High-speed parallel processing on CUDA-enabled Graphics Processing Units

---

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Johan A. Huisman
born in Sliedrecht, The Netherlands

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

High-speed parallel processing on CUDA-enabled Graphics Processing Units

---

by Johan Huisman

Abstract

A new trend in computing is the use of multi-core processors and the use of Graphics Processing Units (GPUs) for general purpose high speed parallel computing. Therefore TNO wants to respond to the current trend of parallel computing and wants to investigate on which architectures they want to apply and parallelize their applications on. This thesis discusses the implementation of two applications used by TNO onto a CUDA-enabled GPU and a multi-core processor. The applications chosen for the implementation are object detection and ultrasound simulation (UMASIS). the development time and performance gain were measured during the implementation of both case. With these measurements an estimation of future projects can be made.

Laboratory : Computer Engineering
Codenumber : CE-MS-2010-05

Committee Members:

Advisor:	Dr. Ir. Ben H.H. Juurlink, Associate Professor, CE, TU Delft
Advisor:	Dr. Ir. Arjan J. van Genderen, Assistant Professor, CE, TU Delft
Advisor:	Ir. Maurits S. van der Heiden, Senior Research Scientist, TNO I&T Delft
Chairperson:	Dr. Ir. Koen L.M. Bertels, Associate Professor, CE, TU Delft
Member:	Ir. Maurits S. van der Heiden, Senior Research Scientist, TNO I&T Delft
Member:	Dr. Ir. Stephan Wong, Assistant Professor, CE, TU Delft
Member:	Dr. Ir. Charles Botha, Assistant Professor, Mediamatics, TU Delft

Acknowledgement

First of all, I am grateful to Nelleke Huisman-van den Berg, my loving wife, and Arie Huisman, my father, who greatly helped, supported and encouraged me throughout my study.
I want to thank TNO Industrie & Techniek and the Computer Engineering Department of the Delft University of Technology for giving me the opportunity to commence this thesis.
My special thanks go to my supervisor at TNO, Maurits van der Heiden, for his patience, time and help. Special thanks also to Ben Juurlink and Arjan van Genderen for their advice, support, and encouragement during the MSc project.

Johan Huisman
Delft, The Netherlands
June 7, 2010

Table of Contents

List of Figures

List of Tables

List of Acronyms

CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DMA	Direct Memory Access
FLOPS	FLoating Operations Per Second
FPGA	Field Programmable Gate Array
FDTD	Finite Difference Time Domain
GPU	Graphics Processing Unit
HPC	High Performance Computing
OpenCL	Open Computing Language
OpenCV	Open Computer Vision
PC	Personal Computer
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Thread
UMASIS	Ultrasonic Modeling and AnalySIS

Chapter 1: Introduction

In early days computing was done on centralized computer systems, called mainframes. These mainframes were large and powerful systems on which all of the computations for a group of users were performed. Executions of the calculations were scheduled in time. Later on mainframes evolved to systems able to execute multiple concurrent computations at a time. It was till that time when personal computers (PCs) where introduced that mainframes became less popular. Mainframes demanded large space and a high energy consumption. Despite that PCs were less powerful than mainframes, but users had their own computer and could execute programs on demand. This instead of having to wait until their programs was executed by the scheduler of the mainframe. After this PCs, were getting more powerful each generation due to smaller production factors resulting in higher clock frequencies.

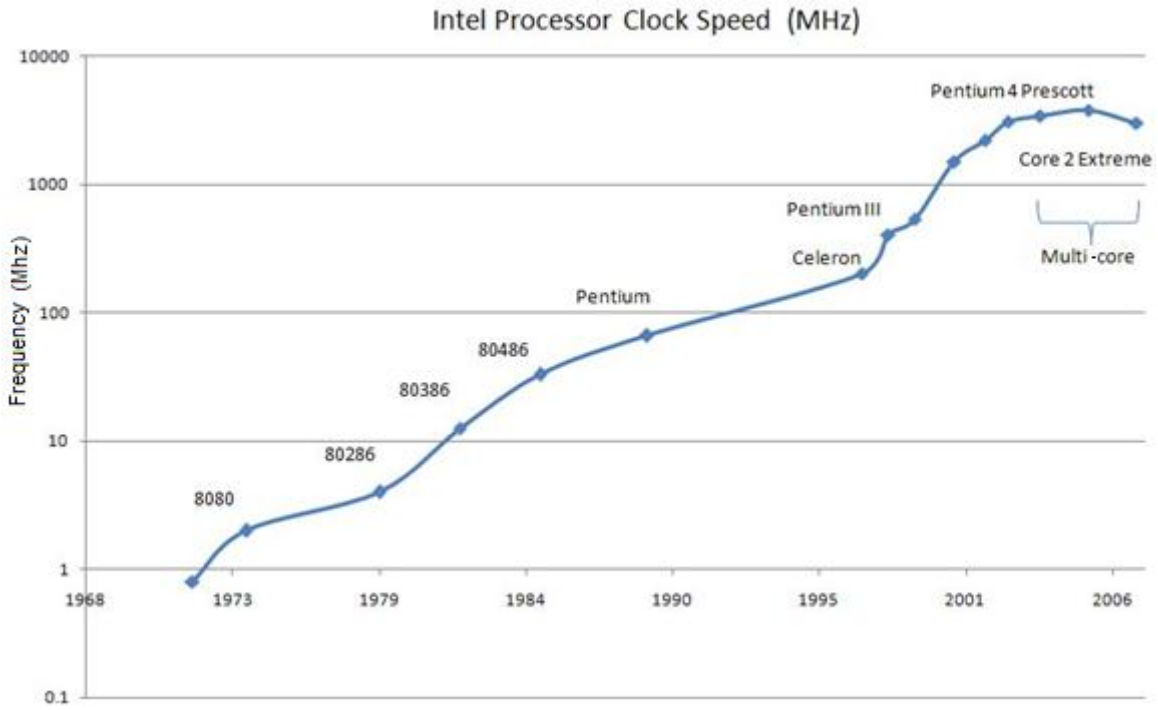

Figure 1. Clock frequency increase and later stagnation over the years [38].

Recent developments in computer industry show an upper limit in core frequency and reveal a trend in parallel approach of computing instead of serial to overcome this limit. Modern processors are composed of multiple processing elements (Multi-core) instead of one more powerful processing unit compared to the previous (Figure 1).

Making full benefits, programs should be optimized to make use of all available processing elements in a processor. Therefore older programs should be optimized for multi-core processors by rewriting processing intensive parts. The time that a speedup could be realized by replacing the old hardware for a new faster one seems to be over. That is why parallelization of the code is

needed to make use of future computing power. In general this task is not trivial and requires a certain amount of knowledge of the processor architecture and parallelization techniques.

Highly parallel processing is extensively used in the gaming and graphical computing segment on specialized hardware. This hardware, called graphics processing units (GPUs) [1], are many-core processors build for graphics acceleration purposes. Recently the GPUs are enabled to execute general purpose tasks by means of the CUDA architecture [2]. Compared to today's multi-core processors, having typically up to four processing elements, these GPUs are providing up to hundreds of processing elements. So parallel execution could be exploited very well on these kind of computing architectures.

1.1 Motivation

This project was conducted at TNO. Inside TNO I&T there are multiple high performance computing (HPC) applications. These applications require high computing power to achieve an acceptable performance. Some of these applications are written for single thread execution. We must therefore respond to the current trend of parallel computing and have to investigate on which architectures we want to apply and parallelize our applications on.

The interest in applying alternative architecture is driven by the specifications that architectures like the IBM Cell B.E. [3] and GPUs in theory can offer. Important specifications like memory bandwidth and massively parallel computing power are making these architectures very promising. For example, the memory bandwidth of 25 GB/s for the Cell/B.E. and more than 100 GB/s for the GPU is much higher than the ±8GB/s for current CPUs. Besides that, the Cell/B.E could theoretically compute 250 GFlops where the GPU could even compute up to 1 TFlops. These specifications are impressive in comparison to a single core processor that can deliver approximate 15 GFlops. Even small clusters, like the Cray XD1 which can deliver per rack about 0.6 TFlops [4], are performing in comparison, in terms of size, prize and energy, a lot less than these new architectures. These new architectures are currently available for a reasonable price. However, development of HPC applications on these new architectures is relatively new and complex and have limited number of programming tools. Nevertheless which platform is chosen to run the HPC applications, parallelization of the code is necessary. Since almost all modern computing architectures use parallelism to improve their performance.

1.2 Research Goals

The research goals of the project are defined using the following criteria with respect to each computing platform and application.

- **Parallelization of code to a CUDA-enabled GPU:** What steps are required to perform a successful implementation on a CUDA-enabled GPU? What are the difficulties and limitations?
- **Efficiency:** What can be achieved in theory and what is gained in performance in practice?
- **Costs:** How are the hardware and development cost in comparison with the performance achieved from the implementations?
- **Maturity of CUDA:** Is it profitable to invest in further development of CUDA-enabled applications? Is there enough ground and assurance that current CUDA applications will be future proof. How has CUDA developed thus far and can predictions for the future be made? All these questions are related to the question of the CUDA maturity.

A selection has been made of computing platforms on which the applications, used and developed within TNO, can run. These computing platforms are selected from available modern computing architectures. The computing architectures that will be used to perform our applications are:

- Single-core (reference)
- Multi-core CPU (OpenMP)
- GPU (CUDA)

Two use-cases are defined from a selection of HPC applications developed and used inside TNO. They should be made able to run on all the selected computing platforms. The criterion for the selection of applications is that they are used within different fields and that they make use of diverse algorithms.

With this criterion the following applications were selected:

- Object detection (Computer Vision)
- Ultrasonic finite difference modelling (UMASIS)

The first application is object detection, used within computer vision. In this application an algorithm based on the so called '*haar features*' is used for detection of objects in a computer image [5]. This by making use of a decision tree whereby many small calculations are performed to achieve a match in the image. Due to the decision tree this algorithm alternates much in runtime between calculations.

UMASIS is an application that simulates ultrasonic pulse measurements to perform inspections on materials. The algorithm relies much on Finite Difference Time Domain (FDTD) calculations which are very computational intensive and generate lots of data, requiring in some cases multiple days of computation. At the moment UMASIS is executed task parallel on a Cray XD1 supercomputer. The Cray XD1 runs multiple simulation jobs concurrently.

These two cases are both already operational but are written as single core/thread applications.

1.3 Analysis strategy

The analysis strategy can be broken up in five phases:

- Define target platforms.
- Define realistic cases.
- Profile the individual case.
- Implement and realize the cases on the target platforms.
- Benchmark and analysis of the results.

Within these phases multiple aspects are being worked out.

Profiling individual cases:

- What are the most time consuming parts?
- What schemes and algorithms are used
- Is it parallelizable?
- What is the maximum speed up that can be realized theoretically?
- How do we map the algorithm on the specific target?

Analysis of the results:

- What is the realized speedup on each target platform?
- What is the effort taken to realize the speedup?
- What can we conclude from these results?
- What goes wrong and what went very well?
- Which further improvements are possible in the near future?

1.4 Contributions

The contributions of this thesis are:

- **Implementation of object detection on a CUDA-enabled GPU:** The application of object detection demand high-speed computing to get real-time performance. By implementing the object detection on a CUDA-enabled GPU this can be achieved.
- **Implementation of UMASIS on a CUDA-enabled GPU:** The UMASIS application is very time consuming.. To reduce runtime the UMASIS application is implemented on a CUDA-enabled GPU.
- **Analysis of the development process for CUDA based on two implementations:** The development time and performance gain are recorded during the implementation of both use cases. With this information performance and time to develop estimations can be done for future projects. To ease this process some important factors related to success rate of the implementation are needed to be identified.

1.5 Related Work

1.5.1 CUDA

The experiences on applying the NVIDIA CUDA programming model to acceleration applications are still recent and the number of publications is small but is growing rapidly. Nevertheless there are a couple of publications on experiences of academic research and industrial products which have used CUDA to achieve significant parallel speedup. The applications fall into a variety of application domains including machine learning [6], database processing [7], bioinformatics [8,9], numerical linear algebra[10,11], medical imaging[12], and physical simulation[13,14].

From these individual experiences the following can be concluded:

- CUDA provides a straightforward means of describing inherently parallel computations particularly for data-parallel algorithms.
- NVIDIA GPU architectures deliver high computational throughput.
- CUDA-enabled GPUs are relatively small, less expensive and highly available compared to systems delivering comparable performance.

However they found that performance can only be realized when the specific architecture is taken in account obeying the following rules:

- **Realize a sufficient amount of fine-grained parallelism:** This by splitting the code in many small subtasks.
- **Make use of blocking computations to fit CUDA thread block abstraction:** Threads are defined in blocks, by dividing the 'problem' over huge amount of small blocks the GPU is used as efficient as possible.
- **Make data-parallel programs efficient, threads of a warp follow same execution path:** Set the right amount of threads per block and avoid the use of too much registers and branches. A good analysis tool for this is the CUDA occupancy calculator.
- **Make full use of the high-speed and low latency per-block shared memory:** Use as much as possible the fast registers and shared memory and avoid overuse of the global memory.

David Luebke gives a more in-depth coverage of the CUDA programming model [15]. Knowledge on the functional concepts is required to make sure that the programmer can make use of the full potential of the GPU.

1.5.2 Image processing on GPUs

GPUs were originally intended to perform graphics related calculations primary for games. As stated earlier, GPUs can now be used to perform other calculations. There are several publications concerning GPUs for image processing tasks. The following papers [16,17,18] are reviewed to examine the application of image processing on GPUs.

1.5.2.1 GPUCV

The application of image processing is very broad and cannot be covered totally on the GPU. One attempt is made to introduce a framework that exploits the GPU processing power to accelerate image processing and computer vision [16]. This framework named GPUCV is an open library

that is similar to OpenCV [19]. The main disadvantage is that the GPUCV provides much less functionality compared to that of OpenCV. Also the library uses the old pixel-shader programming tools and does not use the more advanced CUDA programming tools. Comparing GPUCV and the native OpenCV library results in a speedup of the individual functions ranging between 4x to 18x. The biggest advantage of the framework is in the ease of applying the accelerated functions in existing code written for OpenCV. The user of this library only has to replace the original OpenCV procedures into to the GPUCV equivalents. A disadvantage of the current GPUCV library is that it is not up-to-date anymore due to the lack of CUDA support. Still the results give a good insight in the full potential of GPU processing.

1.5.2.2 Mapping of algorithms

There are several reasons to use GPUs for image processing. For example some reasons are highly parallel floating point computations, increased memory bandwidth and multi-GPU computing. The CUDA programming model can be used to meet these purposes. According to [17] it is important to investigate the mapping of algorithms to GPUs. This to realize more advanced optimization and achieve speedups of 10x or even higher instead of typical speedups of 2x-3x for 'naïve' implementations.

1.5.2.3 Canny edge detection

Canny edge detection is widely used in edge feature detection and then very often in the computer vision domain. That is why a canny edge detector, which makes use the CUDA programming model to accelerate the detection, is created [18]. Most of the code written for this application is meant to optimize memory usage, alignment, and size properties of the GPU architecture. This result in a significant speedup compared to a single-core processor. The results compared to multi-core processors are more moderate because of the use of hysteresis which were not implemented on the GPU.

1.5.3 Comparison between various architecture

It is not easy to make a fair comparison between various architectures. Especially if systems are non-generic and have very different architectures. Still there is a high demand from software developers and system architects to know what the best target system is for their application.

The developers of Simbiosys [20], an application for the bio-industry, report on the choices and findings related to the acceleration of their application. It compares the Cell, GPU and an FPGA where the Cell chosen to be the best architecture for their application. The main downside of the FPGA is the lack of support for floating point operations. GPU and Cell are more similar in hardware architecture compared to the FPGA. Some differences between these architectures are: Cell supports double precision floating point calculations, GPUs use caches where Cell puts complete control in hands of the programmer through direct DMA programming, GPUs use wider register (256-bit vs. 128-bit of Cell), Cells ability to execute two operations in a single cycle and finally the higher throughput of the Cell versus GPU.

A cost comparison between all platforms is made. To compare individual platforms the processing power of 400 general purpose computers were used as reference. Then the required amount of GPUs and Cells were calculated to perform the same processing power. The results

show in 2008 for their application, the Cell as most cost efficient architecture applying PS3 Game Consoles incorporating Cell processors, followed respectively by the GPU and the FPGA.

1.6 Structure of this thesis

The introduction, motivation, research goal, analysis strategy and related work of the project are given in Chapter 1. Background information on the different architectures and the definition of parallelism can be found in Chapter 2. The implementation, realization, optimization and analysis of the two cases can be found in the Chapters 3 and 4. The chapters handle the process of translating the existing serial code to parallel code. In Chapter 5 analysis and findings will be discussed. The conclusion and final recommendation are discussed in Chapter 6.

Chapter 2: Background

In this chapter the background is given for this thesis. In Chapter 2.1 the current and future development in processing are discussed. In Chapter 2.2 and 2.3 the two main architectures, resp. multi-core and NVIDIA CUDA-enabled GPU, are shortly introduced. In Chapter 2.4 levels of parallelism are discussed and samples of parallelization of code are given. Finally in Chapter 2.5 some APIs are discussed and code examples are given.

2.1 Current and future developments in processing

2.1.1 From serial to parallel

For a long time we profited from Moore's law [21] and benefited from an exponential speed increase due to an almost exponential growth of the clock frequency, until a few years ago the maximum clock frequency of modern computers reached a limit of approximately 3Ghz. This is due to power constraints [22] that occur at higher frequencies and the constraints of the maximal speed of light [40]. Several solutions have been introduced to deal with this problem. One of these solutions is integrating multiple independent processors on one chip. Hereby certain parts of the code can be executed concurrently on each core which will theoretically result in a shorter runtime.

2.1.2 Multi-core Architecture Trends

The general trend in processor development is from multi-core to many-core: from dual-, quad-, octo-core chips to ones with tens or even hundreds of cores. In addition, multi-core chips mixed with simultaneous multithreading, memory-on-chip and special-purpose "heterogeneous" cores (Cell/B.E) promise further performance and efficiency gains, especially in processing multimedia, recognition and networking applications. There is also a trend of improving energy efficiency by focusing on performance-per-watt.

2.1.3 General Purpose Graphics Processing Units (GPGPU)

The use of co-processors specialized for certain tasks is not new in the computing history. One of these specialized co-processors is the Graphics Processing Unit (GPU) which has as its main task the computation of computer images and transmitting the computed data to a computer screen. As computing evolved and graphics calculations were transferred from mainly 2D to 3D, the GPUs where evolving from single- to many-core co-processers. It is till recently that GPUs can be used to perform general purpose or non-graphics applications. Big GPU venders such as NVIDIA and AMD/ATI are expanding their drivers to support processing of user generated code, often a C-like language, on their GPUs. Hereby a new platform for numerical calculations to run massively parallel programs is created.

2.2 Multi-core Architecture

A multi-core processor combines two or more independent cores into a single package composed on a single integrated circuit (IC). A dual-core processor contains two cores, and a quad-core four cores. A processor with all cores on a single chip is called a monolithic processor [23]. Each core has optimizations such as superscalar, pipelining and multithreading. A system with N cores is efficient when it can compute N or more threads concurrently. The most common multi-core processors are those used in personal computers, primarily from Intel and AMD. However, the technology is also widely used in other technology areas, especially those of embedded processors and in GPUs.

