 Discussion and Analysis of Obtained Results

Figure 6.12 summarizes the obtained results by using the different aforementioned optimizations tested on `OpenVX` with the three selected kernels. The speedup is also depicted with `OpenMP`. First, we notice that the employed approaches behave differently with respect to each data access pattern of the selected kernels. This confirms the importance of data access pattern in optimizing image processing applications. If we compare C++ and `OpenVX_mod` with the modification we applied on the image access, we notice almost no difference in execution time. However, tiling does not respond similarly with the three kernels. Indeed, `OpenVX` tiling extension does not manage data access pattern. Finally, `OpenMP` allows the reduction of execution time, hence, increasing the performance according to the time consumption of each kernel and the hardware specifications (number of cores).

6.4 kernel-System-level Optimizations

Before discussing the obtained results, we first describe the hardware employed as well as the kernels we use to test the approach we propose which consists on combining `OpenVX` and `boost.SIMD`. We perform our experiments on an x86 architecture and on the host

(ARM quad-core) of NVIDIA Tegra X1 platforms. This proposed approach could not be tested on NVIDIA Tegra X1 GPU since the `CUDA` generator of `NT2` is not yet available. Despite the fact that some optimizations of `NT2` on GPU have been implemented, the equivalent `CUDA-like` required operations and functions for our applications are not available. Hence, we selected to use `boost.SIMD`, which is part of `NT2`, for `CPU-based` vectorization while wrapping the code on `OpenVX`.

The architectural specifications of each platform are as follows:

- **x86**: 2-Core/4-Threads Intel 64-bits i5-3337U, CPU (1.8GHz), SIMD (SSE4.2/AVX), 256KB L2 cache, gcc 4.8.4, ubuntu1 14.04 LTS
- **Tegra X1 CPU**: 8-Core/8-Threads ARM 64-bits: 4xA53(1.7GHz), 4xA57(1.9GHz) SIMD(NEON), 2MB (A57), 512KB(A53) L2 cache, gcc 4.8.4, ubuntu1 14.04

We use for our experiments kernels from our previous work on a stereo vision vehicles detection algorithm based on disparity map segmentation and objects classification [17] presented previously in chapter 3. The selected kernels are:

- *Sky-removal*: it is a point operator where the value of each output point or pixel is computed from its corresponding input point. This kernel removes the sky from the disparity map. The sky is removed based on some tests applied to the saturation canal [17].
- *Road-removal*: it is a local point operator also. It removes the road from the disparity map [17] based on the V-disparity approach [132].
- *Sobel-filter*: it is a local neighbor operator, i.e, the value of each output point depends on neighboring pixels. Sobel edge detector describes pixels for robust stereo matching. This filter computes the horizontal and vertical gradient of each pixel on a 3x3 surrounding patch.

The aforementioned kernels have been selected according to two important criteria : (1) data access pattern and (2) kernel specifications. Data access patterns are crucial when we work on stencils and image processing optimizations. Each pattern type implies different memory management, optimization and parallelization approaches. By kernel specifications, we mean data requirements and their corresponding characteristics, mainly the data flow direction (input and/or output) and type's size (8-bits, 16-bits). These specifications are depicted in Table 6.2.

Table 6.2: Kernels Specifications.

Kernel	*Specifications*
Sky-removal	image-inout : 8-bit, disparity map image-in : 8-bit, saturation canal
Road-removal	image-inout : 8-bit, disparity map road's equation (a,b) : 8-bit
Sobel-filter	image-in : 8-bit, left stereo image image-out1 : 8-bit, horizontal gradient image-out2 : 8-bit, Vertical gradient

Sky-removal kernel requires an input/output 8-bits image. the latter is the disparity map to update or not according to some tests applied to the saturations canal. Road-removal kernel removes the road from the disparity map based on some input parameters corresponding to the road equation. This kernel also uses an input/output image. Sobel-filter works on an input 8-bitd image. This image is used to compute horizontal and vertical gradients. These two gradients are stored on 8-bits images. Temporary 16-bits images are also required to store the results of convolution to be converted later on to 8-bits.

6.4.1 Obtained Results

In this section, we describe the different experiments performed to test the proposed approach. We present the obtained performance on the selected kernels executed on the aforementioned architectures. We start by analyzing the obtained performances with `OpenVX` and `boost.SIMD` separately, then we discuss the results when wrapping `boost.SIMD` codes on `OpenVX`.

6.4.1.1 OpenVX on Embedded Platforms

As first experiment, we developed the selected kernels on `OpenVX`. We followed the same approach as discussed previously on section 6.3.1.1. The obtained results of this experiment will be taken as a reference to compare the results of the following experiments.

Figure 6.13 depicts the obtained execution time in ms of the three selected kernels with `C++` and `OpenVX` on x86 Intel dual-core processor and Tegra X1 ARM quad-core

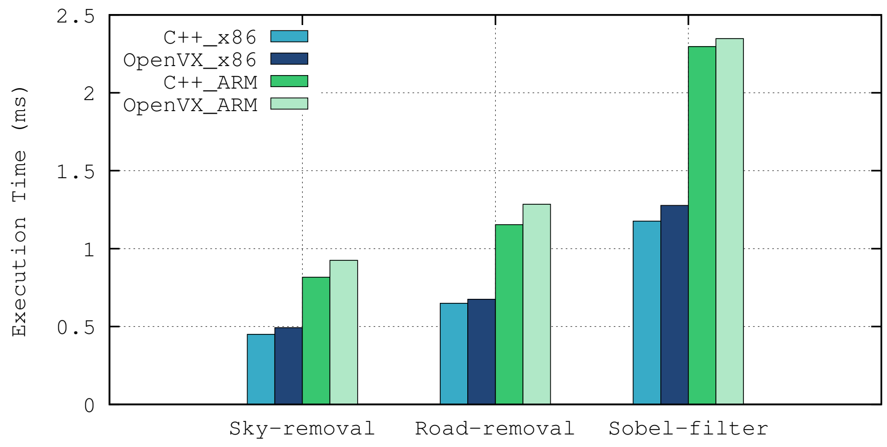


Figure 6.13: C++ vs OpenVX Obtained Performances on Fixed and Embedded Systems.

processor. Kernels are executed on mono-threading mode; only on one core on both architectures.

Sky-removal and road-removal kernels have been tested on KITTI images (1242x375 pixels), Sobel filter has been applied to a 512x512 image. These results confirm the well-known difference in performance between Intel and ARM CPUs, Tegra X1 quad-core ARM is slower by an average factor of 2 compared to Intel dual-core. The results show also that `OpenVX` is around 5% slower than the scalar version on both architectures. This is due to `OpenVX` framework overhead. This non trivial overhead, which is not avoidable comes mainly from the verification stage of `OpenVX` graphs.

6.4.1.2 Vectorization Results with boost.SIMD

To accelerate our kernels with `NT2`, we used `boost.SIMD` [133]. It allows us to write a code once and to execute it in x86 and ARM architectures thanks to the abstraction level of `SIMD` registers. Hence, we do not write our kernels with SSE instructions for x86 architecture and with NEON for ARM architecture.

Listing 6.4 illustrates part of the sky-removal kernel with `boost.SIMD` wrapped on `OpenVX`. First, we need to fix the size of data to get the size of the `SIMD` registers later on. In this kernel, since we work on gray scale data (input/output), we fix the data size `n` to 8 bits as illustrated in line 1 in Listing 6.4 with the type `pack_u8_t`. Then, based on `OpenVX` function `vxFormatImagePatchAddress2d()`, we get the image's line address

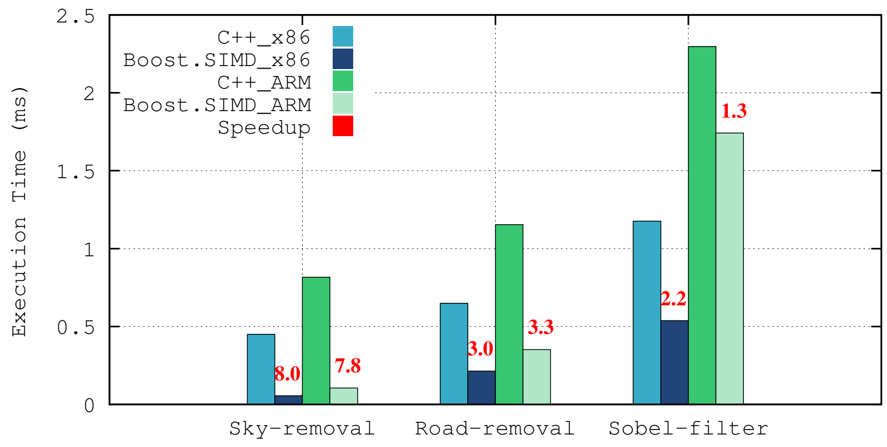


Figure 6.14: C++ vs OpenVX vs NT2 Obtained Performances on Fixed and Embedded Systems.

in memory of the input and output images (lines 3, 5). After that, we compute the image's width which can be vectorized (line 5). In other words, we find the number of pixels along the image's width which is a multiple of `SIMD` register size. For these pixels, the operations will be vectorized (lines 7 → 11). The remaining pixels for which the size is less than the `SIMD` register will be processed sequentially (lines 13, 14). The same approach is employed for the two other kernels; Sobel filter and road-removal kernel.

```
 1 size_t n = pack_u8_t::static_size;
 2 for (int32_t v=0; v<height; v++){
 3   in_1   = (vx_uint8*)vxFormatImagePatchAddress2d(src1_base, 0, v, &src1_addr);
 4   in_out = (vx_uint8*)vxFormatImagePatchAddress2d(src_dst_base, 0, v, &src_dst_addr);
 5   aw = width &~(n-1);
 6   for (int32_t u = 0; u < aw; u+=n){ // For multiple size of SIMD registers, vectorize
 7     pack_u8_t p = boost::simd::load<pack_u8_t>(&in_1[u]);
 8     pack_s8_t r = boost::simd::bitwise_cast<pack_s8_t>(boost::simd::genmask(p==mask));
 9     pack_u8_t disp = boost::simd::load<pack_u8_t>(&in_out[u]);
10     disp = boost::simd::bitwise_andnot(disp,r);
11     boost::simd::store(disp, &in_out[u]);
12   }
13   for (int32_t u=aw; u<width; u++)
14     // Continue what is left in scalar
15 }
```

Listing 6.4: NT2 Snippet Code of Sky-removal Kernel Wrapped on OpenVX.