2.2.1 SIMD

General processor cores are designed to execute each instruction in a sequential way. These instructions perform a single operation on the processor like add, subtract, read or load, etc. After a while extra instruction sets, like MMX and SSE for the Intel x86 or VMX / AltiVec for the IBM PowerPC, were introduced by various processor manufacturers. These instruction sets can handle single instruction multiple data (SIMD). By performing in one instruction multiple operations at a time in parallel, instead of performing this operation in a sequential way, a speedup could be achieved.

2.2.2 Architecture composition

One of the biggest areas for variety in multi-core architecture is the composition and balance of cores themselves. Some architectures apply one core design and repeat it consistently (homogenous), while others apply a mixture of different cores, each optimized for a different role (heterogeneous).

2.2.3 Caches

Caches are primarily used to reduce latencies involved by memory fetches. This by creating small but fast memory spaces close to the processing cores. The cores share the same interconnect to the rest of the system. This can give in some cases issues in providing sustainable memory bandwidth. For example, in case all cores perform a read/write on the memory, congestion on the interconnect can be caused. The interconnect thereby fails to facilitate all the requests at once. To prevent this problem there are multiple levels of caches inside the processor (Figure 2). The core will first request data from the caches before it is addressing the global memory, thereby lowering the amount of communication on the interconnect.

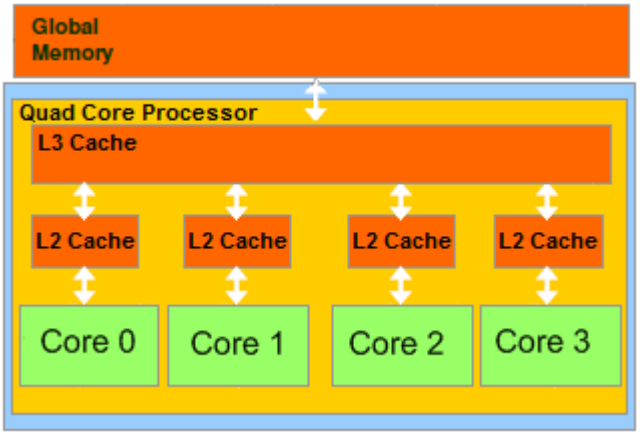

Figure 2. Example of homogeneous multi-core processor

2.3 Definition of parallelism

There are various types of parallelism in computing, where the most important types of parallelism are:

- Task-level parallelism.
- Data-level parallelism.
- Instruction-level parallelism.

2.3.1 Task-level parallelism

Task-level parallelism is achieved in a multiprocessor system when each processor executes a different thread or process on the same or different data. The threads may execute the same or different code. Threads can communicate with each other; this is usually data that is passed from one to another.
An example of task-level parallelism is when there are two tasks and two processors. Then both tasks can run concurrently in parallel, where each processer is assigned to one of the two specific tasks.

2.3.2 Data-level parallelism

Data parallelism is achieved when each processor performs the same task on different pieces of distributed data. In some situations, a single execution thread controls operations on all pieces of data. In others, different threads control the operation, but they execute the same code.

An example of a case where data-level parallelism can be used is the subtraction of two matrices of the size NxN from each other. We can divide the matrix in P parts such that $1 \leq P \leq N^2$ where P is the number of processors. And then let all the processors subtract there part of the matrix and save the result to the corresponding part. In theory, if the bandwidth is unlimited, the matrix subtraction should be in this case P times faster than performed on a single processor.

2.3.3 Instruction-level parallelism

An application consists of a set of instructions which are executed by a processor in order after each other. These instructions can be re-ordered and combined into groups which are then

executed in parallel without changing the result of the program. This principle is called instruction-level parallelism.

Modern processors have multi-stage instruction pipelines. Each stage in the pipeline corresponds to a different action the processor performs on that instruction in that stage. A processor with an N-stage pipeline can have up to N different instructions at different stages. An example of a pipelined processor is a RISC processor that haves five stages: instruction fetch, decode, execute, memory access, and write back.

Some processors can issue more than one instruction at a time. These are known as superscalar processors. Instructions can be grouped together only if there is no data dependency between them.

2.3.4 Dependencies

When implementing parallel algorithms, it is important to understand data dependencies. The basic rule is that a program cannot run quicker than the longest path of concurrent dependent calculations, this is also known as the critical path. Calculations that depend on previous calculations in the path must be executed one after each other. If there is no dependency in algorithm between computations then it is possible to execute the algorithm in independent parts in parallel (Figure 3).

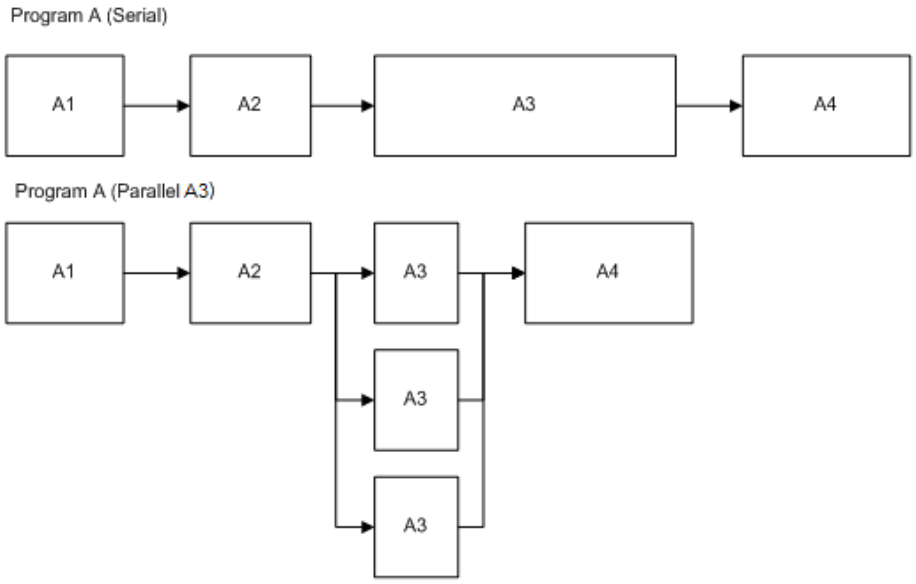

Figure 3. Example of a speedup between serial vs. parallel implementation

2.3.5 Amdahl's law and Gustafson's law

The speedup from parallelism should be theoretically linear, when the number of processors is doubled the processing time should be halved. But in general the speed-up of algorithms in most cases do not approach the theoretical optimum. In general the speed-up is near the optimal for limited amounts of processors and after a certain amount of processors it converges to a constant value. This typical behavior is expressed in Figure 4. This behavior is in most cases the result of

hardware constraints, limiting the maximal performance of the system, for example the global memory I/O speed or the maximum throughput of the peripheral bus.

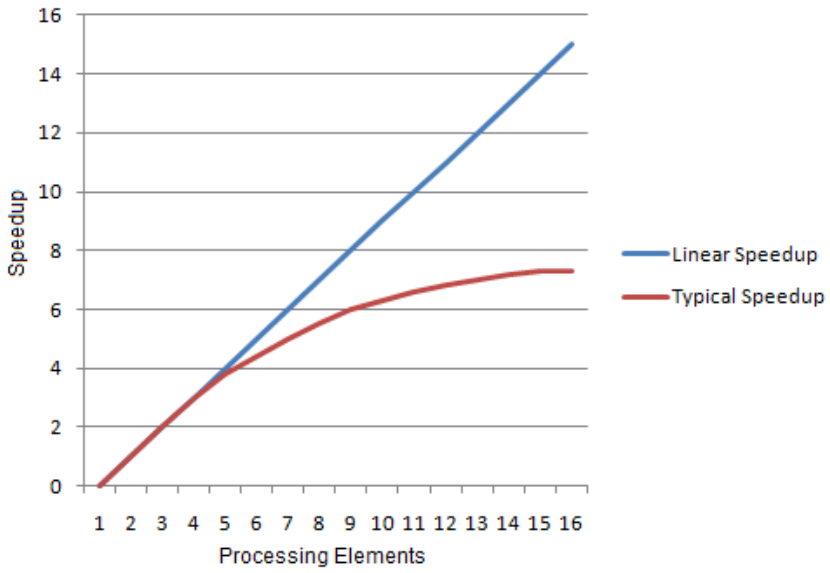

Figure 4. Ideal vs. typical speedup

Amdahl's law [25] can be used to determine the potential speed-up of an algorithm on a parallel computing platform. The law states the portion of the program which cannot be parallelized will limit the overall speed-up available from parallelization. Almost any algorithm consists of parallelizable parts and non-parallelizable parts. This relationship is given by the equation:

$$\boldsymbol{S = \frac{1}{(1-P)-\frac{P}{N}}}$$ **Equation 1**

The speedup S is the result of the number of processing elements N and the proportional part P of parallelizable code in the algorithm. If the part of the program that is parallelizable is 50% of the total runtime then a higher speedup of 2 times cannot be achieved, no matter how many processors are added. The total runtime could be in theory as fast as the minimal runtime of the slowest parts.

Amdahl's law assumes a fixed problem size and that the size of the sequential section is independent of the number of processors. To deal with this, Gustafson's law [26] was formulated. This law is closely related to Amdahl's law but it has a scaling factor. The law can be formulated as:

$$\boldsymbol{S = P - \alpha(P - 1)}$$ **Equation 2**

where P is the number of processors, S is the speedup, and α the non-parallelizable part of the process. An example of Amdahl's law is given in Figure 5.

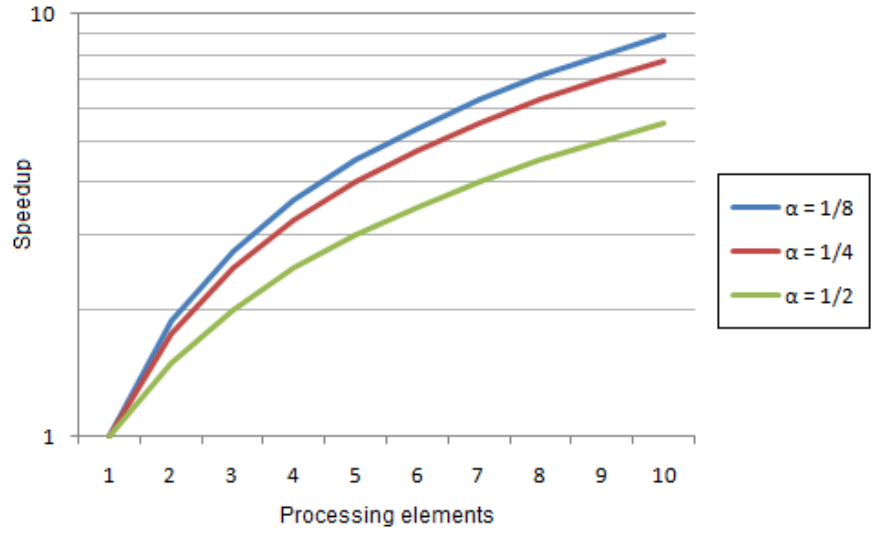

Figure 5. Various speedups with different α (non-parallelizable part)

2.3.6 Parallelization of code

Parallelization of code is a process that is best done in certain steps, the basic steps in designing parallel applications are:

- **Partitioning**, the partitioning stage is intended to find opportunities for parallel execution. The focus is to define a large number of small tasks in order to yield what is termed a fine-grained decomposition of a problem.
- **Communication**, the tasks generated by a partition are intended to execute concurrently but cannot, in general, execute independently. The computation, to be performed in one task, will typically require data from another task (previous). So data must be transferred between tasks before computation can proceed. This information flow is specified in the communication phase of a design.
- **Agglomeration**, in this stage there is a move from the abstract toward the concrete situation/case. Decisions made in the partitioning and communication phases are revisited with the goal of obtaining an algorithm that will execute efficiently on a parallel computer system. In particular, it must be considered whether it is useful to combine tasks identified by the partitioning phase to provide a smaller number of tasks of greater size each. Also it is determined whether it is worthwhile to replicate data and/or computation.
- **Mapping**, this is the final stage of the parallel algorithm design process. The place where each task is executed is specified. This mapping problem does not arise on systems providing automatic task scheduling.

2.3.7 Example of a parallelization with mutual independencies

The following is an example of a parallelization of code with no mutual dependency between the computations. In this example a matrix B is computed in T(X * Y) time steps.

```
Function  Calc_B
        For  each y<Y  do
                For  each x<X do
                        B(y,x) = sqrt( (P(x,y) +(Q(x,y) )
```

Pseudo code of the function Calc_Beam

Observing the inner loop, it is clear that input (P,Q) is not depending on the output B and all the points are unique and have to be fetched from the memory only once for every coordinate.
The double loop can be reduced to one and if there are P(X * Y) processors provided then the processing time can be reduced from T(X * Y) to T(1).

```
#Do parallel for all y<Y, x<X
Function Calc_B_Parallel
        For all Y and X do
           B(y,x) = sqrt( (P(x,y) +(Q(x,y) )
```

Pseudo code of a parallelized function Calc_Beam

2.3.8 Example of a parallelization with mutual dependencies

A second example shows mutual dependency between certain parts of the code. This means that some parts of the code depend on previous computations and should be computed in a sequential way. In the following example an algorithm for computation of an integral image is given (Figure 6) and is used in the object detection use case (Chapter 3). The algorithm sums all the previous values from the upper and left of a point in the plane. This is done in T(X * Y) and can be reduced by applying parallelization. The parallelization difficulty of this example lies in the fact that there are input/output dependency between operations in the algorithm, this complicates the parallelization and results in the algorithm to an executed in two sequential steps. The loop in the horizontal direction has a dependency in the calculation of the tmp_sum where the tmp_sum of the previous iteration is used. The loop in the vertical direction has a dependency in the sum(x,y) calculation where the sum(y-1,x) is calculated in the previous iteration.

```
# Calculate sum of the integral image
Function integral_image
        for  each  y < Y do
                tmp_sum = 0
                for each  x < X do
                        # Vertical Independent
                        tmp_sum = tmp_sum + img(y,x)
                        sum(y,x) = tmp_sum
                        if y>1 do
                                # Horizontal Independent
                                sum(y,x) = sum(y,x) + sum(y-1,x)
```

Pseudo code of the function integral image

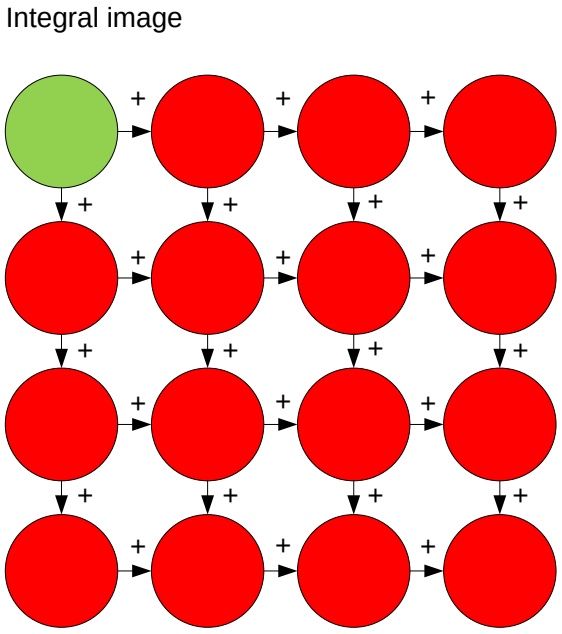

Figure 6. Schematic of the integral image calculations, the green node is the first node wherefrom each other connected node is added to, this with their own value and all the previous related nodes.

The function is split up in two steps:

```
#STEP 1
tmp_sum = tmp_sum + img(y,x)
sum(y,x) = tmp_sum
```

Pseudo code of the first step

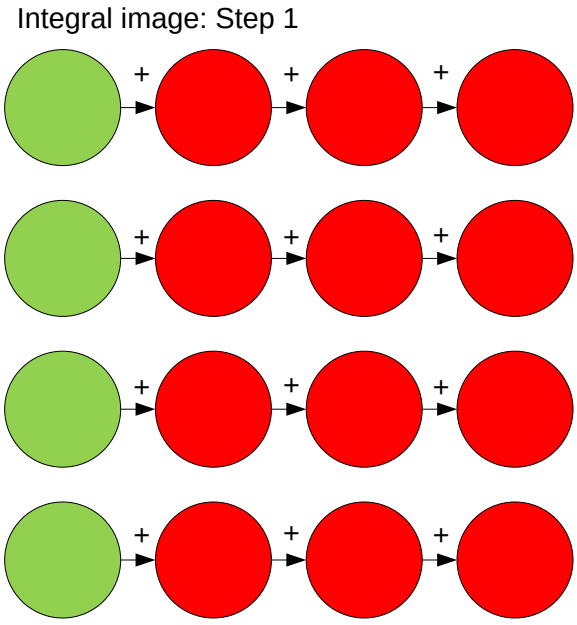

Figure 7. First step to perform the integral image in parallel in the vertical direction.

In the first step (Figure 7) it can be noticed that the sum is calculated from the previous computation in the x-axis, the accumulated value is stored in the tmp_sum and is reset to zero for each row. So there is in this step a dependency in the horizontal direction forcing it to be computed sequentially in this direction. There is no dependency in the vertical direction, this implies that this operation can be done concurrent in the vertical direction.

```
#STEP 2
sum(y,x) = sum(y,x) + sum(y-1,x)
```

Pseudo code of the second step

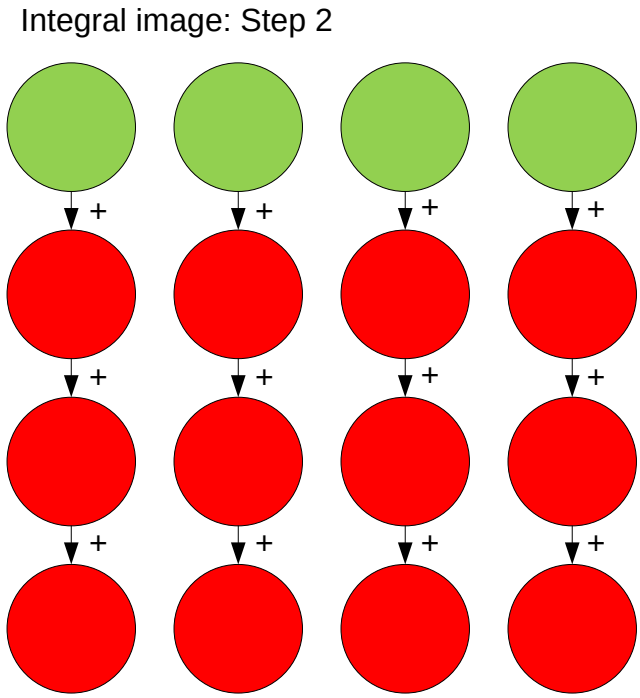

Figure 8. Second step to perform the integral image in parallel in the horizontal direction.

From the second step (Figure 8) it can be noticed that the operation dependents on the upper (previous) calculation in the y-axis so computation is done sequentially in this direction. It does not have dependency in the x-axis making it possible to do this operation concurrently in horizontal direction.

If findings above summarize then the following can be concluded:

- Two steps that has to be executed sequential after each other.
- Each step has in- and dependencies resulting in a parallel and sequential execution.

With this information the following solution formulated:

```
# Do parallel for all y<Y (STEP 1)
Function integral_image_step1
        For all Y do
            tmp_sum = 0
            for each  x < X do
                    tmp_sum = tmp_sum + img(y,x)
                    sum(y,x) = tmp_sum

# Do parallel for all x<X (STEP 2)
Function integral_image_step2
        For all X do
            for each  y < Y
                    sum(y,x) = sum(y,x) + sum(y-1,x)
```

Pseudo code of the parallelized function integral image that is executed in parallel in two steps

The first step is executed and after this step is completed the second step is executed.
Provided that there are P≥X and P≥Y processors then the integral image is computed in T(X+Y), instead of T(X*Y).

2.4 NVIDIA CUDA (GPU)

CUDA (Compute Unified Device Architecture) is a parallel computing architecture developed by NVIDIA. CUDA enables programmers to use GPUs for non-graphics computations. GPUs with their highly parallel architecture are capable of processing over more than a thousand threads at high speeds. This is why CUDA and GPU computing have gained interest from the industry and academia.

2.4.1 GPU Specification

GPUs manufactured by NVIDIA are since the G80 architecture CUDA-enabled, this means that this architecture is able run CUDA applications. All newer architecture is able to support CUDA specification making it downwards compatible. In Table 1 a selection of the CUDA-enabled GPUs is presented divided in three groups. Each group has its price range in respect to the performance range.

Table 1. Specification comparison between GPUs

Type	high-end	mid-range	low-end
Card Type	GTX285	8800GTX	9600GT
CUDA hardware ver.	1.3	1.0	1.1
Cores	240	128	64
Core Clock	648 MHz	575 MHz	600 MHz
Memory Bandwith	158.9 GB/s	86.4 GB/s	57.6 GB/s
Memory Size	1024 MB	768 MB	512 MB
Single precision perf.	933 GFLOPS	518 GFLOPS	312 GFLOPS
Double Precision perf.	78 GFLOPS	not supported	not supported
Consumption (TDP)	~183 Watt	~177 Watt	~59 Watt
Price Range	>€300	>€100 <€300	<€100

The high-end GPU can process almost 1 TFLOPS and has 240 cores. The cores are simplified and lack some features general purpose processors have, for example branch prediction and advance pre-fetching techniques.

2.4.2 CUDA Architecture

All CUDA-enabled GPUs have in general the same architectural design. When a program (kernel) is launched on the GPU it is mapped inside a grid. A grid (Figure 9) is composed by blocks and each block is performing a part of the data operation. Inside each block there are threads, shared memory and registers. The threads process parts of the data assigned to the block. Each thread has access to the registers, shared and global memory. Each thread has its own registers assigned to it, these registers can be read and written very fast with almost no delay. All the threads in the same block share the shared memory with each other. This memory is almost as fast as the registers. The shared memory cannot be shared between other blocks. The largest memory location is the global memory. This memory can be accessed directly by all the threads within the blocks. There can only be one kernel active on a GPU at a time. More detailed information is given appendix A and in the CUDA documentation [24].

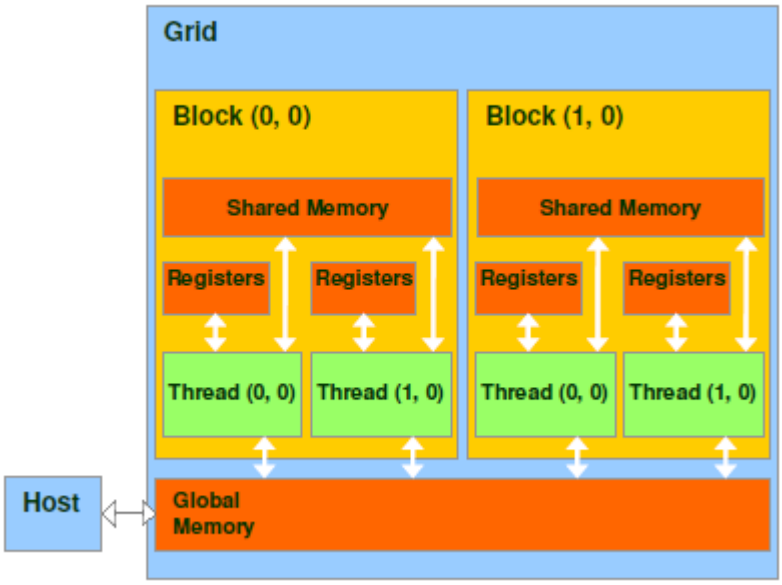

Figure 9. CUDA architecture

2.4.3 Comparison GPU vs. CPU

In Table 2 specifications of a GPU and CPU are shown. Parallelism is about 60 times higher on the GPU than on the CPU. The clock speed of the GPU processor cores are, however, half the speed of the CPU. This makes the CPU to be much better processor for serial processing. Memory Bandwidth of the GPU is up to 10 times higher than the CPU. Memory size of the GPU is limited compared to CPUs equivalent and this will result in extra overhead for memory swapping. The number of single precision floating point operations per second (FLOPS) of the GPU is 18 times higher than the CPU can perform. The performance for double precision is however almost the same. This makes it important to determine what grade of precision the application requires.

Table 2. Specification comparison between GPU and CPU

Type	GPU *	CPU **	(GPU/CPU)
Cores	240	4	60x
Core Clock	1476 MHz	3200 MHz	0.5x
Memory Clock	1242 MHz	800 MHZ	1.5x
Memory bandwith	158 GB/s	16 GB/s	10x
Memory Size	<1024 MB	>1024 MB	∞
Single precision	933 GFLOPS	70 GFLOPS (17.5 single-core)	18x
Double Precision	78 GFLOPS	70 GFLOPS (17.5 single-core)	1.1x
Price	€350	€800	0.4x

* NVIDIA GeForce GTX285
** Intel Core i7 Extreme

2.5 Application programming interfaces (API)

Most applications are not written to use multiple concurrent threads intensively.
Therefore software developments interfaces are created to enable the use of multi-processors and create multiple threads for them. Some existing parallel programming models such as Cilk++ [27], OpenMP [28] and MPI [29] can be used on multi-core platforms. Intel introduced a new abstraction for C++ parallelism called TBB [30]. Other research efforts include the Codeplay Sieve System [31], Cray's Chapel [32], Sun's Fortress [33], and IBM's X10 [34]. NVIDIA introduced their own model named the CUDA API to program parallel programs specialized for their GPUs.

2.5.1 CUDA API

CUDA is a parallel computing architecture developed by NVIDIA, see Paragraph 2.4. CUDA enables programmers to apply GPUs for non-graphics calculation. GPUs, with their highly parallel architecture, are capable of processing over more than a thousand threads at very high speed. NVIDIA developed an API to enable programmers to create applications running on CUDA-enabled GPUs. The CUDA API is a library consisting of functions required for programming CUDA-enabled GPUs. NVIDIA provide the NVCC compiler needed to generate binary code which can be executed on compatible GPUs. CUDA is specified for programming language C/C++ and Fortran.

2.5.1.1 Example

In following short example all elements of an array is calculated by squaring each element.

```
for(int i = 0;i < 512;i++)
{
        array[i] = array[i] * array[i];
}
```

C/C++ sample of a for-loop

Translating the above code to CUDA will result in a function similar to the one below:

```
__global__ void multiply_array_element(Float *array)
{
        int i = threadId.x;

        array[i] = array[i] * array[i];

        return;
}
```

CUDA sample of the parallelized for-loop.

This function is called a kernel and is executed on the GPU. The *__global__* tag enables the kernel to be called from the host device. Inside the kernel the *threadId* variable is read and is used as an index for the array. Depending on how many threads and blocks are defined for execution of the kernel, the device will execute the kernel for each unique *threadId* and *blockId* O(#thread * #blocks). In this sample the *blockId* is always 1 and the *ThreadId* is a number from 0 to 511. The kernel will automatically be mapped by the device on as many streaming multiprocessors available to execute as many threads as possible concurrently.

```
void multiply_array(Float *h_array)
{
        // Set number of blocks
        dim3 grid(1,1,1);

        // Set number of threads
        dim3 threads(512,1,1);

        // Allocate array on device
        cudaMalloc( (void**) &d_array, sizeof(float) * threads)

        // Send Array to device
        cudaMemcpy(d_array, h_array, threads, cudaMemcpyHostToDevice);

        // Execute Kernel
        multiply_array_element<<< grid, threads >>>(d_array);

        // Receive Array from device
        cudaMemcpy(h_array, d_array, threads, cudaMemcpyDeviceToHost);

        return;
}
```

CUDA sample of the initialization of the parallelized for-loop.

The function mentioned above is executed on the host computer. Within this function certain tasks are executed before the kernel is launched on the GPU. The first task is setting up the number of threads and blocks, that are being used to execute the code.
Then the space needed for the array on the memory of the device side (GPU) is allocated.
When that is done transference of the array from the host to the device possible.
Now the kernel can execute on the GPU, this is done by the following line of code.

```
[Name of Device function]<<<[#blocks],[#threads]>>>([arguments],...,[N]);
```

CUDA call kernel syntax

This is very similar to the way general function calls are created in C/C++ except for the <<<…>>> part. Within this part the number of threads and blocks can be defined.
After executing the kernel it is in some cases necessary to retrieve the results from the device. This is performed at the next function call, which copies the array data from the device to the data array on the host side.

2.5.2 OpenMP

OpenMP is an parallel programming model that enables programming for multi-core processors. The MP in OpenMP stands for Multi Processing, where Open implies that the API is an open standard. Everyone can make their own implementation without the need to pay royalties. OpenMP is specified for programming language C/C++ and Fortran.

To utilize multiple-cores or processors, it is needed that some parts of the code is written to be executed in parallel. This is usually done by creating multiple threads and assigns each thread to a part of the application to be executed. If this is done well, a large amount of parallelism can be achieved, but in general this task is very difficult to be managed. Many operations have

dependencies, making concurrent execution impossible. By making use of compiler directives, OpenMP can determine which part is parallelizable and decides in runtime which thread runs which part of the task.

2.5.2.1 Example

Following an example on how OpenMP can be enabled:

```
#pragma omp parallel for
for(int i = 0;i < 512;i++)
{
        array[i] = array[i] * array[i];
}
```

Example of C-code making use of an OpenMP directive

In this example the directive *#pragma omp parallel for* gives a message to the compiler that the content within the brackets can be executed in parallel. The content in the for-loop will then be compiled in such way that it will be executed in parallel.

2.5.3 Future development: OpenCL

OpenCL (Open Compute Language) [35] is an API dedicated to massively parallel computation. This development is initiated by Apple and the Khronos Group, a consortium of manufacturers dealing with the development of open APIs as OpenGL. Applications written for CUDA will not run on other architectures like Cell or ATI Radeon. That is why Apple has proposed OpenCL to harness the power of GPUs and other multi-core targets (Figure 10) without being locked in a range of products from a particular manufacturer (NVIDIA, ATI, IBM etc.). OpenCL can be seen as an API to standardize this.

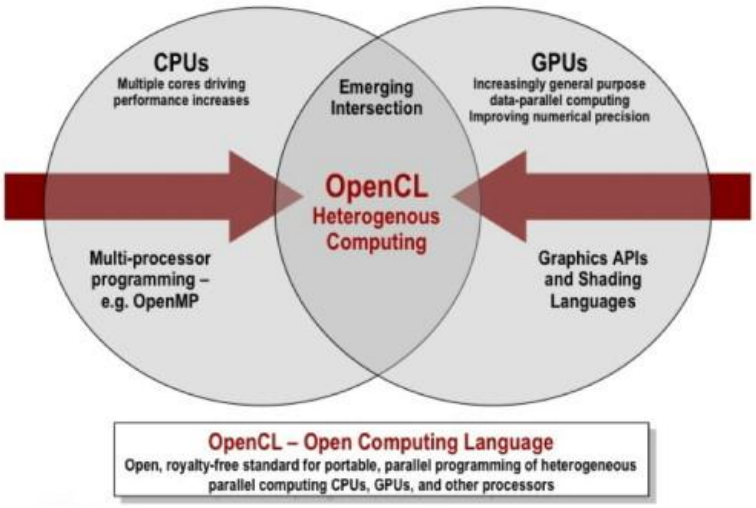

Figure 10. OpenCL, merging two architectural divisions two one computing language [35]

OpenCL has no hard specifications, it allows different specifications, depending on the accelerator used. The developer must take this into account and has to ensure compatibility between the code and the accelerator, because in some cases the code will be incompatible with

some accelerators. An example of a soft-specification is endianness which is not specified. Although the code written in OpenCL is in some cases less portable between some accelerators it opens the possibility to produce more adaptive code to the specifications of an accelerator.

An API supporting a wide range of devices simplifies the portability between them. However, developing a code, compatible with several OpenCL accelerators, will be complex. It will be necessary to write different codes for each supported accelerator, since the effective implementation of an algorithm can differ much from one accelerator to another or less optimized code based on the common points of different accelerators must be accepted.

OpenCL supports two models of parallelism, the data-level and task-level parallelism (Section 2.3). It also supports extensions such as OpenGL. It can include native kernels designed to run on the target without being written and compiled via OpenCL. Each manufacturer is free on how they implement these native kernels. The use of a proprietary API or a kernel directly prevents native code to run on different accelerators.

The question arises whether or not OpenCL could be a threat for the future development of CUDA. It is very unlikely that the arrival of OpenCL conflicts with the development of CUDA. This is because a proprietary API allows the manufacturer to move faster on its own GPUs and develop new features without disclosing to the competition of architectural details on its forthcoming chips.

The Khronos Group has developed with OpenCL an API that opens many doors, but without imposing anything, which has advantages as well as disadvantages. While it is undeniable that the industry is going to massively parallel computing, it will be a new tool with multiple uses. It seems that OpenCL will unfortunately not allow developers to easily utilize the potential computing power of modern architectures.

2.6 Development process evaluation

Another goal of this research is to investigate the time required for the implementation process of an existing single threaded application to a GPU. During the implementation process track is kept on the amount of time spend to realize each function in the application. Based on this information a plot of the whole process was made for further analysis. In Figure 11 an example of the development progression of an implementation on the GPU is shown. Each point on the line of the graph represents a certain step in the implementation process. The graph is divided in the following three development phases:

I) Profiling
II) Implementation
III) Optimization

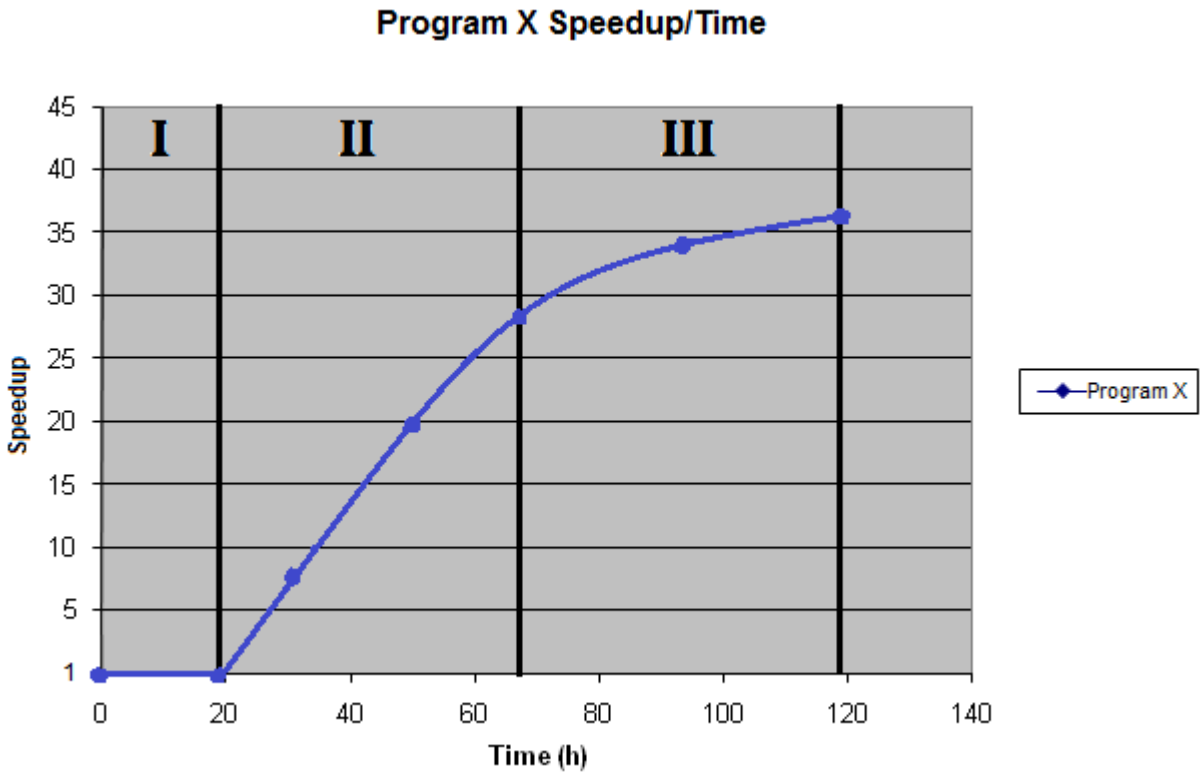

Figure 11. Graph showing development process, divided in three development phases

I) Profiling
In this phase code is converted and analyzed to provide the right data for the GPU. The length of this phase highly depends on the size and complexity of the code. In this phase there is no gain because the code is not executable on the GPU. Only the framework required for the implementation is created.

II) Implementation
In this phase various executable functions are implemented on the GPU. The functions implemented in this step are expected to be the most significant, taking most of the execution runtime. In this step a speedup is expected from the reference version.

III) Optimization
In this phase the implementation is analyzed and functions are optimized. It is expected that this will result in an increased speedup. It depends on the requirements of the application whether it is interesting to perform a full optimization or not.

Chapter 3: Case Study: Object detection

In this chapter one of the case studies done for this research is discussed. This case study concerns object detection. Object detection is used by TNO mainly in the field of security and automotive, see sample Figure 12. Having real-time performance for detection is an important requirement for a real-life environment. The goal of this case study is to exploit a GPU for heavy calculations required for object detection.

In Section 3.1 a short introduction on the theoretical principles behind object detection will be given. In Section 3.2 an overview is given of the object detection application. In Section 3.3 the single threaded application is profiled. After the profiling, the first implementation of the code on the GPU is discussed. In Section 3.4 the results are analyzed, to realize an optimization of the code. In Section 3.5 the overall analysis of the whole process is discussed and finally in the last section of this chapter conclusions based on the previous analysis are presented.

Figure 12. Sample of object detection, detecting of cars from behind

3.1 Object detection (Computer Vision)

3.1.1 Introduction

There is an increased demand for systems, which provide highly detailed information about traffic- and safety-indicators, to evaluate the effects of infrastructural measures like narrowed driving lanes and rush hour lanes. Video-Based Monitoring (VBM) is a technique to automatically evaluate behavior of individual traffic contesters from videos. Trajectories of individual road users can be precisely measured from consecutive video images. Maneuvers, like lane switches, overtake and following can be derived and quantified with time-to-collision, following-time, and mutual longitudinal respectively lateral distances. Detecting extreme differences in the measurements, dangerous and critical situations and potential conflicts can be detected and avoided.