The results of this experiment are depicted in Figure 6.14. First, we notice an important speedup with `boost.SIMD` on both architectures. The speedup of Sobel filter with `boost.SIMD` is relatively slow on ARM. The reason is related to data access pattern

Table 6.3: Kernels Specifications at arithmetic level.

Instruction	sky-removal	road-removal	Sobel-filter
`bs::simd::load<>()`	2	1	13
`bs::simd::store<>()`	1	1	6
`bs::simd::not()`	-	-	8
`bs::simd::and()`	-	-	-
`bs::simd::and_not()`	1	4	-
`bs::simd::add()`	-	-	16
`bs::simd::sub()`	-	-	4
`bs::simd::shift_right()`	-	-	4
`bs::simd::genmask()`	1	6	8
`bs::simd::splat<>()`	-	5	3
`bs::simd::split_high()`	-	1	3
`bs::simd::split_low()`	-	1	3
`bs::simd::groups()`	-	1	2
`bs::simd::bitwise_cast<>()`	1	8	18
Total	**6**	**28**	**68**

of Sobel filter which is a local neighbor operator. Indeed, vectorization is well known for its sensitivity to cache misses. Also, data casting was applied to save temporary data of size 16-bits. We notice also that the speedup decreases on kernels with higher arithmetic intensity. Table 6.3 gives the different `boost.SIMD` instructions used on each kernel. We notice clearly that we do have more instructions on road-removal kernel compared to sky-removal and much more instruction on Sobel-filter compared to both sky-removal and road-removal kernels.
As last remark, it is worth noting one important `NT2` trade-off concerning the compilation time. Compiling heavily templated C++ code is often considered as a limit to the extensive use of such techniques. This is due to the compiler's task consisting in keeping track of all already instantiated template types.

6.4.1.3 Shared Memory Parallelization Results with OpenVX

In this section, we test shared memory parallelization via `OpenMP` on `OpenVX`. We followed the same approach presented previously in section 6.3.1.5. The goal in this chapter is to

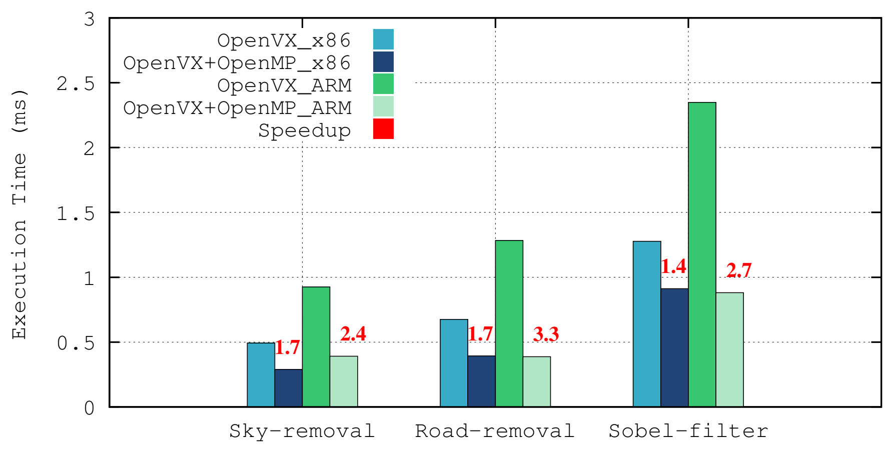


Figure 6.15: OpenMP Parallelization Performances on OpenVX.

show that `OpenVX` allows user-defined or customized kernels to be optimized by different approaches and architectures. In this section, we first show the results of `OpenMP` parallelization on `OpenVX`. Parallelization for different architectures with different approaches is also possible by wrapping optimized kernels on `OpenVX`. as will be discussed in the next section.

Listing 6.5 illustrates how the sky-removal kernel is accelerated with `OpenMP` directives within `OpenVX` framework. Lines 2 → 9 is the `OpenVX` code without any modification. The only difference is the `OpenMP` directive on line 1 which gives hints to the compiler where we wish to parallelize.

```
1 #pragma omp parallel for shared(src1_image, src_dst_image) private(in_1, in_out)
2 for (int y = 0; y < src1_addr.dim_y ; y++) {
3   in_1   = (vx_uint8*)vxFormatImagePatchAddress2d(src1_base, 0, y, &src1_addr);
4   in_out = (vx_uint8*)vxFormatImagePatchAddress2d(src_dst_base, 0, y, &src_dst_addr);
5   for (int x = 0; x < src1_addr.dim_x ; x++) {
6     if ((in_1[x] == 255))
7       in_out[x] = 0;
8   }
9 }
```

Listing 6.5: OpenVX Snippet Code of Sky-removal Kernel with OpenMP.

Figure 6.15 depicts the obtained execution time (ms) of `OpenVX` kernels with and without `OpenMP`. The speedup is also given. We notice a speedup of around 2 on x-86 Intel dual-core processor and an average of 3 on ARM quad-core. We observe also that

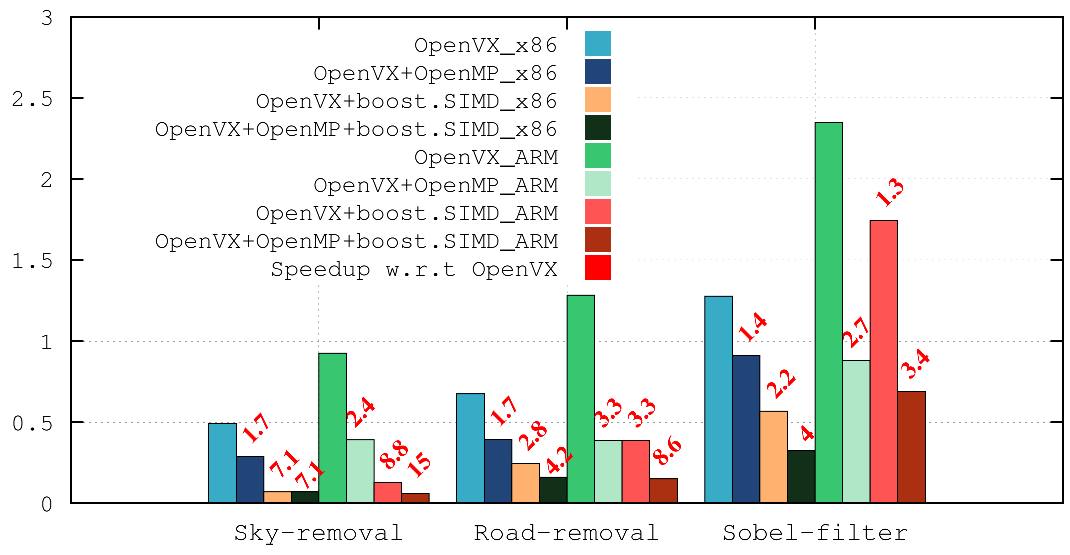


Figure 6.16: Execution time (ms) of boost.SIMD Vectorization and OpenMP parallelization on OpenVX

we reach almost the same execution time on both architectures. Indeed, the difference between ARM and x86 architectures in execution time is compensated on ARM with 4 physical cores compared to Intel i5 processor with only 2 cores.

6.4.1.4 Shared Memory Parallelization and Vectorization on OpenVX with boost.SIMD

In this section, we apply two optimizations to our kernels. First, we apply vectorization through `boost.SIMD` as discussed previously on section 6.4.1.2. Then, we integrate these `boost.SIMD` accelerated kernels on `OpenVX` and apply parallelization on `OpenVX` with `OpenMP`. It worth noting that **no rewriting effort** has been performed on ARM architecture. Setting appropriate compiling options (-mfpu=neon) is sufficient.

Figure 6.16 depicts the obtained execution time as well as the speedup of the three kernels in x86 and ARM architectures. The speedup is computed with respect to the `OpenVX` execution time taken as the reference. We show the speedup obtained with the different approaches employed previously presented to accelerated the execution time. Globally, we notice that the kernels respond differently to each architecture and acceleration technique (`OpenMP`, `boost.SIMD` and both).

If we compare OpenMP and boost.SIMD results, we notice that on x86 architecture, boost.SIMD performs better than OpenMP with a speedup of **7.1** for sky-removal kernel and **2.8** for the road-removal kernel and **2.2** for Sobel filter. However, the observation is not similar on ARM due to the double number of cores -compared to x86 and to kernels specifications. In sky-removal kernel, with few computations, boost.SIMD performs better than OpenMP as on x86 architecture. In road-removal kernel with more computations, OpenMP and boost.SIMD both give the same speedup; **3.3**. In Sobel-filter, OpenMP is better than boost.SIMD. The reason is related to the data access pattern which imposes adaptation at kernel implementation level. In scalar version, Sobel has been implemented as a non-separable filter where one global loop is employed to keep threads busy with OpenMP and reduce threads launch cost. However, non-separable version is not suitable for vectorization. The latter is highly affected by cache misses, so, we implemented the separable version of Sobel filter in boost.SIMD for better locality.

With only boost.SIMD, we notice an important speedup on both architectures. We notice also that the speedup in this case decreases with kernels complexity mainly at arithmetic level as discussed previously. Finally, for maximum performance, we wrapped boost.SIMD kernels on OpenVX framework and applied OpenMP directives to benefit from the hardware performances. We notice that with the sky-removal kernel, the speedup with boost.SIMD is the same as with both boost.SIMD and OpenMP. This can be explained by the fact that this kernel is a not a so time consuming one, it takes less than $0.5ms$ in scalar version.