3.1.2 Object detection

An important feature of VBM is the extraction of objects (cars) from video images. This process, also called object detection, consists of two phases that are required to come to a positive detection (Figure 13). The first phase in object detection is the training phase. In this phase features and weighting factors are derived from a set of known positive and negative example images. This training is very time consuming, depending on the amount of sample images and precision requirements, but has to be processed only once. The results from the training phase are applied in the detection phase, this is the second phase of the object detection. In this phase an image is evaluated by means of feature extraction. From this feature extraction, multiple detections can be extracted from an image. The detection phase is performed many times for each of the provided image. To gain detection performance, best results will be achieved improving the detection phase.

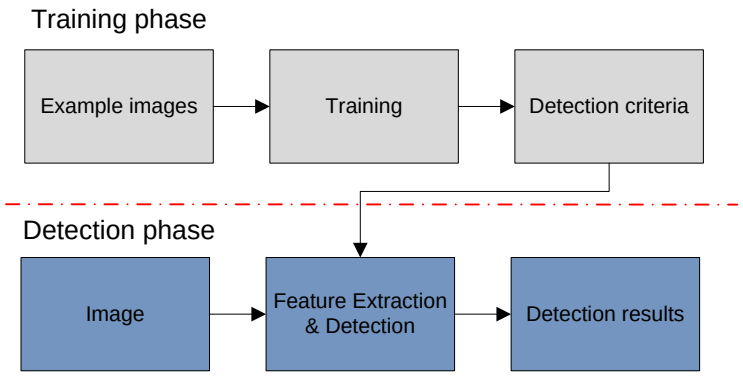

Figure 13. Object detection, above training phase and under detection phase

3.1.3 Feature extraction

Features make use of the fact that objects often have some general properties. An example for faces is the darker region around the eyes compared to the forehead.

The algorithm calculates for features, characteristic to an object called *haar features*, within an image. This is done by placing a rectangular detection window over the image.

The detection window slides over the image scanning the whole image. Within the window haar features are extracted to detect a certain object. Thousands of calculations are applied on this window. Each calculation has the same pattern: calculate the difference between the black and white surfaces. Some samples of characteristic surfaces are shown in Figure 14. The total sum of pixels within the rectangle is needed for the comparison between the surfaces. To perform the detection in a time reduced way the Fiona Jones method [5], based on integral representation of images, is used. This greatly reduces the time to calculate the summation of rectangle.

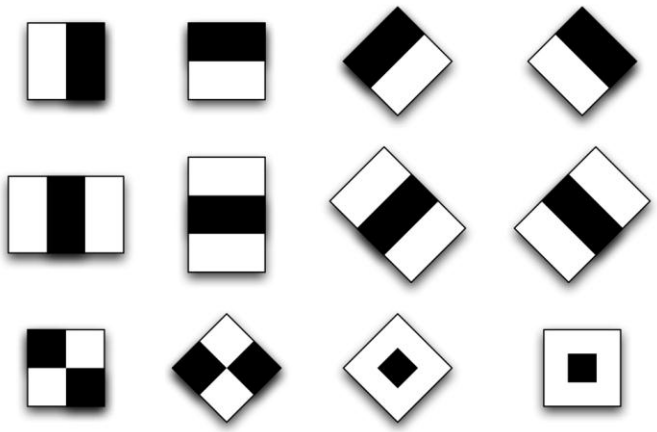

Figure 14. Examples of various types of haar features

An integral image is generated from the source image where the value of a pixel is calculated from the sum of all the pixels above and to the left of the pixel (Figure 15a). The integral sum of pixels can be retrieved from a rectangle within the image. This can be done by performing the following calculation: $A + D – (B + C)$, where the variables are the corners of the rectangle. An example is shown in Figure 15b.

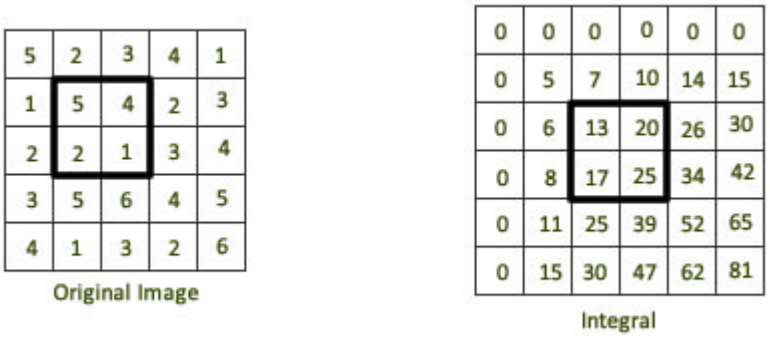

Figure 15a. Original image (left) converted to the integral image (right)

0	0	0	0	0	0
0	5	7	10	14	15
0	6	A 13	20	B 26	30
0	8	C 17	25	D 34	42
0	11	25	39	52	65
0	15	30	47	62	81

Integral

Figure 15b. A+D-(C+B) represents the sum of image values within rectangle(A,B,C,D)

The integral sum of the image is calculated as follows:

$$\boldsymbol{Sum}(\boldsymbol{X},\boldsymbol{Y}) = \sum_{\mathbf{y=0}}^{\mathbf{Y}} \sum_{\mathbf{x=0}}^{\mathbf{X}} \mathbf{I}(\mathbf{x},\mathbf{y}) \qquad \textbf{Equation 3}$$

Where I(x,y) is the function of the integral image. Also the squared and the tilted sum are often used for the object detection. These sums are calculated as following:

$$\boldsymbol{Square}(\boldsymbol{X},\boldsymbol{Y}) = \sum_{\mathbf{y=0}}^{\mathbf{Y}} \sum_{\mathbf{x=0}}^{\mathbf{X}} \mathbf{I}(\mathbf{x},\mathbf{y})^{\mathbf{2}} \qquad \textbf{Equation 4}$$

$$\boldsymbol{Tilted}(\boldsymbol{X},\boldsymbol{Y}) = \sum_{\mathbf{y=0}}^{\mathbf{Y}} \sum_{\mathbf{x=0}}^{\mathbf{abs(x-X)<y}} \mathbf{I}(\mathbf{x},\mathbf{y}) \qquad \textbf{Equation 5}$$

3.1.4 Scaling

Scaling has to be performed to detect objects of various sizes, This can be done by scaling the detection window or by scaling the image (Figure 16). Preferably the window is scaled leading to less computational overhead. The standard factor is set to 125% and is configurable. This means that only up-scaling is applied, resulting in a minimum detectable object of 24x24 pixels. Hereby it is possible to detect multiple sizes of objects by moving the detection window over the image. One object can be detected in multiple scaling factors. These detections can be merged finally to a single detection.

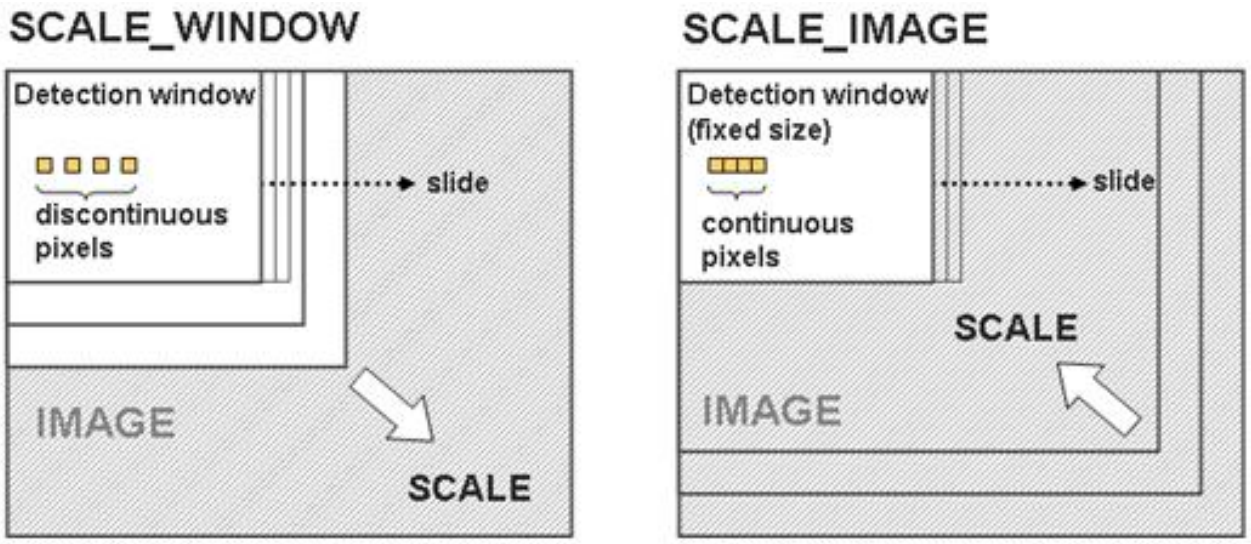

Figure 16. Left: Detection window is scaled, size of the image is fixed.
Right: Image is scaled, detection window size is fixed.

3.2 Object detection application

A popular library to perform object detection is the Open Computer Vision (OpenCV) library [19]. This library is a set of programming functions, mainly aimed at real time computer vision. The version of OpenCV libraries, used for this use case, is version 1.06.
In this chapter the detect object function of the OpenCV library is discussed. This function is called from an application, named object detection, created to detect objects from one or more image files. Figure 17 shows the flow chart of the object detection application. The application first initializes the process. Within a for-loop the image(s) is/are initialized and the object detection is called. After detection, the results are displayed to the output device. If a stream from e.g. a live camera or from a file is provided, then the for-loop will perform the detection on each image it receives from the stream. This will be done until the end of the stream.

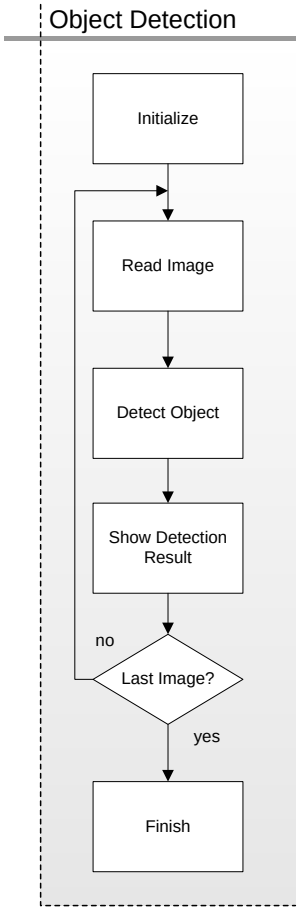

Figure 17. Flow chart of the object detect application

3.2.1 Data structure

The main data structure, shown in Figure 18, used for the object detection is the cascade.
For a detection a set of classifiers (cascade) is required. Each classifier combines a set of features. Each feature has rectangles, weighting-, threshold- and alpha-factors. The features of each classifier are connected by means of a linked list. Each classifier has a threshold value.

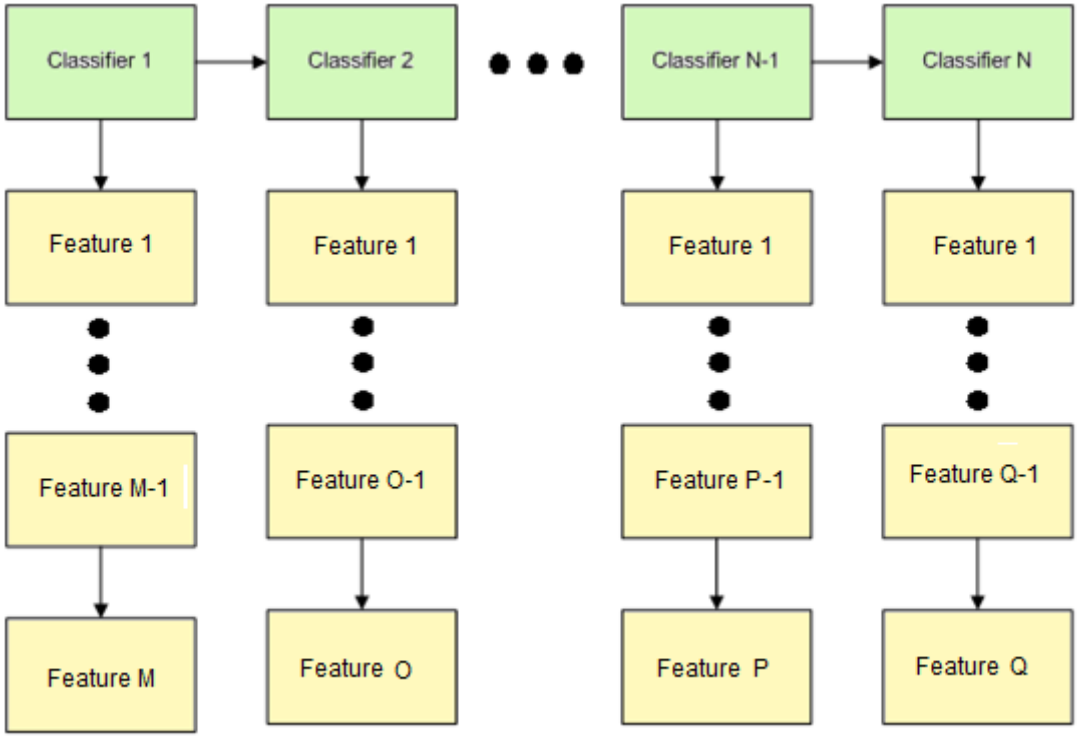

Figure 18. Data Structure of the Cascade, N number of Classifiers where each classifier has a variable number of features

3.2.2 Detect object

The detect object function is called within the object detection application (Figure 17). This function is incorporated in OpenCV and is used to perform object detections from an image by making use of haar features. The detection function is able to detect objects by making use of two detection modes. The flow charts of both detection modes are shown in Figure 19. These modes differ in the way the scaling is done. A mode can be selected by setting a flag to the input variables of the function. The SCALE_IMAGE is used to scale the image. So the integral image for each scale is recalculated. The SCALE_WINDOW scales the window and the features for each scale. Main advantage of scaling the image is that the detection window is fixed, therefore the data/pixels are continuously aligned in an array (Figure 16). Doing so the data can be fetched from memory more efficiently. The disadvantage is that scaling of the image is much more compute-intensive than scaling a window. The sum, square sum and in some cases the tilted sum has to be recalculated for each scale, where in case of window scaling all the features are rescaled to the size of the window for each scale.

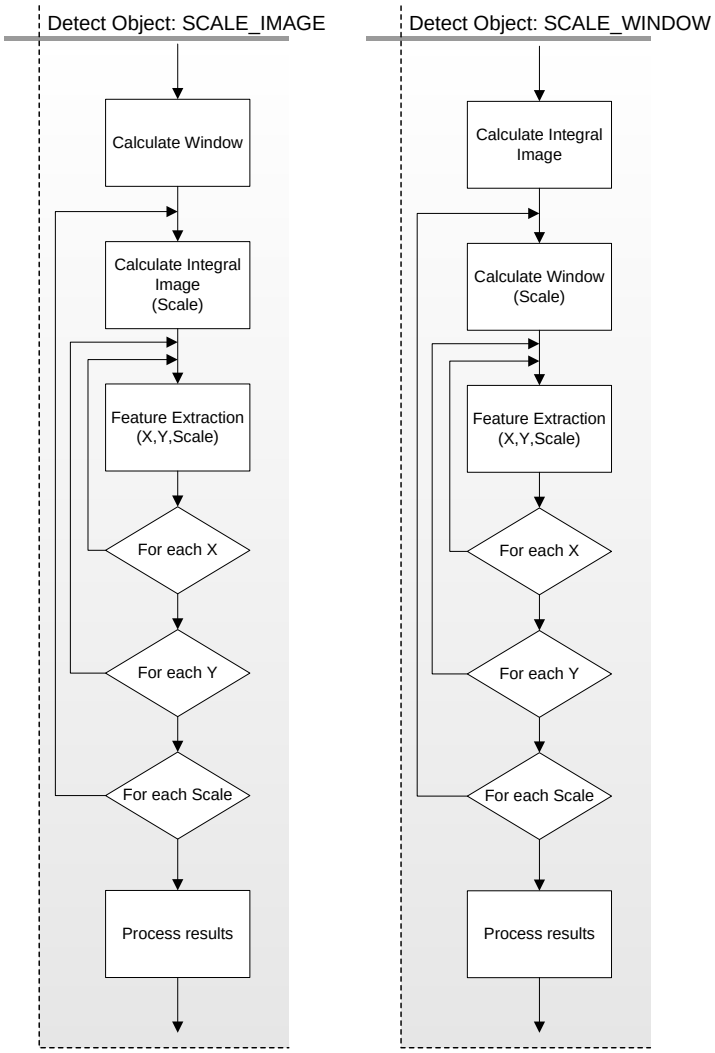

Figure 19. Left: Flow chart of the detect object function (SCALE_IMAGE)
Right: Flow chart of the detect object function (SCALE_WINDOW)

3.2.3 Feature Extraction

The *get haar cascade features* function is part of the detect object function (Figure 19) and is used to extract features from an image. The flow chart of this function is shown in Figure 20. The function extracts haar features from a window positioned at a x-y position on the image. The x-y position is the most upper left position of the window. The size of the window depends on the type of detection, scale image or window respectively fixed or scaled. The function loops through all the classifiers to extract the features from the window. The result of one feature is the sum of 8 extraction points or in case of three rectangles, 12 points. The sum is compared to a feature specific offset that, depending on the value, results in a weighting value.

The results of all weighting values of one classifier are summed and compared with a classifier specific offset, which determines whether the set of features is positive or not. If the result is positive, the next classifier will be evaluated, otherwise the function will abort and return a negative (false) value, notifying the underlying function that on this x-y position no object was found. When all the classifiers of the cascade are positive, the function will return a positive (true) value, meaning that an object was found.

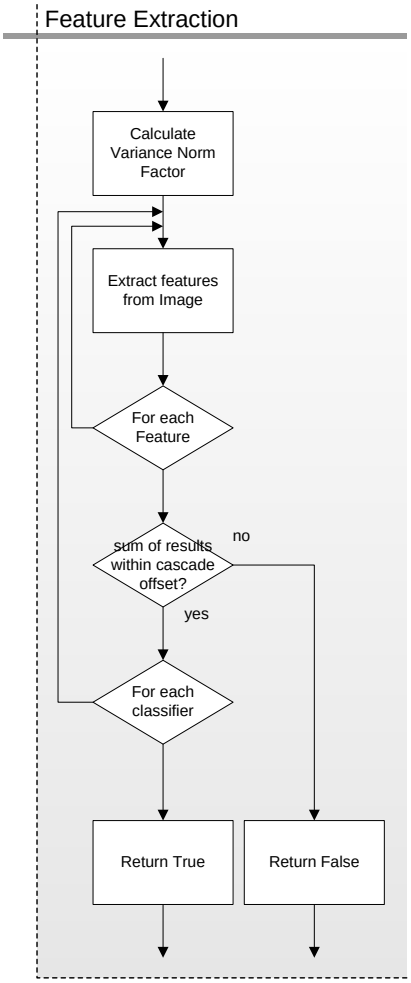

Figure 20. Flow chart of the Features Extraction function

3.3 Profiling

The object detection application is being profiled to get more inside in what the major parts are within the algorithm. With this information decisions can be taken and priorities can be set.

A cascade set is used to do the profiling on each type of scaling mode. The cascade is used for car detection and is trained by TNO to detect cars on major highways in the Netherlands. The total object detect function has been analyzed and split up in some major steps. The output of each step is required for subsequent steps. Both the scale window and the scale image modes are included in the profiling.