6.5 Discussion

In the first part of this chapter, we have investigated the contribution of OpenVX framework to computer vision applications. We first compared OpenVX to traditional C++ language to assess if graph-based approaches add some overhead due to graph management or not. We have observed that the template provided in OpenVX samples is not so optimized concerning pixel accessing in images. So, we have written our own improved codes to get the same performance results compared to C++.
In the unoptimized version of OpenVX, data access pattern is not taken into account. However, access patterns are important for optimizing image processing applications and stencils. Thus, we have used different data access patterns to evaluate how OpenVX

responds to each pattern type.
We tested *kernels merge* optimization of `OpenVX` and access its importance on accelerating operation from memory accesses point of view.
Concerning `OpenVX` *tiling extension*, we noticed that it adds a small overhead via the run-time manager which sets tiles sizes and copies data from input images located in L3 to tiles in L1 buffers. We also faced another issue with tiling when the input and output data have different sizes such with statistical operators like histograms. In this case, tiling does not respond well since in `OpenVX` *tiling extension*, by default, the size of input and output images and tiles are assumed to be equal. Finally, we tested `OpenMP` with `OpenVX` and managed to accelerate the execution of our kernels.

In the second section of this chapter, we managed to accelerate computer vision kernels on two different architectures **with zero rewriting cost**. We merged `OpenVX` framework and `boost.SIMD`. `OpenVX` allows us to develop hardware customized kernels for different architectures. However, targeting different architectures in `OpenVX` requires rewriting kernels' code with compatible languages. To cope with this issue, we propose to integrate `boost.SIMD` in `OpenVX`. `Boost.SIMD` allows us to develop kernels once while targeting different architectures with minimal rewriting cost. We tested this approach on x86 and ARM (Tegra X1) architectures. The kernels are written once and then vectorization is enabled–SSE on x86 and NEON on ARM–. An important speedup has been obtained with this approach on both architectures. Speedup depends on architecture's specifications such as number of cores and also kernel's specifications as data access pattern.

The obtained results with `OpenVX` are far away from the expectations as presented in the first release [55]. Despite the fact that we managed to accelerate `OpenVX` kernels with others approaches such as `OpenMP` and `boost.SIMD`, some important `OpenVX` optimizations such as *tiling* are not yet enough mature to give good performances. Also, there are some optimizations that have not been implemented such as *graph-scheduling* in heterogeneous systems. Consequently, we did not expand our experiments to accelerate our stereo-based algorithm as we did in chapter 4 and chapter 5.

*

7

Conclusion

7.1 Conclusions

Embedding vision-based ADAS applications raised important scientific challenges which were behind the birth of this research work. Vision-based ADAS require high performance computing in real time. This led to the design and development of new architectures and parallel programming tools. Both proposed hardware and software have different features and no one is perfect for a specific application. This thesis research addresses the challenge of programming methodologies of vision-based ADAS applications on parallel architectures. The main **contribution** of this work is to provide a feedback for the development of future image processing applications in adequacy with parallel architectures with a best compromise between computing performance, algorithm accuracy and programming efforts. We evaluated different programming tools of embedded vision-based ADAS application from different aspects and provide the contributions as well as the limitations of each tool.

At first, we proposed and developed an algorithm for vehicles detection based on stereo vision. It is worth noting that this algorithm has not been developed to compete or to give better performance compared to the existing works in the literature. It has been rather developed to be used as a use-case for our experiments on embedding vision-based algorithms. This algorithm has two important features: *High performance computing* and *Various kernels' specifications.* To target the previously mentioned challenge, we need to have a time consuming algorithm which needs to be processed in real-time in embedded systems. For this reason, we selected the stereo vision as perception system. For decades, many stereo matching algorithms have been proposed with new improvements in both matching accuracy and algorithm efficiency. Most of the proposed works tend to be contradictory in reported results: *accurate stereo matching methods are usually time consuming.* For this reason, we selected to include stereo matching in our ADAS application. When parallelizing image processing algorithms, optimizations may differ from one kernel to another. This difference comes from *kernels' specifications.* By specifications we mean for instance data access pattern, data flow, data dependency, etc. Each feature requires a specific optimization. We developed the algorithm in such a way to have a variety of kernels with different features. This allows us to perform different experiments and test various optimizations.

In the second part of this research work, we addressed the optimizing process of computer vision algorithms in embedded systems from both hardware and software aspects. We employed different architectures available in the market for ADAS systems and we used various parallel programming methodologies. We selected multi-core systems (CPU-based) and NVIDIA platforms; Jetson K1 and Tegra X1 (GPU-based). The choice is based on two important criteria. First, we selected the most available and employed platforms in automobile industry nowadays such NVIDIA platforms. Second, we focused on the stability and the performance of the hardware in one hand and the maturity of the corresponding and compatible softwares in the other hand.

After selecting the appropriate hardwares, we moved to the core of this research work: embedding the developed use-case on the selected hardwares with different parallel tools. The **goal** of this work is to investigate some approaches from different aspects of parallel programming to provide the advantages and limitations of each employed tool. We selected the parallel tools based on a set of features and criteria. First **(1)**, from programming level, we took low-level programming languages such as `CUDA` as well as high-level techniques such as directives-based tools (`OpenMP, OpenACC`). Second **(2)**, at portability level, since we target heterogeneous architectures, we selected tools which are compatible with one or more processing units such as CPUs and GPUs. Third **(3)**, for optimizing level, we selected both kernel- and system-level tools. To recall, by kernel-level optimizations we mean all optimizations that we apply on kernels or small functions. In this case, we usually use traditional parallel languages. In this work we worked with `SIMD` instructions, directives-based tools (`OpenMP, OpenACC`) and `CUDA` programming language. With system-level optimizations, we aim to optimize the algorithm as a whole system. This is performed by focusing for instance on data transfers and allocations, memory bandwidth. For this case, we selected `OpenVX` framework. As fourth and last feature (**4**), we focus on programming efforts. Some tools required code rewriting to get the optimized version to be executed in parallel. This happens with `CUDA` for instance. Some other tools require less code rewriting and rely on compliers to generate the parallel code such as `OpenMP` and `OpenACC`. We also used a `EDSL` referred as `NT2` which allows code generation for some specific optimizations such as vectorization with `SIMD` with minimal rewriting cost.

After identifying the bottlenecks–most time consuming kernels/functions– of our algorithm, we wrote the optimized version with the aforementioned tools. The obtained results allowed us to evaluate the different tools from mainly programming efforts and obtained performance. It is worth noting that one important step before starting the parallelization process is to analyze the algorithm to check whether it responds to some important features of parallel computing such as dense repetitive computations and regular memory accesses. In our algorithm, we made some important adaptations at algorithmic level before parallelizing the algorithm for better and efficient results. This process is what we usually call **Algorithm-Architecture-Adequacy (AAA)**. For instance, embedded systems have low memory capacities and even smaller caches, hence it is crucial to ensure contiguous and coalesced memory accesses to benefit from the cache memories.

As a second important result, it is crucial to optimize the code at different levels to reach the required performance. For instance, we can apply vectorization (`SIMD`) at inner operations while parallelizing globally the outer operation with another approach such as `OpenMP`. In other words, a mixture of parallel tools at different algorithmic level is usually necessary for maximum performances.

Table A.1 illustrates the evaluation of the different employed parallel programming tools. The evaluation is performed in terms of:

- **Portability** The possibility to use the parallel tool/language on different architectures.

- **Cross-platforms portability** The fact of executing the same code on different architecture without code rewriting

Table 7.1: Evaluation of Employed Parallel Programming Tools

Parallel Tool	Portability	Cross-Platforms Portability	Performance	Programming Efforts
`OpenMP`	CPUs, GPUs	CPUs, GPUs ***	****	****
`OpenACC`	CPUs,GPUs	CPUs, GPUs ***	****	****
`CUDA`	NVIDIA GPUs	NVIDIA GPUs *	****	**
`NT2`	CPUs,GPUs	CPUs, GPUs ***	**	*
`SIMD`	CPUs	CPUs*	****	**
`Boost.SIMD`	CPUs	SSE, AVX, Altivec, Neon****	****	**
`OpenVX`	CPUs,GPUs	CPUs, GPUs ***	**	*

- **Performance** In this work, we rely on the obtained execution time as a measure of performance.
- **Programming efforts** Two features are taken into consideration: **(1)** the time spent to master the tool and rewrite the code, **(2)** the difficulties encountered during code optimizations.

The obtained results and evaluations showed that directive-based approaches with less programming efforts such as `OpenMP` and `OpenACC` can compete high-level tools as `CUDA` at performance level. Knowing that `CUDA` requires code rewriting and the developer has to master the tool with the different aspects to reach the required performance. However, There are some important aspects which are not yet covered by directive-based tools. For instance, having access to shared memory is not possible with `OpenACC` while `CUDA` allows the developer to use this memory for fast memory accesses. Also, the optimizing process with `CUDA` is a progressive one. The developer has to test different optimizations at several levels. It is worth noting that directive-based tools may not always give good performance since they rely on the compilers to generate the parallel code which may fail with complex algorithms and high data dependency.

`NT2` provides high portability and cross-platforms portability as well. However, as shown in Table A.1, the obtained performances are not as expected. The tool is not yet mature to optimize complex functions with high data dependencies. Also, it requires more programming efforts to master the tool and find the required parallel expressions.

Vectorization results with `SIMD` and `boost.SIMD` give important results in terms of performance and programming efforts. While code rewriting is required through the intrinsics, the time spent and the difficulties encountered are much lower compared to CUDA. However, it is worth noting that in this work, vectorization with SIMD or boost.SIMD is only applied to the cost function **SAD** while CUDA is applied to the whole matching function. At portability level, with SIMD, we used the instructions set corresponding to each architecture such as SSE on Intel CPUs and NEON on ARM CPUs. However, with boost.SIMD, no rewriting code is required and hence provides better cross-platforms portability.

The investigation of OpenVX system-level optimizations revealed several results. The important remark concerns the ability to integrate and embed customized optimized kernels. We managed to integrate and accelerate our kernels in OpenVX with different parallel tools such as OpenMP and boost.SIMD. Here comes another **contribution** of this work. We proposed an approach to target both *kernel-* and *system-level* optimizations. We also managed to launch the optimized kernels on different architectures with zero cost of rewriting thanks to boost.SIMD. We investigated some proposed optimizations such as *kernels merge* and *data tiling*. The results show that there is still work to be done to get better results mainly with tiling.

To summarize, embedding and parallelizing vision-based algorithms is not a trivial task. While different parallel tools and high performance architectures are available, the process of parallelizing is still not a straightforward one. This is why it is important to investigate the different available parallel tools to provide a feedback for future developers to better select the parallel tool in adequacy with the selected hardware and applications.

7.2 Future Works

Several tracks are possible to extend the work presented in this manuscript to provide a more detailed evaluation and feedback concerning programming methodologies of vision-based ADAS on parallel architectures.

First, we consider to investigate other parallel tools and approaches to extend our evaluations. Among the available tools we may cite Halide [16] and HIPAcc [134] framework. The parallel tools investigated in this work have been evaluated from two aspects: *programming efforts* and *obtained performances*. As a **second** perspective, it would be interesting to take into consideration the *energy consumption* as an evaluation criteria which is a crucial point when working on embedded systems.

Third, from hardware point of view, in this work we employed multi-core and heterogeneous (CPU+GPU) platforms in our experiments. For future works, we are considering to work on other architectures such as **FPGAs**, **DSPs** to extend our evaluation on other hardwares and test other tools compatible with these architectures.

Fourth, we propose to work on other use-cases. In this work, we selected a stereo vision based vehicles detection algorithm, it would be interesting to work on others algorithms such as **deep learning** for better evaluation the parallel tools.

Fifth, it would be interesting to perform a comparison or an evaluation with the state of the art such as the implementation of the Semi-Global Matching (**SGM**) algorithm on GPUs [135].

Finally, Concerning the results obtained with `NT2` which were not encouraging, it is worth noting that it requires time to analyze, understand the issues behind the low performances. We are considering to rely on Roofline model [136] to analyze the results for better understanding the obtained performances. Also using STREAM benchmark [137] for benchmarking the ultimate DRAM of a multi-core processor would allow us to get deeper insight of the obtained results.

**

Appendix A

Synthèse en Français

Dans ce chapitre, une synthèse de la thèse en Français est proposée. Nous expliquons le contexte et la problématique de ce travail de recherche. Les objectifs ainsi que l'approche sont aussi présentés. Nous donnons à la fin de ce chapitre un récapitulatif des résultats obtenus ainsi que quelques conclusions.

A.1 Contexte

L'Organisation mondiale de la santé (OMS) [2] a montré en 2015 que chaque année, environ 1,25 million de personnes meurent à la suite d'un accident de la route. La plupart de ces accidents sont attribués à des erreurs humaines ou à des distractions. Bien que des efforts considérables sont déployés depuis les années 1990 pour développer des solutions technologiques pour améliorer la sécurité telles que les coussins gonflables (airbag), les pertes humaines dans l'environnement routier sont encore trop élevées.

Pour mieux gérer la circulation routière et réduire les risques d'accidents, de nouvelles technologies ont été proposées, appelées Application d'aide à la conduite (Advanced Driver Assistance Systems ,ADAS), telles que les systèmes de régulation de vitesse adaptative (ACC) et d'avertissement de sortie de voie (LDW). Les ADAS sont des systèmes intelligents embarqués développés pour éviter les risques d'accidents et améliorer la sécurité routière en aidant les conducteurs dans leurs tâches de conduite.

Un ADAS est considéré comme un système embarqué complexe temps réel composé de trois couches importantes. La couche de *perception* comprend un ensemble de capteurs tels que les radars et les caméras. Il peut également comprendre une unité de fusion de données qui permet le calcul de données de capteurs appropriées pour estimer un état cohérent d'un véhicule et de son environnement. La couche de *décision* utilise les sorties de l'unité de fusion de données pour analyser la situation actuelle et décider des actions appropriées à transmettre aux actionneurs. La couche d'*action* reçoit les actions de la couche de décision, et soit elle délivre des informations d'avertissement visuelles, acoustiques et/ou même des vibrations au conducteur, soit elle fournit des actions automatiques comme le freinage.

A.1.1 ADAS : Challenges et Opportunités

La conception, le développement et le déploiement d'ADAS présentent plusieurs défis. Le système devrait être *rapide* dans le traitement des données, *prédire avec précision* le contexte et réagir en *temps réel*. En outre, il est requis d'être *robuste*, *fiable*, et avoir des taux d'erreur *faible*. Les ADAS utilisent beaucoup de données rapportées par plusieurs capteurs. Ces données doivent être mises à jour régulièrement pour transmettre l'état actuel de l'environnement. Ainsi, ces applications doivent être gérées par des systèmes de bases de données en temps réel afin de stocker et de manipuler efficacement les données en temps réel. Cependant, la conception d'ADAS est très complexe; il est difficile de modéliser les contraintes de temps liées à la fois aux données et aux transactions. Des efforts et des recherches considérables ont été déployés pour résoudre tous ces problèmes et développer la technologie qui fera des ADAS et de la conduite autonome une réalité.

A.2 Problématique

Le rôle de la vision par ordinateur dans la compréhension et l'analyse d'une scène routière est primordial pour construire des systèmes d'aide à la conduite (ADAS) plus intelligents. Cependant, le développement et l'implémentation de ces applications dans un réel environnement automobile et loin d'être simple. En effet, le portage de ces applications a entraîné d'importants défis scientifiques à relever qui étaient derrière la naissance de ce travail de recherche.

L'étude de l'état de l'art montre que la majorité des algorithmes utilisés dans les applications d'aide à la conduite sont développés et implémentés sur des PC standards. Lorsque ces algorithmes sont portés sur des systèmes embarqués, leurs performances se dégradent et parfois, le portage n'est plus possible. En effet, il y a plusieurs exigences et contraintes à prendre en compte notamment la performance de calcul. Il y a un grand écart entre ce qui est testé sur un PC standard et ce qui finalement s'exécute dans la plateforme embarquée. De plus, il n'y a pas de matériel et de logiciel standard pour une application spécifique. Ainsi, différentes architectures et outils de programmation ont été proposés par l'industrie et la communauté scientifique qui s'avèrent être pas assez matures pour répondre aux besoins des ADAS sans fournir un effort supplémentaire.

A.3 Motivations

Les algorithmes de traitement d'images nécessitent une haute performance de calcul en plus d'une précision algorithmique pour faire face à l'évolution importante de la résolution d'images ainsi que la fréquence de capture dans les capteurs. Pour répondre à ces exigences, de nouvelles architectures hétérogènes sont apparues. Elles sont composées de plusieurs unités de traitement avec différentes technologies de calcul parallèle: multi-core, GPU, SIMD, accélérateurs dédiés, etc. Au niveau de la programmation, pour mieux bénéficier et exploiter les performances de ces architectures, différents langages et outils sont nécessaires en fonction du modèle d'exécution parallèle. Toutes ces fonctionnalités rendent la tâche d'embarquer les algorithmes de traitement d'images cruciale et contraignante.

A.4 Approche et Objectives

Dans cette thèse, nous étudions diverses méthodologies et modèles de programmation parallèle d'algorithmes de traitement d'images embarqués. nous utilisons une étude de cas complexe basée sur la stéréo-vision développée au cours de cette thèse. Nous présentons les caractéristiques et les limites pertinentes de chaque approche. Nous évaluons ensuite les outils de programmation utilisés principalement en matière de performances de calcul et de difficulté de programmation. Le retour de ce travail de recherche est crucial pour le développement de futurs algorithmes de traitement d'images en adéquation avec les architectures parallèles avec un meilleur compromis entre les performances de calcul, la précision algorithmique et le temps de développement.

A.4.1 Cas d'usage

Dans ce travail, nous proposons d'abord un algorithme de détection de véhicules basé sur la vision stéréo [17]. L'algorithme est basé sur la segmentation de la carte de disparité et la classification d'objets. Il est à noter que cet algorithme n'a pas été développé pour concurrencer ou donner de meilleures performances par rapport aux travaux existants dans la littérature. Il a été plutôt développé pour être utilisé comme un cas d'utilisation pour nos expériences sur l'intégration d'algorithmes de vision par ordinateur. Cet algorithme a deux caractéristiques importantes:

1. *Calcul haute performance* Pour cibler les défis discutés précédemment, nous devons disposer d'un algorithme gourmand en terme de puissance de calcul et qui doit être traité en temps réel dans les systèmes embarqués. Pour cette raison, nous avons choisi la vision stéréoscopique comme système de perception. La mise en correspondance est ensuite effectuée pour générer une carte de disparité. La mise en correspondance dans la stéréo vision est l'un des problèmes les plus étudiés en vision par ordinateur. Pendant des décennies, de nombreux algorithmes de mise en correspondance ont été proposés avec de nouvelles améliorations à la fois en termes de précision d'appariement et d'efficacité des algorithmes. La plupart des travaux proposés tendent à être contradictoires dans les résultats rapportés: *des méthodes d'appariement stéréo précises prennent habituellement beaucoup de temps.* Pour cette raison, nous avons choisi d'inclure la correspondance stéréo dans notre application ADAS.

2. *Traitements à spécifications diverses*: Lors de la parallélisation d'algorithmes de traitement d'image, les optimisations peuvent différer d'un traitement à l'autre. Cette différence provient des spécifications de chaque traitement. Par spécifications, nous entendons par exemple le pattern d'accès aux données, le flux de données, les dépendances des données, etc. Chaque fonctionnalité nécessite une optimisation spécifique. Nous avons développé l'algorithme de manière à avoir une variété de traitements avec des caractéristiques différentes. Cela nous permet d'effectuer différentes expériences et de tester plusieurs optimisations.