The application is split-up in the following major steps:

- Feature extraction
- Calculate window
- Calculate integral
- Resize image

All steps are executed in the detect object function (Figure 14), The resize image step is only used in the scale window mode. The runtime of each step is the average time required for a single detection. The results are generated from car detections on a traffic video of 2 minutes length.

3.3.1 Specifications of the reference PC

For the profiling, a personal computer (PC) is used to execute and measure the runtime of the object detection application. The PC also serves as reference for the performance comparison to the GPU. In Table 3 the specifications of the system are shown. The results of the profiling are based on a single-cored execution of the application.

Table 3. Specification of the reference PC used for profiling

Processor	Intel Core 2 Duo E6400
CPU Speed	2.13 GHz
Number of Cores	2 (only 1 used for profiling)
L2 Cache Size	2 MB
Memory Size	2GB
Bus Speed	1066 MHz
Operating System	Microsoft Windows XP 32bit

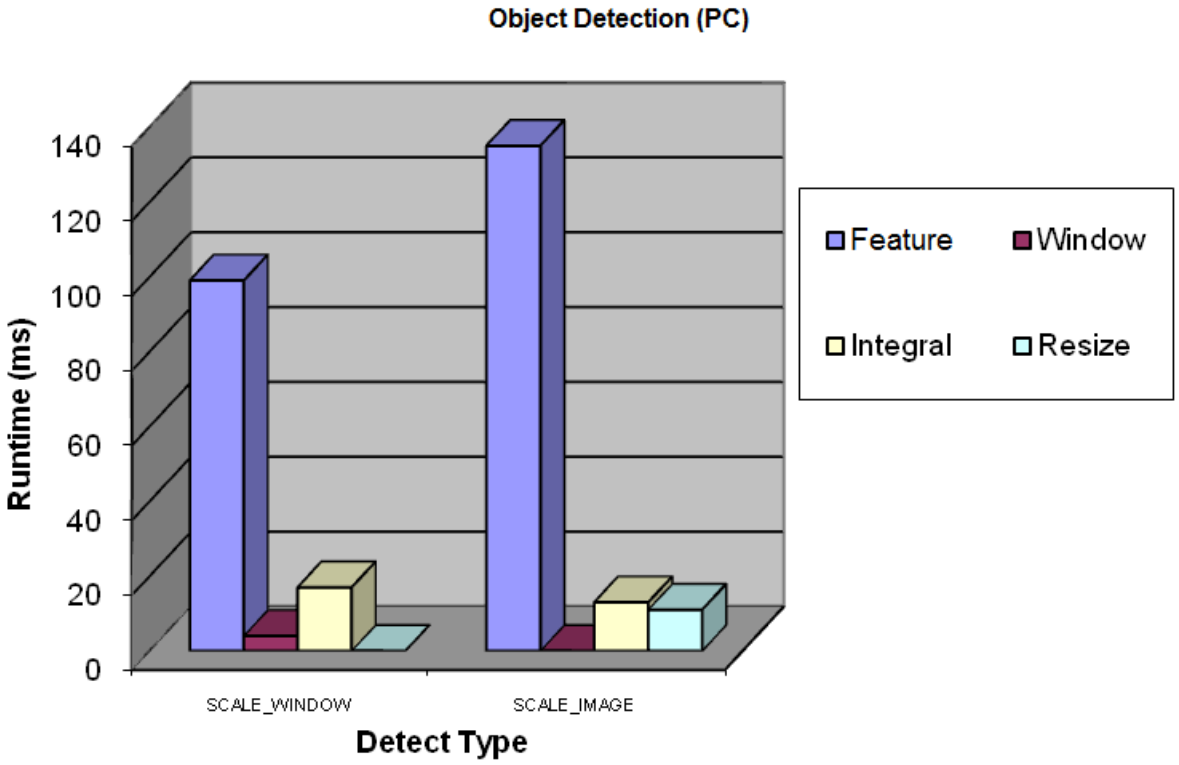

Figure 21. Profiling Results for the Scale window vs. Scale image modes

Table 4. Profiling results, scale window and scale image modes

Object detection				
Function name	SCALE_WINDOW		SCALE_IMAGE	
	Average time (ms)	% total	Average time (ms)	% total
Feature Extraction	99	83	135	85
Calculate window	4	3	-	-
Calculate integral	17	14	13	8
Resize image	-	-	11	7
Total	120	100	159	100

3.3.2 Scale window mode

As can be seen from the results of Table 4 and Figure 21, the feature extraction step is in time the most significant step, taking 83% of the total runtime. The next factor of significance is the calculation of the integral image (14%). The calculation of the window is less significant and can be, considered as insignificant (Amdahl). Concluding: the main target is the parallelization of the detect object step, followed by the integral calculation.

3.3.3 Scale image mode

The feature extraction step is in time the most significant step for the detection, taking 85% of the total runtime. The next factors of significance are the calculations of the integral image (8%) and the resizing image step (17%). These two steps take up almost all of the remaining time. The calculation time of the window is negligible, since it is calculated once. As for the scale window mode the parallelization of the detect object part will be investigated first. The parallelized version of the integral image step, done for the scale window mode, can be reused.

3.3.4 Scale image vs. scale window comparison

The profiling results for the scale image modes differ from the scale window modes.
Most remarkable is the higher time consumption of the feature extraction part of the scale image versus the scale window. The reason for this is that the scale image mode needs more steps to scan the whole image. What also can be determined is that for the scale image, the integral and resizing runtime combined, do not significantly exceed the runtime required for the scale window implementation (resp. 21 ms vs. 24 ms). The reason for this is resizing of images and lowering the amount of calculations for the integral images. Analyzing the total runtime of the feature extraction step and comparing the result to the scale window does also not show any improvement in runtime. This means, no speedup due to the locality within the window is obtained. So, based on these results, the scale window mode is the most reasonable method to be implemented on the GPU.

3.4 Implementation on GPU

In this paragraph the GPU implementation of the object detection algorithm is discussed.
The steps defined in profiling are analyzed and ported to the GPU.
Implementation results are analyzed and discussed.

3.4.1 Converting the data structure

The first step being accomplished is transferring the data structure of the cascade from the host computer to the global memory of the GPU. The main problem is the data structure being a linked list (Figure 18). The problem with a linked list is that an object or structure is pointing to the physical memory address where the next or previous object is stored. If an object is sent to another computer, or in this case a GPU, the links will not work, because they are pointing to the wrong memory address. So the linked list has to be build up again. There are several options to handle such an issue. The linked list on a GPU could for example be rebuild by recreating all the structures on the GPU and relink them to each other. The problem is that there are many small transmissions needed from the host computer to the GPU. This can be expressed in the following function:

$$\boldsymbol{Transmissions} = \sum_{\boldsymbol{i=0}}^{\boldsymbol{\#classifiers}} \boldsymbol{features}[\boldsymbol{i}] \qquad \textbf{Equation 6}$$

Where transmissions is the number of all the individual data structures sent to the GPU and features[i] is the number of features belonging to a classifier. Because of the high latency involved in sending data (transmission) to the GPU, this approach is not acceptable.
Another solution is the conversion of the linked list to one large structure with a fixed size. This data structure can be transferred to the GPU in one single transmission. The drawback of this solution is that some flexibility is lost and extra data overhead is created, caused by the fixed data size. Therefore the length of the array is set to the maximum number of features required for a certain classifier. The design of the new structure is shown in Figure 22.

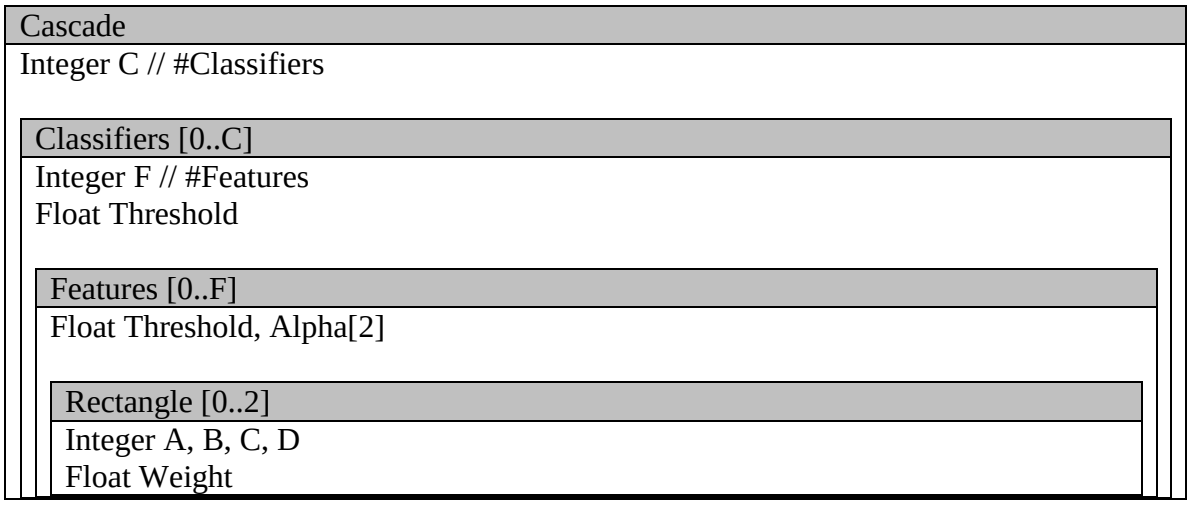

Figure 22. Design of the fixed data structure

3.4.2 Calculate window and variance norm factor

Most of the existing code for calculating the detection windows can be re-used for the GPU implementation. The biggest changes applied, are the calculations for the detection window, running concurrently with the calculation of the variance norm factor.
The variance norm factor calculation is stripped from the object detect function and can be calculated in advance on the GPU, while the detection windows are calculated on the host computer. This saves time and keeps the GPU more occupied.

3.4.3 Parallelization of the feature extraction step

This is the most important step in the object detection and also the most task intensive one.
It consists of three for-loops, as can be seen in Figure 19. The most outer loop is the scale loop. For each scale the window is scaled and the x-y positions are recalculated. Then the function loops for every x-y position and calls the feature extraction function.
The total number of calls for one image can be expressed in the following equation:

$$\boldsymbol{Fuction\ Calls} = \sum_{\boldsymbol{i=0}}^{\boldsymbol{\#scales}} \boldsymbol{x}[\boldsymbol{i}] * \boldsymbol{y}[\boldsymbol{i}] \qquad \textbf{Equation 7}$$

Where x[i] and y[i] are the number of respectively horizontal and vertical detection points for each scale expressed in i. The total sum of these points for each scale results in the number of function calls. Instead of calculating each x-y position after each other, multiple positions are calculated on the GPU concurrently in parallel. In Figure 23 the two inner for-loops are replaced for a parallel loop. This, since the detection of each position is not depending on another detection. If there are P(X * Y) processing elements, where X and Y are the maximum numbers of horizontal and vertical detection positions, the detection can be performed in T(number of scales) time steps were each scale is executed as fast as the slowest feature extraction.

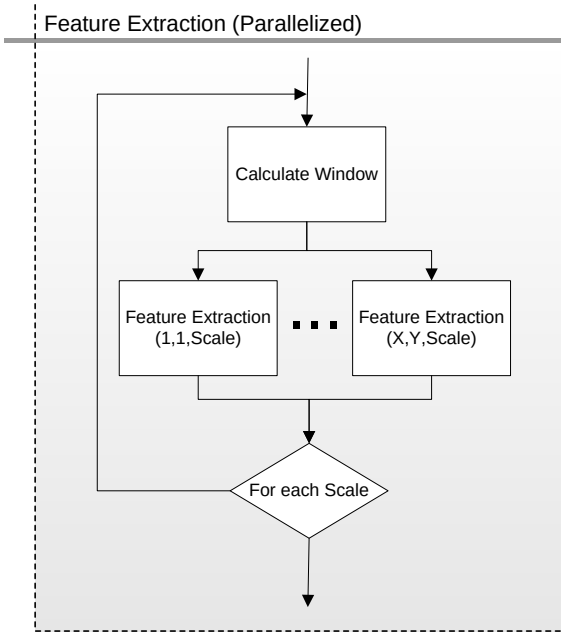

Figure 23. Flow chart feature extraction step running parallel in two axis (X,Y)

3.4.4 Parallelization of the integral image calculation

The integral image calculation is depending on previous iterations, see mutual dependencies in Section 2.5.7. The sum of values in the column is depending on the result of the sum of values in the rows. This is why the integral image calculation is done in two steps to perform parallelization. The flow chart of the integral image calculation is shown in Figure 24.

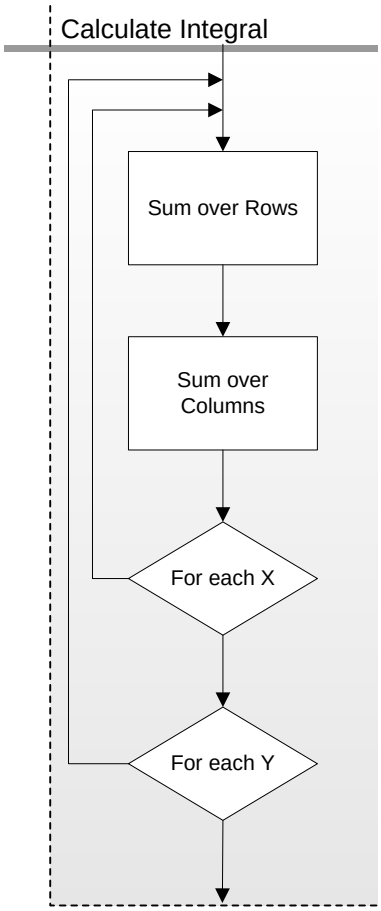

Figure 24. Flow chart non-parallel calculation of the integral image

The flow charts of the parallelized calculate integral steps are shown in Figure 25.
The sum of values in the rows is calculated in serial in the horizontal direction where the vertical direction is calculated in parallel. In the next step the sum of values in the columns is calculated in serial, where in the horizontal direction the sum is calculated in parallel. By this approach the whole process takes T(X+Y) time steps instead of T(X*Y) if there are $P(X_{max} \mid Y_{max})$ processing elements available.

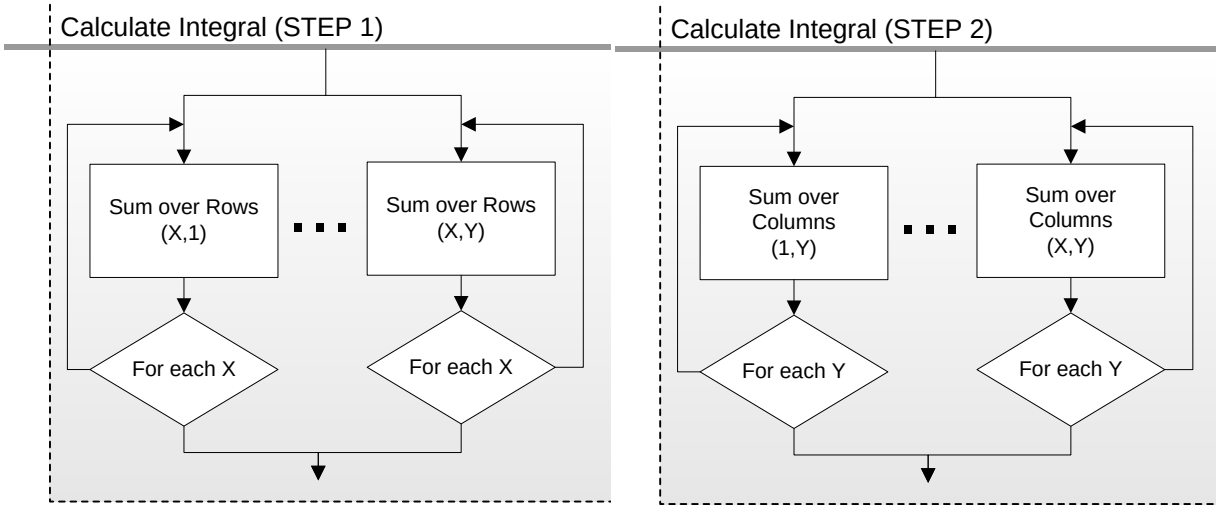

Figure 25. Left: Flow chart step 1, sum of values in the columns in parallel
Right: Flow chart step 2, sum of values in the rows in parallel

3.4.4 Implementation results

The implementation of the object detection is performed on a CUDA-enabled GPU from NVIDIA. In Table 5 the specifications of the GPU are shown. The GPU is selected for its high-end performance combined with relatively low costs.

Table 5. Specification of the GPU used for the implementation

Card Type	GTX285
CUDA chipset	GT200
CUDA version	1.3
I/O Interface	PCI-E 2.0 x16
Unified Shaders	240
Core Clock	648 MHz
Shader Clock	1476 MHz
Memory Clock	1242 MHz
Memory bandwith	158.9 GB/s
Memory Size	1024 MB
Memory Interface	512 bit
Single precision	933 GFLOPS
Double Precision	78 GFLOPS
Price	~250 Euro
Consumption (TDP)	~183 Watt

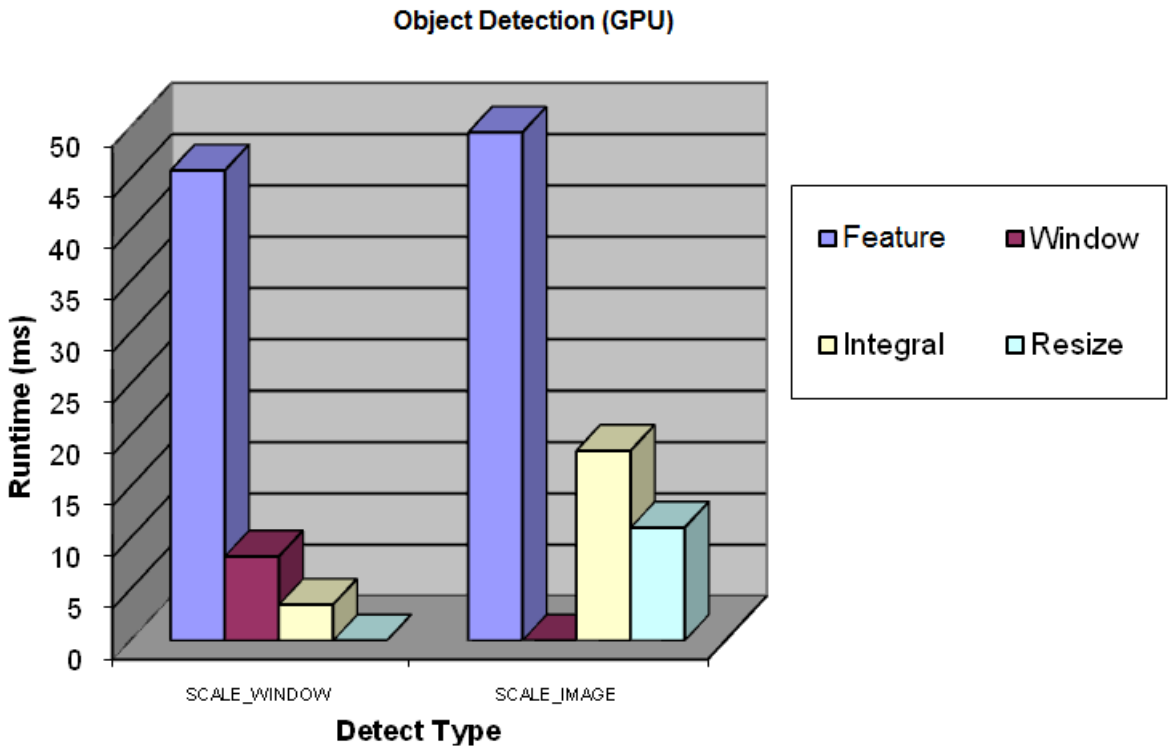

Figure 26. Implementation results, scale window and scale image modes

Table 6. Implementation results, scale window and scale image modes. The speedup is the host CPU runtime divided by the GPU runtime

Object detection				
Function name	SCALE_WINDOW		SCALE_ IMAGE	
	Total time (ms)	Speedup	Total time (ms)	Speedup
Feature Extraction	45.8	2.2x	49.5	2.7x
Calculate Window	8.2	0.5x	-	-
Calculate Integral	3.5	4.8x	18.5	0.7x
Resize image	-	-	11	1x
Total	57.5	2x	79	2x

3.4.5.1 Results scale window mode

It can be retrieved from the results of Table 6 and Figure 26 that the performance of the feature extraction step is increased with a factor 2.2, resulting in an average time of 45.8 ms. This result is not satisfying. The average runtime has to be around the 30 ms to realize real-time detection. So, further optimization is needed to increase the speed. The calculate window function has decreased in speed with a factor 2. This is since each window is first calculated on the host and then sent to the GPU, resulting in additional communication overhead. The biggest speedup of almost 5 times is achieved in the calculate integral step. An important factor for further optimization of the code is reducing communication overhead (I/O) by pre-calculating the windows for each scale and executing the scales in parallel.