A.4.2 Architectures

Une fois l'algorithme développé et validé, nous passons au cœur de cette thèse. Nous commençons le processus d'intégration de l'algorithme. Nous ciblons différentes architectures disponibles sur le marché et nous utilisons différents modèles et langages de programmation parallèle. Nous avons sélectionné des systèmes multi-core (basés sur CPU) et des plateformes NVIDIA; Jetson k1 et Tegra X1 (basé sur GPU). Le choix est basé sur deux critères importants. Tout d'abord, nous avons sélectionné les plateformes les plus disponibles et les plus utilisées dans l'industrie automobile de nos jours, telles que les plateformes de chez NVIDIA. Deuxièmement, nous nous concentrons sur la stabilité et la performance du matériel d'une part et sur la maturité du logiciel correspondant d'autre part. Pour chaque architecture employée, nous présentons ses caractéristiques pertinentes, puis nous décrivons leurs effets sur l'algorithme appliqué et les approches de parallélisation.

A.4.3 Langages de Programmation Parallèle

L'objectif principal de ce travail est d'étudier certains modèles/outils de programmation parallèle en prenant en considération différents aspects pour fournir les avantages et les limites de chaque outil employé. Nous avons sélectionné les outils en fonction d'un ensemble de caractéristiques et de critères. **(1)** Nous avons pris des langages de programmation de bas niveau tels que `CUDA` ainsi que des techniques de haut niveau telles que les outils basés sur des directives (`OpenMP, OpenACC`). **(2)** Au niveau de la portabilité, comme nous ciblons des architectures hétérogènes, nous avons sélectionné des outils compatibles avec une ou plusieurs unités de traitement telles que les CPU et les GPU. **(3)** Concernant le niveau d'optimisation, nous avons sélectionné les deux axes, niveau kernel et niveau système. Pour rappel, par optimisations au niveau kernel, nous entendons toutes les optimisations que nous appliquons sur les kernel/fonctions de notre programme. Dans ce cas, nous utilisons généralement des langages parallèles traditionnels. Dans ce travail, nous avons travaillé avec les instructions `SIMD`, les outils basés sur des directives (`OpenMP, OpenACC`) et le langage de programmation `CUDA`. Avec les optimisations niveau système, nous visons à optimiser l'algorithme dans son ensemble. Ceci est effectué en se concentrant par exemple sur les transferts de données et les allocations, la bande passante mémoire. Pour ce cas, nous avons sélectionné le framework `OpenVX`. En tant que quatrième et dernière caractéristique (**4**), nous nous concentrons sur les efforts de programmation. Certains outils nécessitent une réécriture du code pour que la version optimisée soit exécutée en parallèle. Cela arrive avec `CUDA` par exemple. D'autres outils nécessitent moins de réécriture de code et s'appuient sur les compilateurs pour générer le code parallèle tel que `OpenMP` et `OpenACC`. Nous avons également utilisé un `EDSL` appelé `NT2` qui permet la génération du code pour certaines optimisations spécifiques telles que la vectorisation avec `SIMD` avec un coût de réécriture minimal.

A.5 Résultats

Il convient de noter qu'une étape importante avant de commencer la parallélisation consiste à analyser l'algorithme pour vérifier s'il répond à certaines caractéristiques importantes de la programmation parallèle, telles que les calculs répétitifs denses et les accès mémoire réguliers. Dans notre algorithme, nous avons fait quelques adaptations importantes au niveau algorithmique avant de paralléliser l'algorithme pour des résultats meilleurs et plus efficaces. Ce processus est ce que nous appelons habituellement **Adéquation-Algorithme-Architecture (AAA)**. Par exemple, les systèmes embarqués ont des capacités de mémoire faibles et même des caches plus petits, il est donc crucial de garantir des accès mémoire contigus pour bénéficier des mémoires cache.

En tant que deuxième résultat important, il est crucial d'optimiser le code à différents niveaux pour atteindre la performance requise. Par exemple, nous pouvons appliquer la vectorisation (`SIMD`) aux opérations (boucles) internes tout en parallélisant globalement l'opération externe avec une autre approche telle que `OpenMP`. En d'autres termes, un mélange de modèles de programmation parallèles à différents niveaux algorithmiques est nécessaire pour des performances maximales.

Le tableau A.1 illustre l'évaluation des langages et outils de programmation parallèles employés dans ce travail. L'évaluation est faite en termes de:

- La **portabilité** La possibilité d'utiliser l'outil sur différentes architectures.
- La **portabilité multi-plateformes** Le fait d'exécuter le même code sur différentes architectures sans réécrire le code

Table A.1: Evaluation d'Outils de Programmation Parallèle

Outil de Programmation	**Portabilité**	**Portabilité multi-plateformes**	**Performance**	**Efforts de Programmation**
`OpenMP`	CPUs, GPUs	CPUs, GPUs ***	****	****
`OpenACC`	CPUs,GPUs	CPUs, GPUs ***	****	****
`CUDA`	NVIDIA GPUs	NVIDIA GPUs *	****	**
`NT2`	CPUs,GPUs	CPUs, GPUs ***	**	*
`SIMD`	CPUs	CPUs*	****	**
`Boost.SIMD`	CPUs	SSE, AVX, Altivec, Neon****	****	**
`OpenVX`	CPUs,GPUs	CPUs, GPUs ***	**	*

- La **performance** Dans ce travail, nous utilisons le temps d'exécution obtenu comme mesure de performance.
- Les **Efforts de programmation** Deux fonctionnalités sont prises en compte: **(1)** le temps passé à maîtriser l'outil et à réécrire le code, **(2)** les difficultés rencontrées lors de l'optimisations de code.

Les résultats obtenus et les évaluations montrent que les approches basées sur des directives telles que `OpenMP` et `OpenACC` peuvent rivaliser avec les outils de haut niveau comme `CUDA` au niveau de la performance tout en demandant moins d'effort au niveau de la programmation. Sachant que `CUDA` nécessite une réécriture du code, le développeur doit maîtriser l'outil avec les différents aspects pour atteindre les performances requises. Cependant, certains aspects importants ne sont pas encore couverts par les outils basés sur des directives. Par exemple, avoir accès à la mémoire partagée n'est pas possible avec `OpenACC` tandis que `CUDA` permet au développeur d'utiliser cette mémoire pour des accès mémoire plus rapides. En outre, le processus d'optimisation avec `CUDA` est progressif. Le développeur doit tester différentes optimisations à plusieurs niveaux. Il convient de noter que les outils basés sur des directives ne donnent pas toujours de bonnes performances car l'optimisation est faite par le compilateur qui peut échouer avec des algorithmes complexes et une dépendance de données élevée.

`NT2` offre également une grande portabilité et une portabilité multi-plateforme importante. Cependant, comme indiqué dans la tableau A.1, les performances obtenues sont loin de celles attendues. L'outil n'est pas encore mature pour optimiser les fonctions complexes avec des dépendances de données élevées. En outre, il nécessite plus d'efforts de programmation pour maîtriser l'outil et trouver les expressions parallèles requises.

Les résultats de la vectorisation avec `SIMD` et `boost.SIMD` donnent des résultats importants en termes de performance et d'efforts de programmation. Alors que la réécriture du code est nécessaire à travers les intrinsèques, le temps passé et les difficultés rencontrées sont beaucoup plus faibles par rapport à `CUDA`. Cependant, il est à noter que dans ce travail, la vectorisation avec `SIMD` ou `boost.SIMD` est seulement appliquée à la fonction du coût **SAD** tandis que `CUDA` est appliqué à l'ensemble de la fonction de correspondance. Au niveau de la portabilité, avec `SIMD`, nous avons utilisé les instructions correspondantes à chaque architecture comme `SSE` sur les processeurs Intel et `NEON` sur les processeurs ARM. Cependant, avec `boost.SIMD`, aucune réécriture du code n'est requise et offre donc une meilleure portabilité multi-plateformes.

L'étude des optimisations niveau système avec `OpenVX` a révélé plusieurs résultats. Le résultat le plus important concerne la possibilité d'intégrer des kernels optimisés avec d'autres outils dans `OpenVX`. Nous avons réussi à intégrer et accélérer nos kernels dans `OpenVX` avec différents outils parallèles tels que `OpenMP` et `boost.SIMD`. L'une des **contributions** de ce travail c'est de proposer une approche pour cibler à la fois les optimisations niveau *kernel* et niveau *système*. Nous avons également réussi à exécuter les kernels optimisés sur différentes architectures **sans aucun** coût de réécriture grâce à `boost.SIMD`. Nous avons étudié certaines optimisations proposées telles que *la fusion de kernels* et *le tiling* de données. Les résultats montrent qu'il y a encore du travail à faire pour obtenir de meilleurs résultats principalement avec le tiling.

A.6 Contributions et Conclusions

Le portage d'applications ADAS –basées sur la vision– sur des plateformes embarqués a soulevé d'importants défis scientifiques qui étaient à l'origine de la naissance de ce travail de recherche. Les ADAS basés sur la vision nécessitent un calcul haute performance en temps réel. Cela a conduit à la conception et au développement de nouvelles architectures et d'outils de programmation parallèle. Le matériel et les logiciels proposés ont des caractéristiques différentes. Ce travail de recherche aborde le défi des méthodologies et modèles de programmation parallèle d'applications ADAS basées sur la vision sur des architectures parallèles et embarquées. La contribution principale de ce travail est de fournir un retour pour le développement de futures applications de traitement d'image en adéquation avec les architectures parallèles avec un meilleur compromis entre les performances de calcul, la précision algorithmiques et les efforts de programmation. Nous avons évalué différents outils de programmation parallèle de différents aspects et nous avons fourni les contributions ainsi que les limites de chaque outil.

Pour résumer, paralléliser des algorithmes basés sur la vision et les porter sur des architectures embarqués n'est pas une tâche triviale. Bien que différents outils parallèles et diverses architectures haute performance soient disponibles, le processus de la parallélisation n'est toujours pas simple. C'est pourquoi il est important d'étudier les différents outils parallèles disponibles pour fournir un retour aux futurs développeurs afin de mieux choisir le modèle de programmation parallèle en adéquation avec le matériel et les applications.

References

[1] Daniel Scharstein and Richard Szelisk. *"Middlebury Stereo Datasets"*. http://vision.middlebury.edu/stereo/data/. x, 44, 71, 110

[2] *"WHO: Road traffic injuries. [online]"*. http://www.who.int/mediacentre/factsheets/fs358/en/. 1, 176

[3] Mahdi Rezaei and Reza Sabzevari. *Multisensor data fusion strategies for advanced driver assistance systems*. I-Tech Education and Publishing, 2009. 2

[4] Angelos Amditis, Matthaios Bimpas, George Thomaidis, Manolis Tsogas, Mariana Netto, Saïd Mammar, Achim Beutner, Nikolaus Möhler, Tom Wirthgen, Stephan Zipser, Aria Etemad, Mauro Da Lio, and Renzo Cicilloni. **A Situation-Adaptive Lane-Keeping Support System: Overview of the SAFELANE Approach**. *IEEE Trans. Intelligent Transportation Systems*, **11**(3):617–629, 2010. 2

[5] Piotr Dollar, Serge Belongie, and Pietro Perona. **The Fastest Pedestrian Detector in the West**. In *Proceedings of the British Machine Vision Conference*, pages 68–68. BMVA Press, 2010. doi:10.5244/C.24.68. 3

[6] Marc Green. **" How long does it take to stop?" Methodological analysis of driver perception-brake times**. *Transportation human factors*, **2**:195–216, 2000. 3

[7] Jaeseok Kim and Hyunchul Shin. *Algorithm & SoC Design for Automotive Vision Systems: For Smart Safe Driving System*. Springer Publishing Company, Incorporated, 2014. 5

[8] *"NVIDIA Drive PX Platform. [online]"*. http://www.nvidia.com/object/drive-px.html. 6, 21

[9] *"Texas Instruments TDAx Socs. [online]"*. http://www.ti.com/lsds/ti/processors/dsp/automotive_processors/tdax_adas_socs/overview.page. 7, 21

[10] *"Nvidia OpenCV Extension. [online]"*. http://www.nvidia.fr/object/tegra-fr.html. 7, 19, 21, 145

[11] Eduard Ayguadé, Rosa M. Badia, Pieter Bellens, Daniel Cabrera, Alejandro Duran, Roger Ferrer, Marc González, Francisco D. Igual, Daniel Jiménez-González, and Jesús Labarta. **Extending OpenMP to Survive the Heterogeneous Multi-Core Era.** *International Journal of Parallel Programming*, **38**(5-6):440–459, 2010. 8, 23

[12] *"OpenACC .[online]"*. https://www.openacc.org/. 8, 23, 104

[13] *"CUDA .[online]"*. https://developer.nvidia.com/about-cuda. 8, 104

[14] John E. Stone, David Gohara, and Guochun Shi. **OpenCL: A parallel programming standard for heterogeneous computing systems**. *Computing in science & engineering*, **12**(1-3):66–73, 2010. 8, 30

[15] Tomasz S. Czajkowski, Utku Aydonat, Dmitry Denisenko, John Freeman, Michael Kinsner, David Neto, Jason Wong, Peter Yiannacouras, and Deshanand P. Singh. **From OpenCL to high-performance hardware on FPGAS**. In Dirk Koch, Satnam Singh, and Jim Tørresen, editors, *FPL*, pages 531–534. IEEE, 2012. 8

[16] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. **Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines**. *ACM SIGPLAN Notices*, **48**(6):519–530, 2013. 8, 32, 173

[17] Djamila Dekkiche, Bastien Vincke, and Alain Mérigot. *Vehicles Detection in Stereo Vision Based on Disparity Map Segmentation and Objects Classification*, pages 762–773. Springer International Publishing, Cham, 2015. 11, 39, 146, 147, 159, 179

[18] Djamila Dekkiche, Bastien Vincke, and Alain Mérigot. **Investigation and performance analysis of OpenVX optimizations on computer vision applications**. In *ICARCV*, pages 1–6. IEEE, 2016. 13, 138

[19] Djamila Dekkiche, Bastien Vincke, and Alain Mérigot. **Targeting system-level and Kernel-level optimizations of computer vision applications on embedded systems**. In *ISED*. IEEE, 2016. 13, 139

[20] Marc Duranton. **The Challenges for High Performance Embedded Systems**. In *Ninth Euromicro Conference on Digital System Design: Architectures, Methods and Tools (DSD 2006), 30 August - 1 September 2006, Dubrovnik, Croatia*, pages 3–7, 2006. 15

[21] Geoffrey Blake, Ronald Dreslinski, and Trevor Mudge. **A survey of multicore processors**. *IEEE Signal Processing Magazine*, **26**(6):26–37, 2009. 18

[22] Eduardo Romera, Luis M. Bergasa, and Roberto Arroyo. **A real-time multi-scale vehicle detection and tracking approach for smartphones**. In *Intelligent Transportation Systems (ITSC), 2015 IEEE 18th International Conference on*, pages 1298–1303. IEEE, 2015. 18

[23] R. Ach, N. Luth, and A. Techmer. **Real-time detection of traffic signs on a multi-core processor**. In *Intelligent Vehicles Symposium, 2008 IEEE*, pages 307–312. IEEE, 2008. 18

[24] Yongpan Liu, Yihao Zhu, Sunny Zhang, Senjie Zhang, and Huazhong Yang. **Acceleration of pedestrian detection algorithms on a multi-core vehicle computing platform**. In *Information, Computing and Telecommunication, 2009. YC-ICT'09. IEEE Youth Conference on*, pages 208–211. IEEE, 2009. 18

[25] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips. **GPU Computing**. *Proceedings of the IEEE*, **96**(5):879–899, 2008. 18

[26] Cristóbal A. Navarro, Nancy Hitschfeld-Kahler, and Luis Mateu. **A Survey on Parallel Computing and its Applications in Data-Parallel Problems Using GPU Architectures**. *Communications in Computational Physics*, **15**(2):285–329, 2014. 19

[27] *"BMW adopts Nvidia GPU for in-car displays. [online]"*. https://www.cnet.com/news/bmw-adopts-nvidia-gpu-for-in-car-displays/. 19

[28] *"NVIDIA Drive PX for Autonomous Driving. [online]"*. http://www.nvidia.com/object/drive-px.html. 19

[29] *"NVIDIA Supercomputing Technology into the Car with Tegra K1. [online]"*. http://www.nvidia.in/object/tegra-k1-car-advanced-driver-assist-jan5-2014-in.html. 19

[30] V. Campmany, S. Silva, A. Espinosa, J.C. Moure, D. Vázquez, and A.M. López. **GPU-based Pedestrian Detection for Autonomous Driving**. *Procedia Computer Science*, **80**:2377–2381, 2016. International Conference on Computational Science 2016. 19

[31] Kristian Kovačić, Edouard Ivanjko, and Hrvoje Gold. **Real time vehicle detection and tracking on multiple lanes**. *WSCG Conference on Computer Graphics, Visualization and Computer Vision*, 2014. 19

[32] Romain Saussard, Boubker Bouzid, Marius Vasiliu, and Roger Reynaud. **The embeddability of lane detection algorithms on heterogeneous architectures**. In *ICIP*, pages 4694–4697. IEEE, 2015. 19

[33] Tam Phuong Cao, Guang Deng, and David Mulligan. **Implementation of real-time pedestrian detection on FPGA**. In *Image and Vision Computing New Zealand, 2008. IVCNZ 2008. 23rd International Conference*, pages 1–6. IEEE, 2008. 20

[34] Marek Wójcikowski, Robert Żaglewski, and Bogdan Pankiewicz. **FPGA-Based Real-Time Implementation of Detection Algorithm for Automatic Traffic Surveillance Sensor Network**. *Journal of Signal Processing Systems*, **68**(1):1–18, 2012. 20

[35] Erke Shang, Jian Li, Xiangjing An, and Hangen He. **A real-time lane departure warning system based on FPGA**. In *Intelligent Transportation Systems (ITSC), 2011 14th International IEEE Conference on*, pages 1243–1248. IEEE, 2011. 20

[36] Mehdi Darouich, Stéphane Guyetant, and Dominique Lavenier. **A Reconfigurable Disparity Engine for Stereovision in Advanced Driver Assistance Systems**. In *Reconfigurable Computing: Architectures, Tools and Applications, 6th International Symposium, ARC 2010, Bangkok, Thailand, March 17-19, 2010. Proceedings*, pages 306–317. 20

[37] M. Mielke, A. Schäfer, and R. Brück. **ASIC implementation of a Gaussian Pyramid for use in autonomous mobile robotics**. In *2011 IEEE 54th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 1–4, 2011. 20

[38] Gideon P. Stein, Elchanan Rushinek, Gaby Hayun, and Amnon Shashua. **A Computer Vision System on a Chip: a case study from the automotive domain**. In *CVPR Workshops*, page 130. IEEE Computer Society, 2005. 20

[39] *"Mobileye: The Evolution of EyeQ. [online]"*. `https://www.mobileye.com/our-technology/evolution-eyeq-chip/`. 20

[40] *"The R-Car SoCs of Renesas. [online]"*. `https://www.renesas.com/en-us/products/automotive-lsis/r-car.html`. 21

[41] Takashi Machida and Takashi Naito. **GPU & CPU cooperative accelerated pedestrian and vehicle detection**. In *Computer Vision Workshops (ICCV Workshops), 2011 IEEE International Conference on*, pages 506–513. IEEE, 2011. 21

[42] Pei-Yung Hsiao, Chun-Wei Yeh, Shih-Shinh Huang, and Li-Chen Fu. **A Portable Vision-Based Real-Time Lane Departure Warning System: Day and Night**. *IEEE Transactions on Vehicular Technology*, **58**:2089–2094, 2009. 21

[43] *"The OpenMP API Specification for Parallel Programming. [online]"*. http://www.openmp.org/specifications/. 23

[44] *"Intel OpenCL sdk. [online]"*. https://software.intel.com/en-us/intel-opencl. 30

[45] *"AMD OpenCL sdk. [online]"*. http://developer.amd.com/tools-and-sdks/opencl-zone/. 30

[46] *"NVIDIA OpenCL sdk. [online]"*. https://developer.nvidia.com/opencl. 30

[47] *"Arm-mali OpenCL sdk. [online]"*. https://developer.arm.com/products/software/mali-sdks. 30