3.4.5.2 Results scale image mode

The feature extraction runtime is decreased with a factor of 2.7. This is better than the scale window mode implementation, but on average it is slightly slower. The calculate integral part, unlike the scale window mode, is increased. The reason for this is, each time the integral image is calculated, the resized image is sent from the host computer to the GPU. This results in a communication overhead of 10 ms, taking almost 60% of the total calculate integral time.
For further optimization, implementation of resize image is required to gain a reduction of communication overhead. Therefore only the input image has to be sent to the GPU.

3.5 Optimization

In this paragraph the optimizations of the implementation of the GPU is discussed. The steps are analyzed and optimized to gain performance. Optimization results are analyzed and discussed. Only the scale image is optimized for the GPU, this decision was made based on the implementation result in the previous chapter.

3.5.1 Feature extraction optimization

In the previous implementation the scales are still executed serially, after each other. Therefore it is required to send new data to the GPU for each scale and wait for the scale to finish until the next scale can be processed. To reduce the amount of data transfer and the waiting times, the detection for each scale is made to be executed in parallel.
The number of parallel threads generated by including the scales to the parallel process can be expressed in the following equation:

$$\boldsymbol{Threads} = \sum_{\boldsymbol{i=0}}^{\boldsymbol{\#scales}} \boldsymbol{x(i)} * \boldsymbol{y(i)} \qquad \textbf{Equation 8}$$

Where x[i] and y[i] are the numbers of respectively horizontal and vertical detection points for each scale expressed in i. The total sum of points for each scale result in the number of threads.

All windows are pre-calculated and stored in an array of cascades. Each cascade in the array has its own pre-calculated scaling factor. Next the array of cascades is sent as one block to the GPU. While the windows are calculated on the host, the variance norm factor is concurrently calculated on the GPU. After this, the feature extraction function is executed for every x-y position and scale. The results of this operation are stored on the GPU in a large array of the size $O(X_{max}*Y_{max}*scales)$. When all the detections are performed the results are fetched from the GPU. In Figure 27 the flow chart of the parallelization is shown.

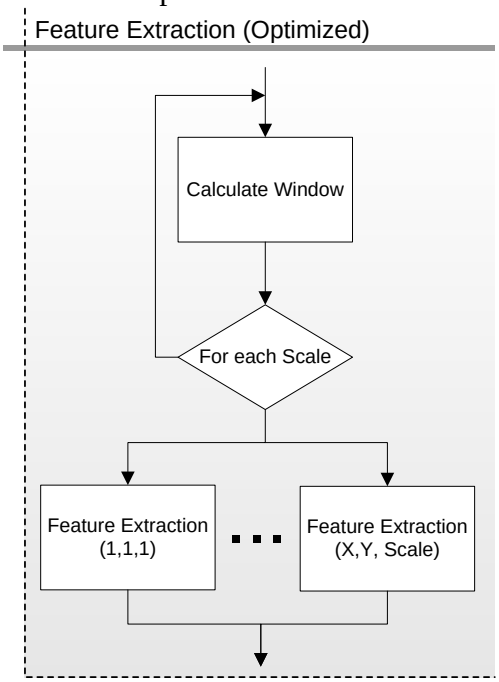

Figure 27. Flow chart feature detection step running parallel in three axis (X,Y,Scale)

3.5.2 Window calculation optimization

Since the window is pre-calculated once for each scale and the results are stored on the GPU, there is no need to calculate the window for the next image again. This is only the case if the object detection is fed with the same image size and detect precision settings. By re-using the pre-calculated windows of the previous image, the calculation window step can be skipped and the object detection can be started directly. This saves calculation time and data transfer to the GPU.

3.5.3 Optimization results

From the results in Table 7 and Figure 28 it can be observed that the performance of the detect object function is not significantly improved (1.78x) compared to the first GPU implementation. The time, to send the cascade and receive the results from the GPU is still relatively high, 8 ms of the average time of 25.7 ms. The calculate window part is improved very much due to the optimizations. The total average runtime is 30.2 ms enabling a frame rate of ~30 fps. This result in a speedup of almost 4 compared to the PC runtime.

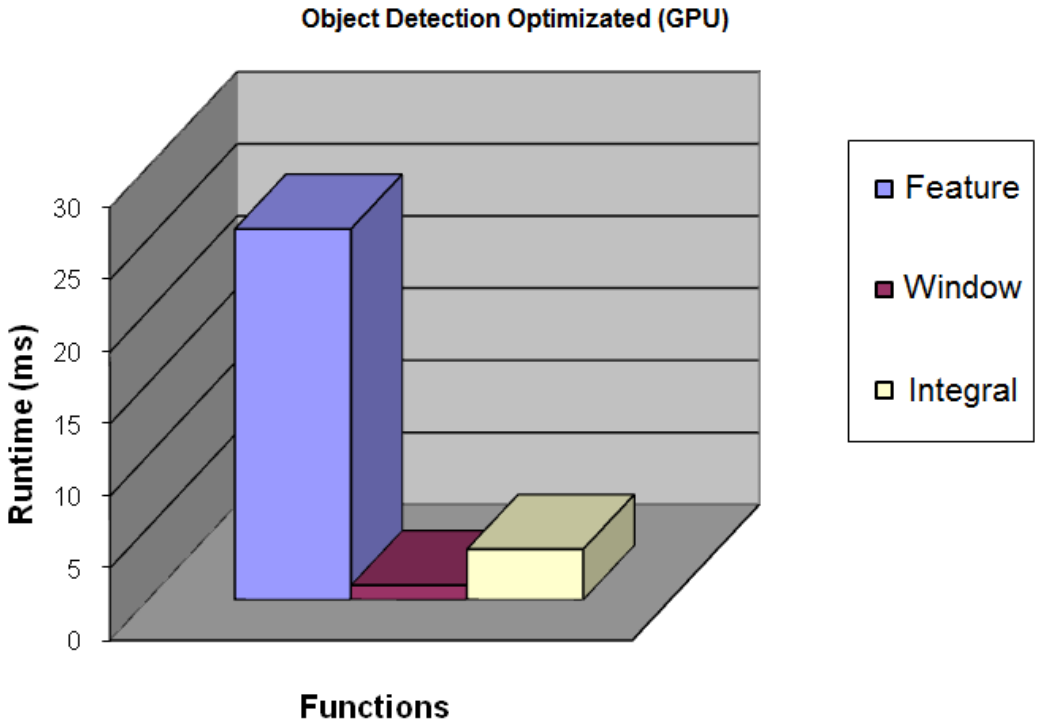

Figure 28. Optimization results for SCALE_WINDOW

Table 7. Optimization results for SCALE_WINDOW. The speedup is the host CPU runtime divided by the GPU runtime

Object detection			
Function Name	Total time (ms)	Opt. Speedup	Tot. Speedup
Feature Extraction	25.7	1.78x	3.9x
Calculate Window	1	8.2x	4x
Calculate Integral	3.5	1x	4.8x
Total	30,2	1.9x	3.97x

3.6 Analyses

In this paragraph the overall analyses of the implementation of the GPU is discussed.
The achieved performance is compared with the theoretical performance and the limiting factors are analyzed. The total development process of this use case is discussed and analyzed.
Further future optimizations are introduced and explained.

3.6.1 Limitations

To evaluate the results, a comparison is made between the results and the specifications of the NVIDIA GTX 285. The object detect function is used for this comparison. The GTX 285 can, according the specifications, deliver up to 933 Gflops for single precision floating point calculations and delivers an estimated memory bandwidth of 158.9 GB/s. The reference PC (Table 3) can theoretically deliver up to 6 Gflops and has an estimated memory bandwidth of 16 GB/s.

The calculation of the flops is very difficult for this use case because of the alternating length of the loops for the feature extraction. The average number of tested classifiers is estimated to be around 3, this is based on the way the classification is done, first classifiers falsifies ~50% of the feature extraction the second ~75% and the third ~87.5%. The amount of floating points operations required to perform the feature extraction were counted to determine the flops. The measured overall memory throughput is 30 GB/s where the estimated number of flops is 15.8 Gflops. The reference PC delivers 2.8 Gflops and a memory bandwidth of ~5.8 GB/s.

The results of the measured memory bandwidth and the estimated amount of Gflops are far beneath the theoretical max. The current implementation uses less than 2% of the theoretical performance in Gflops and is very disappointing. The memory throughput uses only ~20% of the theoretical available bandwidth. The question arises on what the limiting factors are.

After analysis, three limitations are found:

- Use of registers.
- Decision tree (branching).
- Scattered reads.

3.6.1.1 Registers

Analyzing the detect object function, it is shown that the required amount of registers (25) is exceeding the limitations to fully use the warp size of the GPU. Hereby the warp size is half the total available warp size. This means that the application could only make use of a part of the available resources on the GPU. A way to limit the amount of registers is by moving the register to the global memory. The disadvantage is that the global memory is much slower than the registers, resulting in a lower performance.

3.6.1.2 Decision tree (branching)

The feature extraction function, which is the most significant step in the object detection, uses a decision tree to determine whether an object can be found on a certain position of the image. Conditional branching is used to run through a decision tree. The cores of the GPU are not optimized to perform branching as modern general purpose processors are. These processors use techniques like branch prediction to reduce stalling as much as possible. GPU cores lack these extra hardware features, which results in less optimal performance when conditional branching is used.

An additional problem is the way the code is executed in warps. The GPU tries to put as much of the same operation, maximal 32, in one warp to be executed by the cores. The hardware will not be used to its full extend if there are not enough threads available that executes the same operation concurrently. In practice, this will very often be the case.

The final problem is the variances in runtime for the execution of the feature extraction function. As multiple feature extractions are being executed in parallel the process will take as long as the longest path. Figure 29 shows a plot of the GPU time in the vertical axis and the images provided to the GPU in the horizontal axis. The plot shows that the detection time strongly depend on the image used for detection. This is caused by the variances in runtime of the object detect function, till a false feature is found or all features are positive. Detection time increases when multiple objects are found. Fake objects having a certain amount positive features and at least one false feature can stall the runtime. The detection of the false features can vary between the first and the last feature comparison.

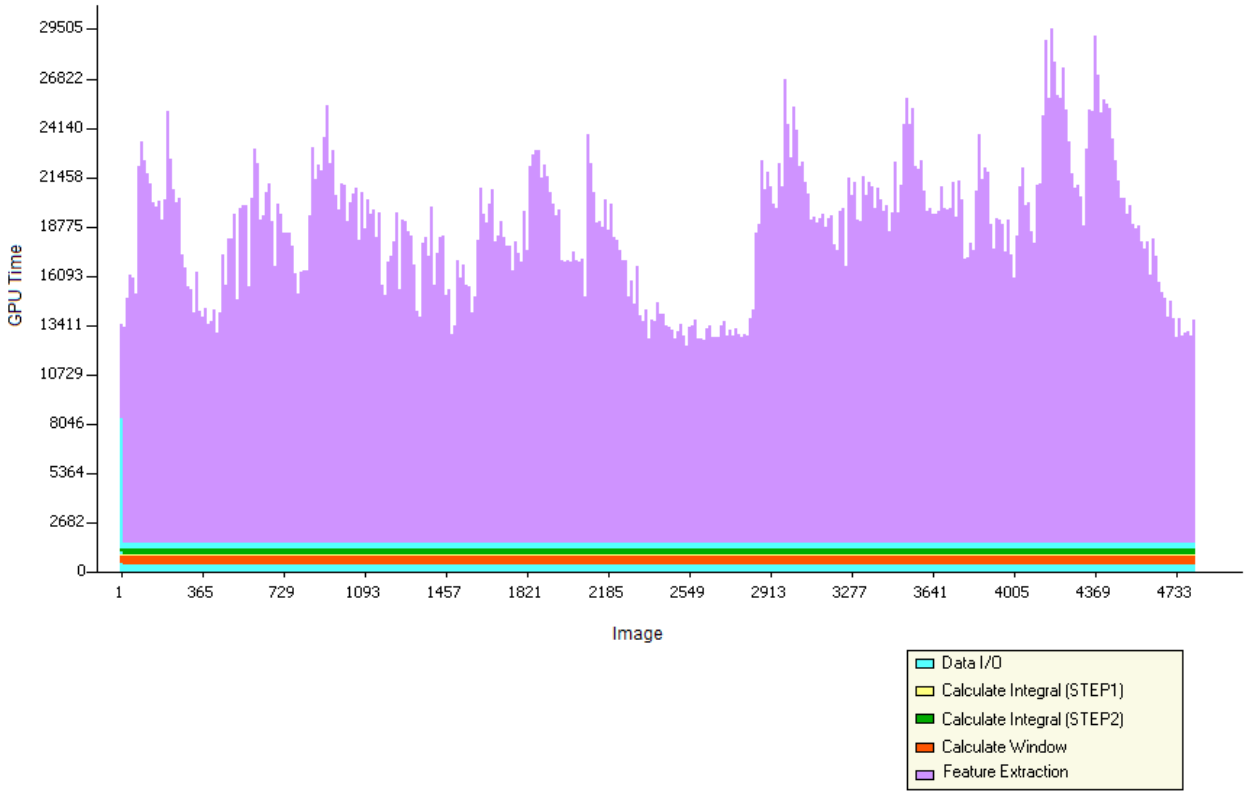

Figure 29. Object detection GPU activity time plot

3.6.1.3 Scattered reads

The amount of scattered reads and writes performed on the GPU global memory is a limiting factor.

There are basically three types of memory reads (Figure 30):

- Quads and Triangles
- Line Segments
- Point Clouds

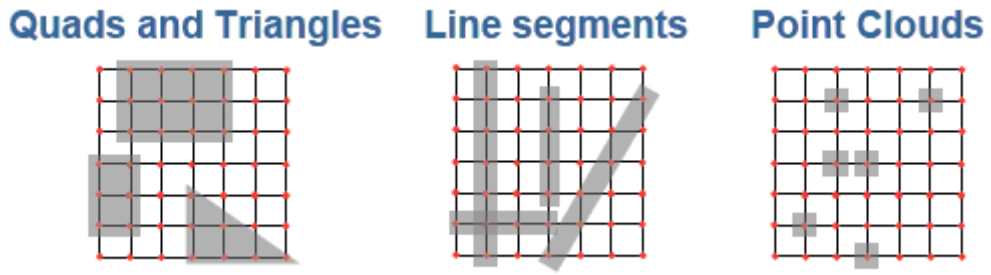

Figure 30. Three types of memory reads

The object detection utilizes mainly point clouds as type of memory reads, limiting data throughput. This is due to the locality and alignment of the data. The memory bus width is 512 bits, if a single floating point value is fetched from the memory, only 32 bits are required. Thereby only 1/16 of the memory bus width is effectively used. The measured maximum bandwidth throughput for the object detect function is 30 GB/s, this is only ~20% of theoretical maximum available memory bandwidth of 158 GB/s. An important limiting factor is the memory latency. From every fetch from the memory a small delay is introduced before the memory is accessed.

3.6.2 Future optimizations

There are two possible approaches for further optimization of the application:

- Front-end, change input data (easier)
- Back-end, change algorithm (harder)

The two ways differ in difficulty and approach and can even be combined to achieve a better optimization.

Front-end optimization

This approach is the easiest way to implement. The problem of the current implementation of the object detection is that each scale is being applied on the whole image. Most of the comparisons are redundant, because in the perspective of the image, the objects will never have the scale size in certain areas. By excluding parts of the image for certain scales, the runtime can be reduced dramatically and is most effectively for smaller scale factors.

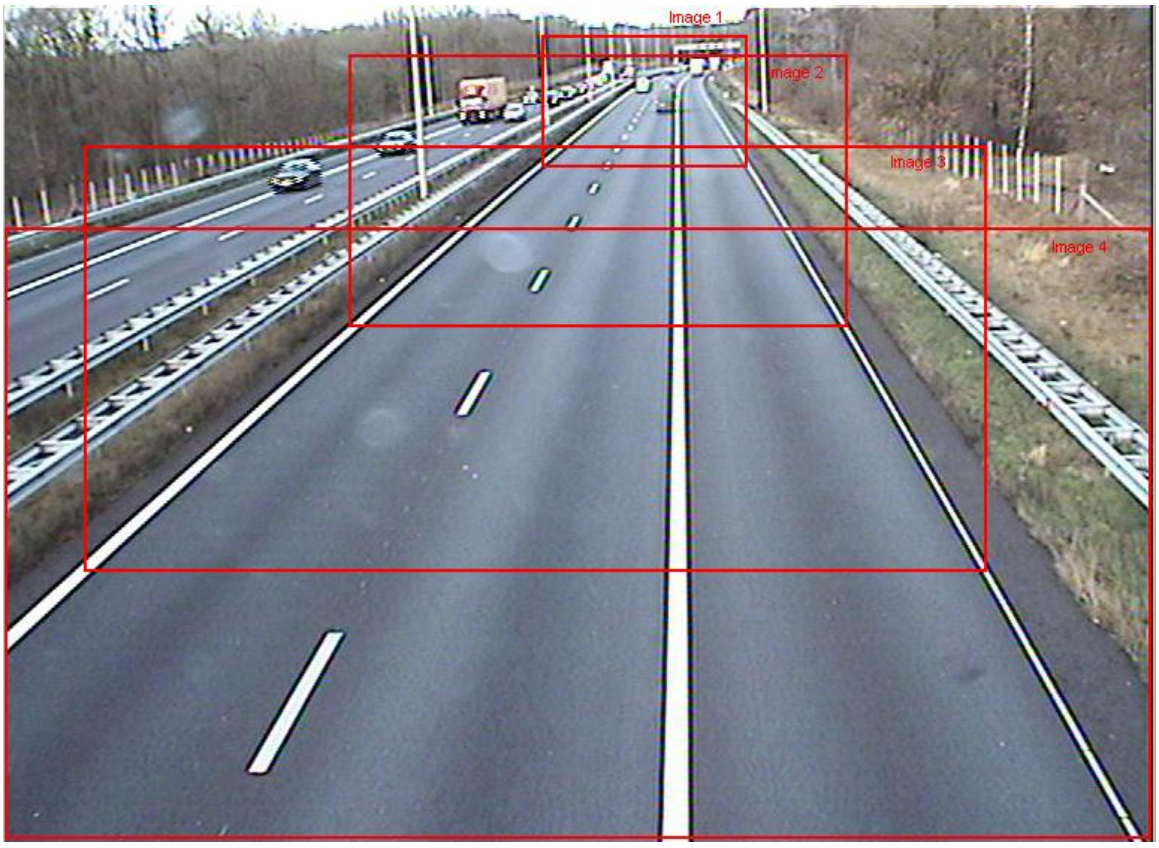

Figure 31. Example on how the image can be split in multiple images for various scales.