[48] Jia-Jhe Li, Chi-Bang Kuan, Tung-Yu Wu, and Jenq Kuen Lee. **Enabling an OpenCL Compiler for Embedded Multicore DSP Systems**. pages 545–552. IEEE, 2012. 30

[49] Wendell F. S. Diniz, Vincent Frémont, Isabelle Fantoni, and Eurípedes G. O. Nóbrega. **An FPGA-based architecture for embedded systems performance acceleration applied to Optimum-Path Forest classifier**. *Microprocessors and Microsystems - Embedded Hardware Design*, **52**:261–271, 2017. 30

[50] Jonathan Tompson and Kristofer Schlachter. **An introduction to the opencl programming model**. *Person Education*, **49**, 2012. 31

[51] Clément Farabet, Berin Martini, Benoit Corda, Polina Akselrod, Eugenio Culurciello, and Yann LeCun. **Neuflow: A runtime reconfigurable dataflow processor for vision**. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 109–116, 2011. 31

[52] Farouk Mansouri, Sylvain Huet, and Dominique Houzet. **A domain-specific high-level programming model**. *Concurrency and Computation: Practice and Experience*, **28**(3):750–767, 2016. 31, 32

[53] Farouk Mansouri, Sylvain Huet, and Dominique Houzet. **SignalPU: A Programming Model for DSP Applications on Parallel and Heterogeneous Clusters**. In *2014 IEEE International Conference on High Performance Computing and Communications, 6th IEEE International Symposium on Cyberspace Safety and Security, 11th IEEE International Conference on Embedded Software and Systems, HPCC/CSS/ICESS 2014, Paris, France, August 20-22, 2014*, pages 937–944, 2014. 32

[54] Cédric Augonnet, Samuel Thibault, and Raymond Namyst. *StarPU: a runtime system for scheduling tasks over accelerator-based multicore machines.* PhD thesis, INRIA, 2010. 32

[55] Erik Rainey, Jesse Villarreal, Goksel Dedeoglu, Kari Pulli, Thierry Lepley, and Frank Brill. **Addressing System-Level Optimization with OpenVX Graphs**. In *IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 658–663, 2014. 32, 138, 139, 142, 155, 167

[56] Pierre Esterie, Joel Falcou, Mathias Gaunard, Jean-Thierry Lapresté, and Lionel Lacassagne. **The Numerical Template toolbox: A Modern C++ Design for Scientific Computing**. *Journal of Parallel and Distributed Computing*, **74**(12):pp. 3240–3253, July 2014. 32, 34

[57] Pierre Esterie, Joel Falcou, Mathias Gaunard, and Jean-Thierry Lapresté. **Boost.SIMD: generic programming for portable SIMDization**. In *Proceedings of the 2014 Workshop on Programming models for SIMD/Vector processing, WPMVP 2014, Orlando, Florida, USA, February 16, 2014*, pages 1–8, 2014. 34, 35, 92

[58] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. 34

[59] Eric Niebler. **Proto: A Compiler Construction Toolkit for DSELs**. In *Proceedings of the 2007 Symposium on Library-Centric Software Design*, LCSD '07, pages 42–51. ACM, 2007. 34

[60] Krzysztof Czarnecki and Ulrich W. Eisenecker. **Components and Generative Programming (Invited Paper)**. *SIGSOFT Softw. Eng. Notes*, **24**(6):2–19, October 1999. 34

[61] Bihao Wang, Sergio Alberto Rodriguez Florez, and Vincent Frémont. **Multiple Obstacle Detection and Tracking using Stereo Vision: Application and Analysis**. In *13th International Conference on Control, Automation, Robotics & Vision (ICARCV)*, Singapour, Singapore, December 2014. 39, 40

[62] Qian Yu, Helder Araújo, and Hong Wang. **A Stereovision Method for Obstacle Detection and Tracking in Non-Flat Urban Environments**. *Autonomous Robots*, **19**(2):141–157, 2005. 40

[63] Raphael Labayrade and Didier Aubert. **In-vehicle obstacles detection and characterization by stereovision**. In *Proceedings of the 1st International Workshop on In-Vehicle Cognitive Computer Vision Systems, Graz, Austria*, 2003. 40

[64] Yan Jiang, Feng Gao, and Guoyan Xu. **Computer vision-based multiple-lane detection on straight road and in a curve**. In *International Conference on Image Analysis and Signal Processing (IASP)*, pages 114–117. IEEE, 2010. 40

[65] Hiroshi Iwata and Keiji Saneyoshi. **Forward obstacle detection in a lane by stereo vision**. In *Industrial Electronics Society, IECON 39th Annual Conference*, pages 2420–2425. IEEE, 2013. 40

[66] Waseem Khan and Reinhard Klette. **Stereo accuracy for collision avoidance for varying collision trajectories**. In *Intelligent Vehicles Symposium (IV)*, pages 1259–1264. IEEE, 2013. 40

[67] Yuan Gao, Xiao Ai, John Rarity, and Naim Dahnoun. **Obstacle detection with 3D camera using UV-Disparity**. In *7th International Workshop on Systems, Signal Processing and their Applications (WOSSPA)*, pages 239–242. IEEE, 2011. 40, 41

[68] Clemens Rabe, Uwe Franke, and Stefan Gehrig. **Fast detection of moving objects in complex scenarios**. In *Intelligent Vehicles Symposium, 2007*, pages 398–403. IEEE, 2007. 40

[69] Ion Giosan and Sergiu Nedevschi. **Superpixel-based obstacle segmentation from dense stereo urban traffic scenarios using intensity, depth and optical flow information**. In *17th International Conference on Intelligent Transportation Systems (ITSC), 2014*, pages 1662–1668. IEEE, 2014. 40

[70] Cosmin D. Pantilie, Silviu Bota, Istvan Haller, and Sergiu Nedevschi. **Real-time obstacle detection using dense stereo vision and dense optical flow**. In *International Conference on Intelligent Computer Communication and Processing (ICCP), 2010*, pages 191–196. IEEE, 2010. 40

[71] Mathias Perrollaz, John-David Yoder, Amaury Negre, Anne Spalanzani, and Christian Laugier. **A Visibility-Based Approach for Occupancy Grid Computation in Disparity Space**. *IEEE Transactions on Intelligent Transportation Systems*, **13**(3):1383–1393, September 2012. 40

[72] Raphael Labayrade, Didier Aubert, and Jean-Philippe Tarel. **Real time obstacle detection in stereovision on non flat road geometry through" v-disparity" representation**. In *Intelligent Vehicle Symposium*, **2**, pages 646–651. IEEE, 2002. 40, 53

[73] Richard O. Duda and Peter E. Hart. **Use of the Hough transformation to detect lines and curves in pictures**. *Communications of the ACM*, **15**(1):11–15, 1972. 40, 54

[74] M. Perrollaz, A. Spalanzani, and D. Aubert. **Probabilistic representation of the uncertainty of stereo-vision and application to obstacle detection**. In *Intelligent Vehicles Symposium (IV)*, pages 313–318, June 2010. 41

[75] Kai Zhu, Jiangxiang Li, and Haofeng Zhang. **Stereo vision based road scene segment and vehicle detection**. In *2nd International Conference on Information Technology and Electronic Commerce (ICITEC)*, pages 152–156, December 2014. 41

[76] Min Zhang, Peizhi Liu, Xiaochuan Zhao, Xinxin Zhao, and Yuan Zhang. **An obstacle detection algorithm based on UV disparity map analysis**. In *International Conference onInformation Theory and Information Security (ICITIS)*, pages 763–766. IEEE, 2010. 41

[77] *"Epipolar Geometry"*. https://en.wikipedia.org/wiki/Epipolar_geometry. 42

[78] Daniel Scharstein and Richard Szeliski. **A Taxonomy and Evaluation of Dense Two-Frame Stereo Correspondence Algorithms**. *Int. J. Comput. Vision*, **47**(1-3):7–42, April 2002. 43, 44

[79] David G. Lowe. **Distinctive Image Features from Scale-Invariant Keypoints**. *International Journal of Computer Vision*, **60**(2):91–110, 2004. 44

[80] Herbert Bay, Andreas Ess, Tinne Tuytelaars, and Luc Van Gool. **Speeded-Up Robust Features (SURF)**. *Computer Vision and Image Understanding*, **110**(3):346 – 359, 2008. Similarity Matching in Computer Vision and Multimedia. 44, 49

[81] Masatoshi Okutomi and Takeo Kanade. **A Locally Adaptive Window for Signal Matching**. *Int. J. Comput. Vision*, **7**(2):143–162, January 1992. 45

[82] Hai Tao, Harpreet S. Sawhney, and Rakesh Kumar. **A Global Matching Framework for Stereo Computation.** pages 532–539, 2001. 45

[83] Aaron F. Bobick and Stephen S. Intille. **Large Occlusion Stereo**. *International Journal of Computer Vision*, **33**:181–200, 1999. 45, 46, 49

[84] Takeo Kanade, H. Kato, S. Kimura, A. Yoshida, and K. Oda. **Development of a Video-Rate Stereo Machine**. **3**:95–100, 1995. 45, 46

[85] Yong Seok Heo, Kyong Mu Lee, and Sang Uk Lee. **Robust Stereo Matching Using Adaptive Normalized Cross-Correlation**. *IEEE Trans. Pattern Anal. Mach. Intell.*, **33**(4):807–822, 2011. 45

[86] Julien Lerouge, Maroua Hammami, Pierre Héroux, and Sébastien Adam. **Minimum cost subgraph matching using a binary linear program**. *Pattern Recognition Letters*, **71**:45 – 51, 2016. 45

[87] Sing Bing Kang, Richard Szeliski, and Jinxiang Chai. **Handling Occlusions in Dense Multi-view Stereo**. In *2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2001)*, pages 103–110, 2001. 46

[88] Olga Veksler. **Stereo Correspondence with Compact Windows via Minimum Ratio Cycle**. *IEEE Trans. Pattern Anal. Mach. Intell.*, **24**(12):1654–1660, December 2002. 46

[89] Pedro F. Felzenszwalb and Daniel P. Huttenlocher. **Efficient Belief Propagation for Early Vision**. *Int. J. Comput. Vision*, **70**(1):41–54, October 2006. 46

[90] Jian Sun, Nan-Ning Zheng, and Heung-Yeung Shum. **Stereo Matching Using Belief Propagation**. *IEEE Trans. Pattern Anal. Mach. Intell.*, **25**(7):787–800, July 2003. 46

[91] Jae-Chul Kim, Kyoung Mu Lee, Byoung-Tae Choi, and Sang Uk Lee. **A Dense Stereo Matching Using Two-Pass Dynamic Programming with Generalized Ground Control Points**. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2005), 20-26 June 2005, San Diego, CA, USA*, pages 1075–1082, 2005. 46

[92] V Kolmogorov and R Zabih. **Computing visual correspondence with occlusions using graph cuts**. In *IEEE ICCV*, pages 508—-515, 2001. 46

[93] Stan Birchfield and Carlo Tomasi. **A Pixel Dissimilarity Measure That Is Insensitive to Image Sampling**. *IEEE Trans. Pattern Anal. Mach. Intell.*, **20**(4):401–406, April 1998. 47

[94] Andreas Geiger, Martin Roser, and Raquel Urtasun. **Efficient large-scale stereo matching**. In *Computer Vision–ACCV*, pages 25–38. Springer, 2011. 47, 48, 49, 52, 65, 72

[95] Sven Wanner and Bastian Goldluecke. **Globally consistent depth labeling of 4D light fields**. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 41–48. IEEE, 2012. 48

[96] Sayanan Sivaraman and Mohan Manubhai Trivedi. **Looking at Vehicles on the Road: A Survey of Vision-Based Vehicle Detection, Tracking, and Behavior Analysis**. *IEEE Transactions on Intelligent Transportation Systems*, **14**(4):1773–1795, 2013. 48

[97] Changil Kim, Henning Zimmer, Yael Pritch, Alexander Sorkine-Hornung, and Markus H. Gross. **Scene reconstruction from high spatio-angular resolution light fields.** *ACM Trans. Graph.*, **32**(4):73–1, 2013. 48

[98] Daniel Scharstein, Heiko Hirschmüller, York Kitajima, Greg Krathwohl, Nera Nešić, Xi Wang, and Porter Westling. **High-resolution stereo datasets with subpixel-accurate ground truth**. In *German Conference on Pattern Recognition*, pages 31–42. Springer, 2014. 48

[99] Xiaoyan Hu and Philippos Mordohai. **Evaluation of stereo confidence indoors and outdoors.** In *CVPR*, pages 1466–1473. IEEE Computer Society, 2010. 49

[100] Radim Sára. **Finding the Largest Unambiguous Component of Stereo Matching**. In *Proceedings of the 7th European Conference on Computer Vision-Part III*, ECCV '02, pages 900–914. Springer-Verlag, 2002. 49

[101] Andreas Geiger, Philip Lez, Christoph Stiller and Raquel Urtasun. *"KITTI Vision Benchmark Suite"*. `http://www.cvlibs.net/datasets/kitti/`. 59, 71, 110, 149

[102] *"The KITTI Vision Benchmark Suite. [online]"*. `http://www.cvlibs.net/datasets/kitti/setup.php`. 59

[103] A Geiger, P Lenz, C Stiller, and R Urtasun. **Vision Meets Robotics: The KITTI Dataset**. *Int. J. Rob. Res.*, **32**(11):1231–1237, September 2013. 59

[104] Zehang Sun, G. Bebis, and R. Miller. **Monocular precrash vehicle detection: features and classifiers**. *IEEE Transactions on Image Processing*, **15**(7):2019–2034, July 2006. 62

[105] Daniel Alonso, Luis Salgado, and Marcos Nieto. **Robust vehicle detection through multidimensional classification for on board video based systems**. In *IEEE International Conference on Image Processing.*, **4**, pages IV–321. IEEE, 2007. 62

[106] Peter Bergmiller, Mario Botsch, Johannes Speth, and Ulrich Hofmann. **Vehicle rear detection in images with generalized radial-basis-function classifiers**. In *Intelligent Vehicles Symposium.*, pages 226–233. IEEE, 2008. 62

[107] Ben Southall, Mayank Bansal, and Jayan Eledath. **Real-time vehicle detection for highway driving**. In *IEEE Conference on Computer Vision and Pattern Recognition.*, pages 541–548. IEEE, 2009. 62

[108] Wen-Chung Chang and Chih-Wei Cho. **Online Boosting for Vehicle Detection**. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, **40**(3):892–902, June 2010. 62

[109] T. Kowsari, S. S. Beauchemin, and J. Cho. **Real-time vehicle detection and tracking using stereo vision and multi-view AdaBoost**. In *14th International Conference on Intelligent Transportation Systems (ITSC).*, pages 1255–1260. IEEE, 2011. 62

[110] *"Intel Intrinsics Guide. [online]"*. `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`. 102

[111] *"OpenCL .[online]"*. `https://www.khronos.org/opencl/`. 104

[112] Mark Jason Harris. *Real-time Cloud Simulation and Rendering*. PhD thesis, 2003. AAI3112020. 105

[113] *"Compute memory bandwidth with CUDA.[online]"*. https://devblogs.nvidia.com/parallelforall/how-implement-performance-metrics-cuda-cc/. 106

[114] *"CUDA C Best Practices Guide. [online]"*. http://docs.nvidia.com/cuda/cuda-c-best-practices-guide. 107

[115] *"Nvidia CUDA Toolkit Documentation. [online]"*. http://docs.nvidia.com/cuda/profiler-users-guide/index.html. 108, 112

[116] *"accULL, the OpenACC research implementation.[online]"*. https://accull.wordpress.com/. 110

[117] *"Omni compiler project.[online]"*. http://omni-compiler.org/. 110

[118] *"IPMACC, Open Source OpenACC to CUDA/OpenCL Translator.[online]"*. https://arxiv.org/abs/1412.1127. 110

[119] J. A. Herdman, W. P. Gaudin, S. McIntosh-Smith, M. Boulton, D. A. Beckingsale, A. C. Mallinson, and S. A. Jarvis. **Accelerating Hydrocodes with OpenACC, OpeCL and CUDA**. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, SCC '12, pages 465–471, Washington, DC, USA, 2012. IEEE Computer Society. 135

[120] 135

[121] Bui Tat Minh, Michael Förster, and Uwe Naumann. **Towards Tangent-linear GPU Programs Using OpenACC**. In *Proceedings of the Fourth Symposium on Information and Communication Technology*, SoICT '13, pages 27–34, New York, NY, USA, 2013. ACM. 135

[122] Giuseppe Tagliavini, Germain Haugou, Andrea Marongiu, and Luca Benini. **ADRENALINE: an OpenVX environment to optimize embedded vision applications on many-core accelerators**. In *IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, pages 289–296, 2015. 138

[123] Pierre Estérie, Joel Falcou, Mathias Gaunard, and Jean-Thierry Lapresté. **Boost. simd: generic programming for portable simdization**. In *Proceedings of the 2014 Workshop on Programming models for SIMD/Vector processing*, pages 1–8. ACM, 2014. 144

[124] *"ARM Limited, Introducing NEON - Development Article, 2009. [online]"*. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dht0002a/index.html. 145

[125] Gaurav Mitra, Beau Johnston, Alistair P. Rendell, Eric McCreath, and Jun Zhou. **Use of SIMD vector operations to accelerate application code performance on low-powered ARM and Intel platforms**. In *27th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, pages 1107–1116. IEEE, 2013. 145

[126] Huayong Wang, Henrique Andrade, Buğra Gedik, and Kun-Lung Wu. **A Code Generation Approach for Auto-vectorization in the SPADE Compiler**. In *Proceedings of the 22Nd International Conference on Languages and Compilers for Parallel Computing*, LCPC'09, pages 383–390. Springer-Verlag, 2010. 145

[127] Dorit Nuzman, Sergei Dyshel, Erven Rohou, Ira Rosen, Kevin Williams, David Yuste, Albert Cohen, and Ayal Zaks. **Vapor SIMD: Auto-vectorize Once, Run Everywhere**. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 151–160. IEEE Computer Society, 2011. 145

[128] Jaewook Shin. **Introducing Control Flow into Vectorized Code**. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07. IEEE Computer Society, 2007. 145

[129] *"Intel. Math kernel library MKL. [online]"*. https://software.intel.com/en-us/mkl. 145

[130] Khronos Group. *"The OpenVX API for hardware acceleration. [online]"*. https://www.khronos.org/openvx, 2013. 147

[131] Arun K. Somani Craig M. Wittenbrink. *"Cache tiling for high performance morphological image processing. [online]"*. https://www.researchgate.net/publication/225270282_Cache_tiling_for_high_performance_morphological_image_processing. 155

[132] Raphael Labayrade, Didier Aubert, and Jean-Philippe Tarel. **Real time obstacle detection in stereovision on non flat road geometry through " v-disparity" representation**. In *Intelligent Vehicle Symposium*, **2**, pages 646–651. IEEE, 2002. 159

[133] Pierre Esterie, Mathias Gaunard, Joel Falcou, Jean-Thierry Lapresté, and Brigitte Rozoy. **Boost.SIMD: generic programming for portable SIMDization.** In Pen-Chung Yew, Sangyeun Cho, Luiz DeRose, and David J. Lilja, editors, *PACT*, pages 431–432. ACM, 2012. 161

[134] *"HIPAcc, the Heterogeneous Image Processing Acceleration Framework. [online]"*. http://hipacc-lang.org/. 173

[135] Daniel Hernández Juárez, Alejandro Chacón, Antonio Espinosa, David Vázquez, Juan Carlos Moure, and Antonio Manuel López Peña. **Embedded real-time stereo estimation via Semi-Global Matching on the GPU**. *CoRR*, **abs/1610.04121**, 2016. 174

[136] Samuel Williams, Andrew Waterman, and David Patterson. **Roofline: An Insightful Visual Performance Model for Multicore Architectures**. *Commun. ACM*, **52**(4):65–76, April 2009. 174

[137] *"STREAM Benchmark. [online]"*. https://www.cs.virginia.edu/stream/. 174