An opportunity is splitting the original image into several smaller fragmentary images. Each fragment receives a range of scales applied at that part of the image. This way, the amount of detections can be reduced drastically, without the needs to change the current object detect function.

Figure 31 shows how these images can be split and a scale range assigned:

Image 1: Scale ≤ 4
Image 2: $4 <$ Scale ≤ 10
Image 3: $10 <$ Scale ≤ 15
Image 4: $15 <$ Scale ≤ 20

Back-end optimizations

One of the most effective optimizations could be done by stripping the current detect object kernel to minimal functionality. Stripping concerns, removing the two for-loops around the inner-function, leaving only the feature extraction (Figure 20). This makes the kernel much smaller and resolves three problems, discussed in Section 3.6.1. The amount of registers will be reduced drastically, the amount of branching will be reduced greatly and the memory can be read as line segment instead of point clouds. The smaller kernel is executed very often and since branching for the decision tree is done on the CPU the amount of data communication between the GPU and the host has to be reduced as much as possible. The kernel performs a single feature extraction

and applies this to the whole image. This way the image data can be read in a more optimized fashion, as horizontal line segments. This results in a higher memory bus occupancy, which leads to an increased performance.
This optimization will be very time consuming to implement. The reason that this is not done immediately, is because the basic plan was, to stay as close as possible to the initial algorithm. As expected, the mentioned implementation will also stay close to the initial algorithm.
To realize this implementation with another 300 man-hours must be reckoned.

Pipelining

Pipelining is a method where an operation, having multiple subtasks, is arranged in such way that subtasks can be executed independently. In this way, it is possible that subtasks do not stall others, resulting in a speedup. Analyzing Figure 32 where the occupation of the GPU is shown, it can be concluded that the occupation is very low (~50%). A higher occupation of the GPU is possible by implementing pipelining.

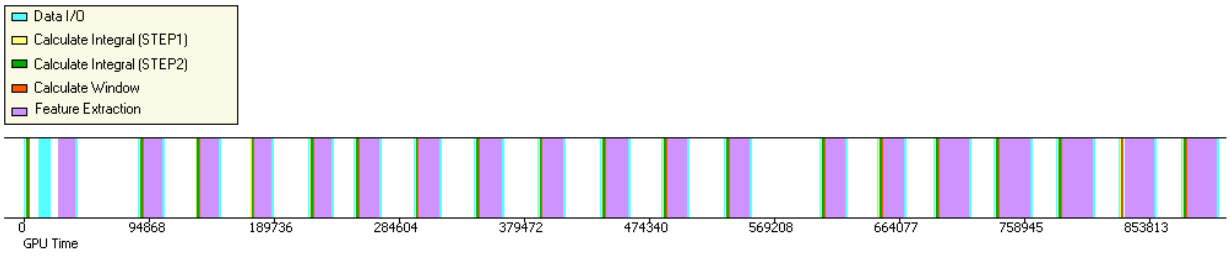

Figure 32. Occupation of the GPU over time

The detect object operation can be split in two subtasks, read image and detect object.
Figure 33 shows the increase of occupation of the GPU to almost 100%, by creating two threads. The first thread initializes and reads the first image and starts the object detection.
When the detection of the first thread is started, a second thread is reading the next image and waits until the first thread completes its detection. When this is done the second thread starts the detection and the first thread is reading the next image. This process continues until the last image is processed.

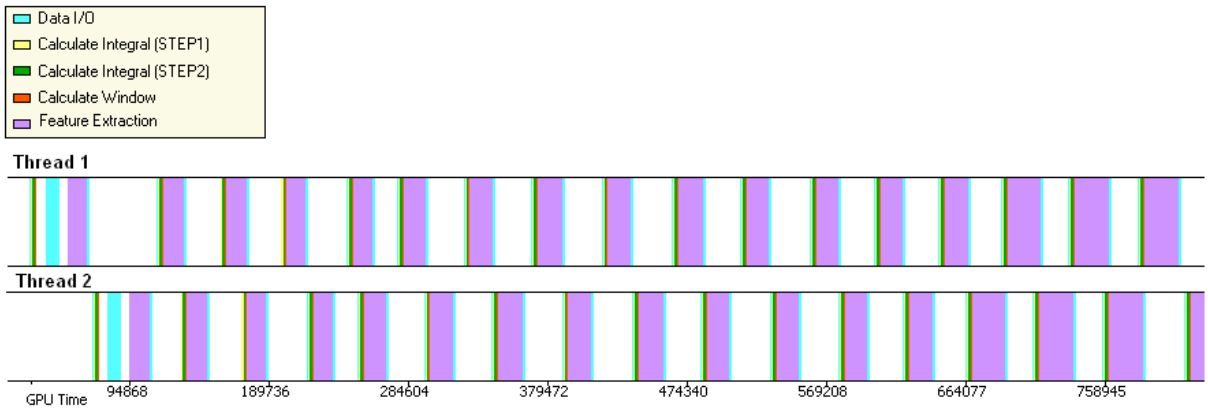

Figure 33. Pipelined object detection with two threads

Multiple GPUs
If there are multiple GPUs available, the image can be divided in multiple parts to be processed by these GPUs. Each GPU processes a part of the image and returns results from the detection. The easiest way to do this kind of implementation is to send the entire image to all GPU's. Doing so, no maximum overlapping part has to be calculated for each image window.

3.6.3 Development process

The development phases of the object detection implementation on the GPU are shown in figure 34. It can be clearly seen that the first phase (I), where the code was converted and analyzed to provide the right data for the GPU, is a very time consuming phase, taking ~40% (125 h.) of the total development time. The main reason for this large amount is due to the size and complexity of the code. In the next phase (II) the functions were implemented on the GPU. In this phase a small speedup was derived in relation to the reference version (~2x). The development time of this phase took ~30% (95 h.) of the total development time. The total time required to achieve this result (2x speedup) was about 220 hours. It would be a waste of time to stop and accept this result. This is why an optimization phase was inserted, to gain some serious results from the invested development time. In the optimization phase an increased speedup, compared to the previous implementation phase, can be determined. This speedup is the result of a reduction of communication overhead and a higher grade of parallelization. This increase justifies the amount of time required for this phase (80 h.), since it boosts the overall speedup to 4x. So, the total development time took 300 hours for a 4x speedup. Analyses show that the CUDA-enabled GPU used for this use case is still performing far below the theoretical maximum.
If the trend line of the development progress is extrapolated, a still growing line can be seen. This may indicate that there is still some room to push up the performance of the object detection application.

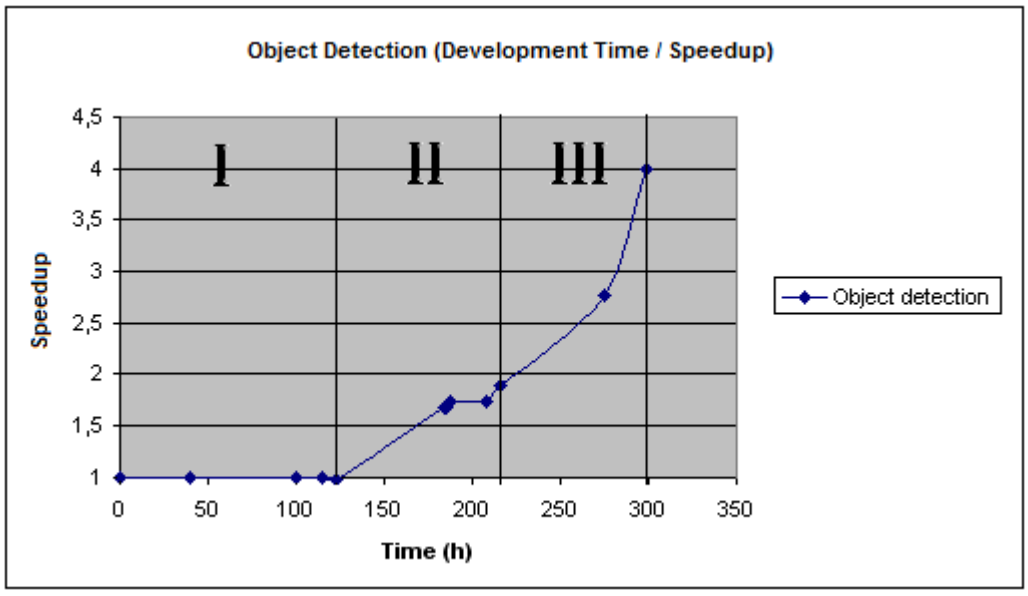

Figure 34. Implementation process divided in three development phases

3.7 Summary

In this chapter the implementation of object detection on the GPU was discussed. Results show an overall speedup of 4 times (Figure 34). The total development process took 300 hours, divided in three phases. Each phase was analyzed to determine the steps for the next stage and to evaluate the whole process. The algorithm was analyzed to find the limiting factors. Three reasons were found: In the use of registers, decision tree and scattered reads. For this problem a solution is given to avoid the limitations and achieve a higher speedup. This by changing the way the algorithm is parallelized. The main drawback of this solution is the time required to apply the changes. Also other optimizations were discussed, estimated to require less development time. Finally the development process of the object detection case was discussed.

Chapter 4: Case Study: UMASIS

In this chapter the case study being discussed is the UMASIS project. UMASIS is an application, simulating ultrasonic measurements, developed to simulate inspections on material.
The algorithm is based on Finite Difference Time Domain (FDTD) calculations [41], which are very computationally intensive and can generate a lot of data. In some cases UMASIS requires several days of computation to process the data, although it is installed on a Cray XD1 supercomputer. This Cray XD1 can execute multiple UMASIS jobs in parallel, but the aim for this case study is to execute a single UMASIS job in parallel.

In Section 4.1 a short introduction on the theoretical principles behind UMASIS is given. In Section 4.2 a functional breakdown is given so that the profiling can be done in Section 4.3. The first implementation of the code on the GPU will be discussed in Section 4.4. In Section 4.5 the results are analyzed, intended to optimize the code. In Section 4.6, the analysis of the development process and results are discussed.

4.1 UMASIS

For many years, TNO has developed state of the art knowledge on ultrasonic technology.
For more than one decade, TNO has successfully applied modeling software to simulate the propagation of ultrasonic waves through various media. UMASIS is an intuitive and accurate software tool allowing users to perform simulations of ultrasonic wave propagation. Erroneous interpretation leads to false calls or defects being missed. By making use of simulation the insight, reliability and availability are improved and costs are reduced. Ultrasonic simulation tools play an important role in improving the understanding of the measured signals and optimizing the inspection geometry. Furthermore, since issues as safety, environment and health are becoming more important, there is a growing demand for inspection to prevent failures from happening.

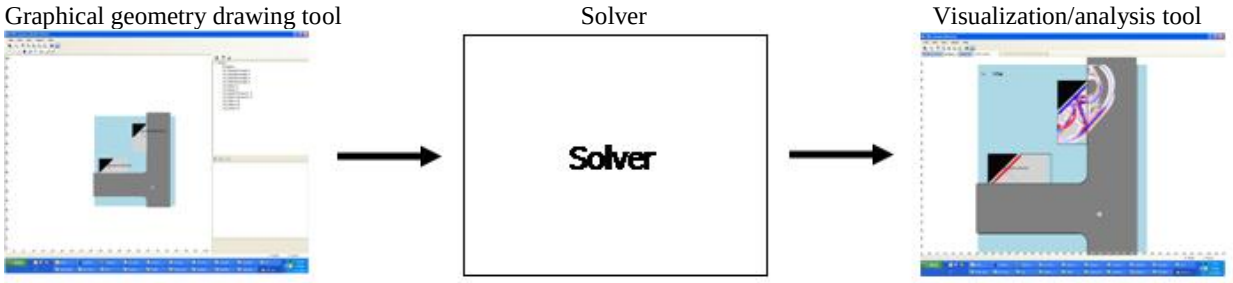

Figure 35. UMASIS tools

UMASIS consists of a (Figure 35)

- Graphical geometry drawing tool
- Solver
- Visualization/analysis tool of simulation results

4.1.1 Graphical geometry drawing and visualization/analysis tool

The visualization of wave propagation has proven to give new insights. Ultrasonic inspection is a method to perform Non Destructive Testing (NDT) and is widely applied in industries as Oil & Gas, Aerospace, Process Industry and Nuclear Industry. For ultrasonic inspection, transducers are used to emit and receive ultrasonic waves that propagate through the medium under inspection. Defects or discontinuities in the material disturb the wave propagation, resulting in a complex signal at the receiver. UMASIS is used to get a reliable estimate of the geometrical behavior. The results of the simulations can be visualized and analyzed in the user interface. Pictures and movies can be exported for reporting and presentation purposes. For visibility optimization, the graphical presentation can be adjusted by the user.

4.1.2 Solver

UMASIS is a software tool that uses the power of wave equation based algorithms to calculate the full elastic wave field. During the calculation, the propagation of ultrasound and complex phenomena such as reflection, refraction, diffraction and wave conversion are all automatically taken into account. In this way, UMASIS not only calculates the responses at the receivers, but is furthermore able to visualize the propagation of waves through the inspection object.

The base of the solver is the Finite-Difference Time-Domain (FDTD) modeling technique [36]. The main equation consists of a set of coupled partial differential equation which describes the wave propagation through a 2-D medium.

$$\rho \frac{\partial u_t}{\partial t} = \frac{\partial \tau_{xx}}{\partial x} + \frac{\partial \tau_{xz}}{\partial z}$$

$$\rho \frac{\partial w_t}{\partial t} = \frac{\partial \tau_{zx}}{\partial x} + \frac{\partial \tau_{zz}}{\partial z}$$

Equation 9

$$\tau_{xx} = (\lambda + 2\mu) \frac{\partial u}{\partial x} + \frac{\partial w}{\partial z}$$

$$\tau_{zx} = \mu \left(\frac{\partial u}{\partial x} + \frac{\partial w}{\partial z} \right)$$

$$\tau_{zz} = (\lambda + 2\mu) \frac{\partial w}{\partial z} + \frac{\partial u}{\partial x}$$

Equation 10

The two main equations are the equations of motion (Equation 9) and the constitutive laws for an isotropic medium (Equation. 10). The *u* and *w* are the particle velocities, the *txx*, *txz* and *tzz* are the elements of the stress tensor. The ρ is the mass density and the λ and μ are lamé parameters.

These equations are numerically solved in a staggered grid by means of a 1st order time derivative and a 4th order spatial derivative [41]. This means that for each time step the particle velocity and the stresses are calculated consecutively. Each update of one set of parameters is determined by a weighted sum of the spatial derivative of the other parameters. By means of a Taylor expansion, the spatial derivative can be calculated efficiently with a convolution operator (Figure 36). For a staggered grid, the coefficient of the derivative operator are [1/24 -9/8 9/8 -1/24].

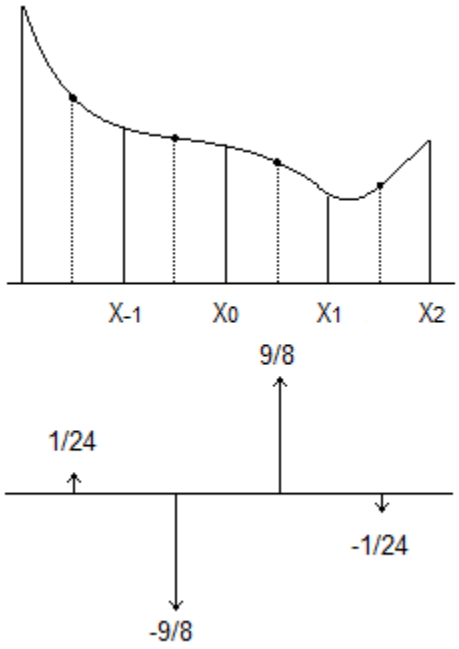

Figure 36. Determination of the spatial derivative (da/dx) at x0 by means of a 4th order convolution operator.

4.2 Application

The UMASIS solver is the part being implemented on the GPU. This is where all heavy calculations are performed for the ultrasonic simulation. It consists of various steps that are required to be performed for the simulation. In Figure 37, the steps and their mutual relationships are shown.

The input of each step depends on the output of the previous step, so that steps have to be calculated subsequently. The first step initializes a file in which the model is defined, followed by the initialization of pre-required data meant for properly use in the calculations.

The main steps for the simulations are within a loop that loops thru a couple of steps, till all the calculations are performed. Each step of the for-loop is a time step. The data of the final result is stored after all steps are being processed.
The main calculations of the solver are that concerning velocities and stresses (Table 8). They both consist of a double for-loop where in the calculations are being performed (Figure 38 and 39). The velocities are calculated in two directions, where the stresses are calculated in three directions. The calculations of the velocities and stresses are convolutions and are very appropriate to be parallelized.

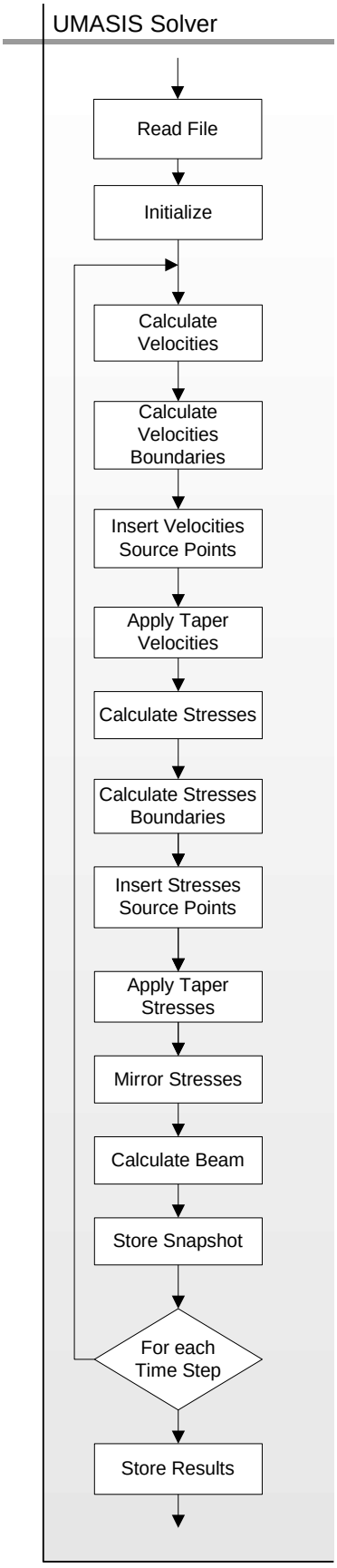

Figure 37. Flow chart UMASIS solver

4.2.1 Calculate velocities / stresses

The calculations of the velocities and stresses are the main calculations of the solver. They both consist of a double for-loop where within the calculations are being performed (Figure 38 and 39). The velocities are calculated in two directions where the stresses are calculated in three directions. The calculations of the velocities and stresses are convolutions and are very appropriate to be parallelized.

```
Vx(x,y) = Vx(x,y) +
            ( dt*(9/8)   / dx) * B(x,y) * (Txx(x,y)    - Txx(x,y-1)) +
            (-dt*(1/24)  / dx) * B(x,y) * (Txx(x,y+1)  - Txx(x,y-2)) +
            ( dt*(9/8)   / dz) * B(x,y) * (Txz(x,y)    - Txz(x-1,y)) +
            (-dt*(1/24)  / dz) * B(x,y) * (Txz(x+1,y)  - Txz(x-2,y))
```

Calculation of the velocities in horizontal (Vx) direction, where the input Txx, Txz and are the directional stresses

```
Txz(x,y) = Txz(x,y) +
   ( dt*(9/8) / dz) *M(x,y) * (Vx(x+1,y) - Vx(x,y))    +
   (-dt*(1/24)/ dz) *M(x,y) * (Vx(x+2,y) - Vx(x-1,y)) +
   ( dt*(9/8) / dx) *M(x,y) * (Vz(x,y)    - Vz(x,y-1)) +
   (-dt*(1/24)/ dx) *M(x,y) * (Vz(x,y+1) - Vz(x,y-2))
```

Calculation of the stresses diagonal (Txz) direction, where the input Vx, Vz are the directional velocities

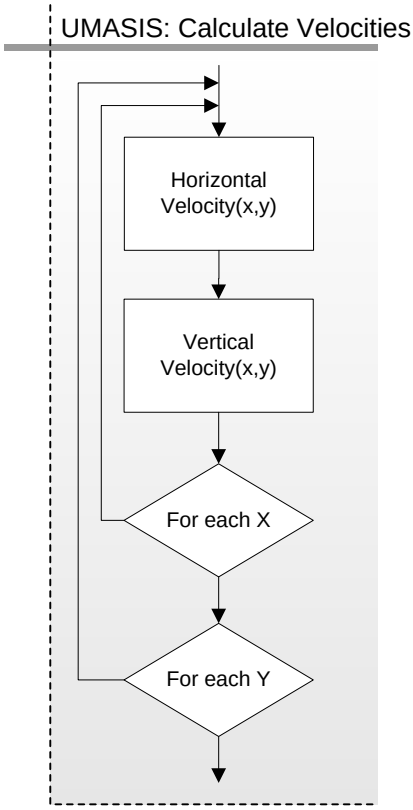

Figure 38. Flow chart calculate velocities

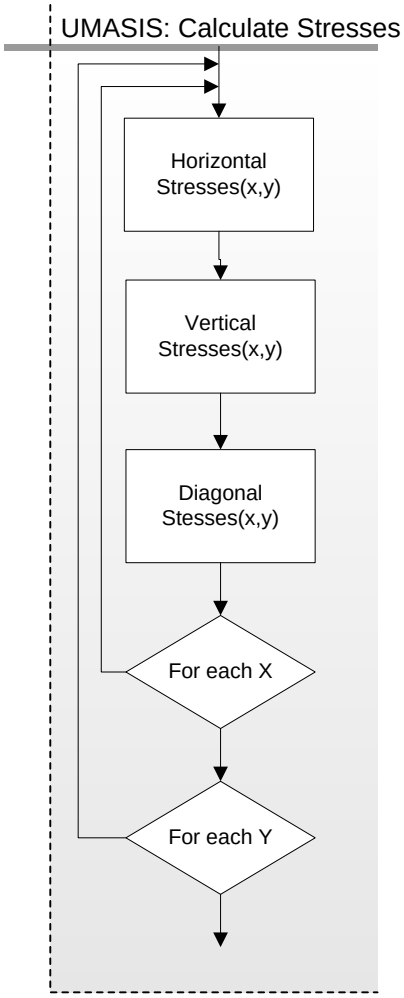

Figure 39. Flow chart calculate stresses

4.2.2 Calculate boundaries velocities / stresses

The boundaries of the velocities and stresses (Figure 40) are calculated in a different step. First the upper and lower horizontal boundaries are calculated, then the left and right vertical. The function of the velocities and stresses are differing, but they use the same scheme. Different is the amount of directions calculated and the input data within the calculate boundary (x,y) function. The edges for the velocities and stresses are being calculated after the calculation of the horizontal and vertical boundaries.

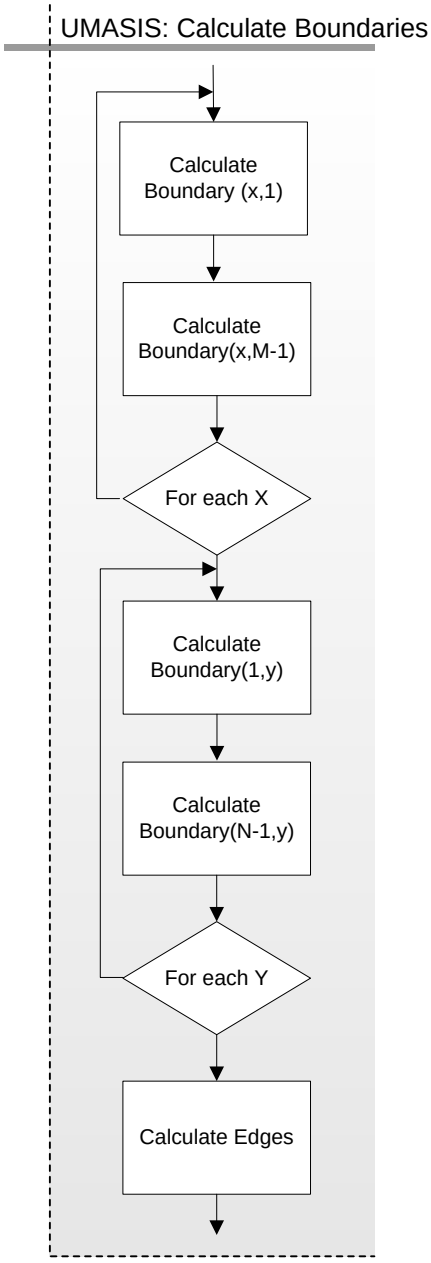

Figure 40. Flow chart calculate boundaries

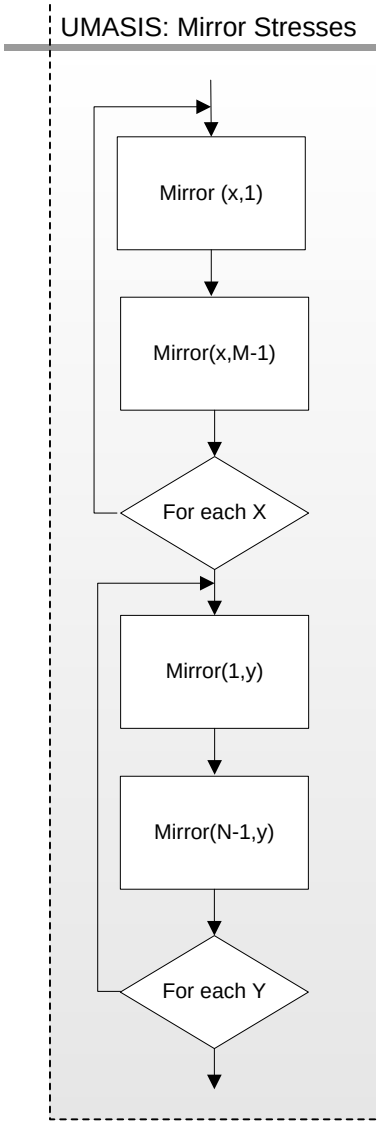

Figure 41. Flow chart mirror stresses

4.2.3 Mirror stresses

This step is only used to mirror the stresses (Figure 41) on the outer boundaries of the plain. First the most upper and lower horizontal stresses are mirrored, then the most left and right vertical.

```
Txz(1,y)    = -Txz(2,y)
Txz(N-1,y) = -Txz(N-1,y)
Txz(x,M-1) = -Txz(x,M-1);
Txz(x,1)]   = -Txz(x,2);
```

Mirror stress calculations of all the borders

4.2.4 Calculate beam

The beam calculation (Figure 42) concerns the product of stresses and velocities, resulting in a spatial distribution of the total energy over time. This step consists of a double for-loop where each point on the plain is calculated. The beam step uses point operation which is expected to be very well applicable for parallelization.

```
Beam(x,y) = Beam(x,y) +
          sqrt(
            (Tzz(x,y)*Vz(x,y) + Txz(x,y)*Vx(x,y))^2 +
            (Txx(x,y)*Vx(x,y) + Txz(x,y)*Vz(x,y))^2
              )
```

Beam calculation

4.2.5 Apply taper velocities / stresses

The apply taper step (Figure 43) applies absorbing boundaries to prevent reflections on the boundaries. In that way a semi-infinite medium is created. The flowchart is very similar to the beam calculation. This step consist of a double for-loop where within the tapering is applied. This step differs in function for velocities and stresses in the amount of directions and input data. The tapering step uses point operation which is expected to be very well applicable for parallelization.

```
Txx(x,y) = Txx(x,y) * B(x,y)
Tzz(x,y) = Tzz(x,y) * B(x,y)
Txz(x,y) = Txz(x,y) * B(x,y)
```

Apply taper stresses calculation

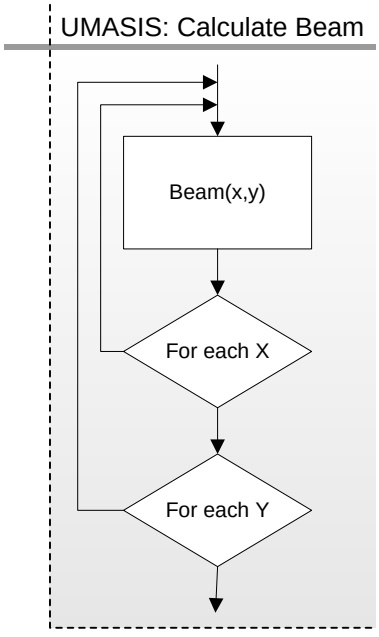

Figure 42. Flow chart calculate beam

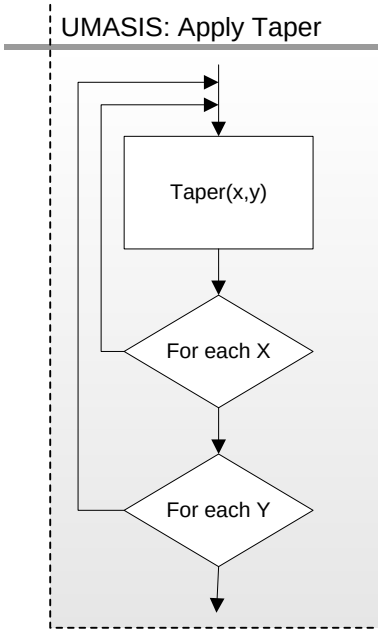

Figure 43. Flow chart apply taper

4.2.6 Insert source points velocities / stresses

This step inserts velocities or stresses (Figure 44) to certain points on the plain. These points are pre-defined in the input data file. This step differs in function for the stresses and velocities in the amount of directions and source points data.

```
gq = Amp(point) * exp( -alpha * (tk - t0 - Tau(point)) ) * dt;
gf = -2.0 * alpha * (tk - t0 - Tau(point)) * gq;

Txx(Zs(point),Xs(point)) = Txx(Zs(point),Xs(point)) + (stxxq*gq + stxxf*gf)
Tzz(Zs(point),Xs(point)) = Tzz(Zs(point),Xs(point)) + (stzzq*gq + stzzf*gf)
Txz(Zs(point),Xs(point)) = Txz(Zs(point),Xs(point)) + (stxzq*gq + stxzf*gf)
```

Insert source points stresses calculation

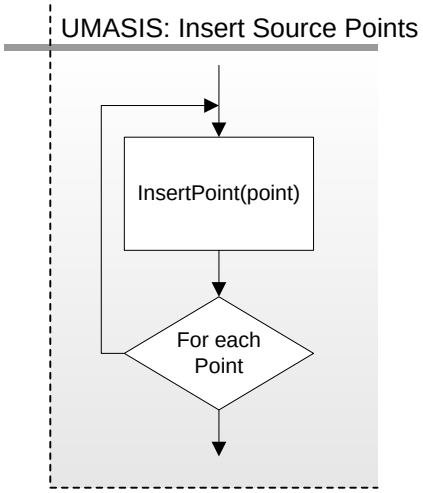

Figure 44. Flow chart insert source points

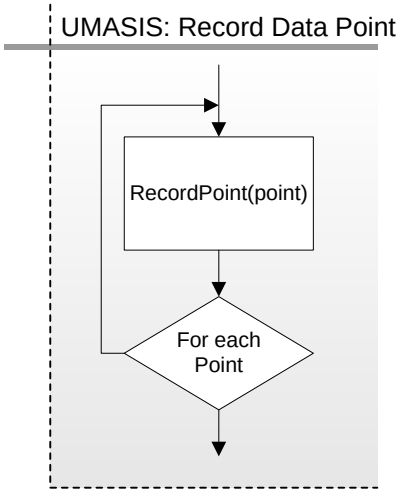

Figure 45. Flow chart record data points

4.2.7 Record data points

This step is meant to save intermediate results (Figure 45). The frequency of saving the intermediate results is predefined in the input data file. The function is comparable with the insert source point step. A selection of predefined points are being recorded and stored on the computer.

```
recPoint = (kRec*iNrPointsRec) + point

RecVx(recPoint)  = Vx(Zr(point) , Xr(point))
RecVz(recPoint)  = Vz(Zr(point) , Xr(point))
RecTxx(recPoint) = Txx(Zr(point), Xr(point))
RecTzz(recPoint) = Tzz(Zr(point), Xr(point))
RecTxz(recPoint) = Txz(Zr(point), Xr(point))
RecPhi(recPoint) = Phi(Zr(point), Xr(point))
RecPsi(recPoint) = Psi(Zr(point), Xr(point))
```

Record data point function

4.3 Profiling

As for the other case study, a profile of the application is needed to get information for the first implementation of the application to the GPU. The application is split up in multiple steps, as defined in Chapter 4.2, determining the time needed to execute each specific step.
The model used for the profiling and for all the other benchmarks is a simple model (Figure 46) with one transducer and one object. The size of the model is 404 x 593 elements.

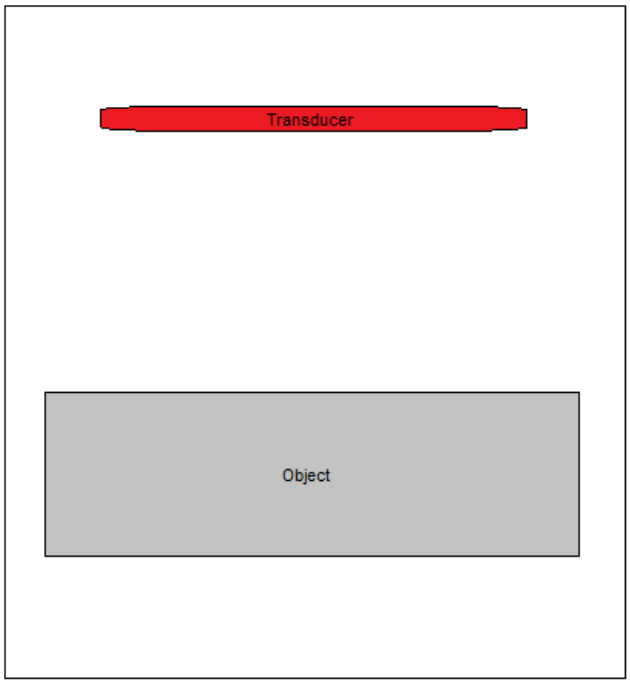

Figure 46. Simulation model

4.3.1 Specifications of the reference PC

For the profiling a personal computer (PC) is used to execute and measure the runtime of the UMASIS application. The PC also serves as reference for the performance comparison between the GPU and the PC. In Table 3 the specifications of the system are shown. The results of the profiling are based on a single-cored execution of the application on a PC.

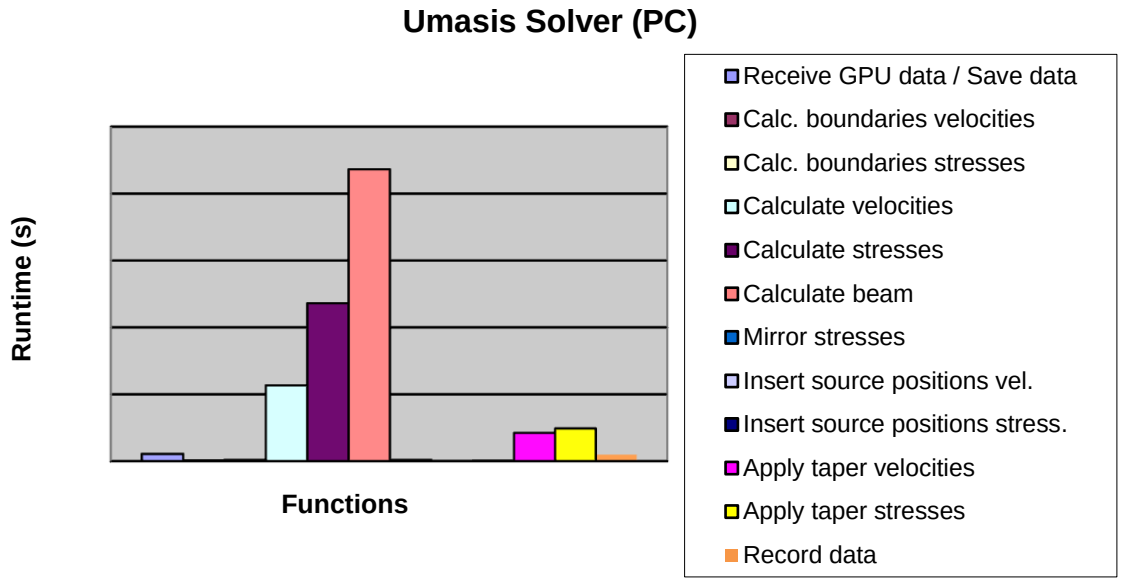

Figure 47. Profiling results

Table 8. Profiling Results

UMASIS solver loop		
Function name	Runtime (s)	Total %
Calculate velocities	11.323	13
Calculate boundaries velocities	0.100	<1
Insert source points velocities	0.054	<1
Apply taper velocities	4.226	4
Calculate stresses	23.608	24
Calculate boundaries stresses	0.166	<1
Insert source points stresses	0.057	<1
Apply taper stresses	4.884	5
Mirror stresses	0.042	<1
Calculate beam	43.639	48
Record data	0.960	1
Store data	1.032	1
Total	90.091	100%

4.3.2 Profiling results

From the results in Table 8 and Figure 47, it can be determined that there are three parts that take up to 85% of the total runtime. These parts are the calculation of the beam (48%), stresses (24%) and velocities (13%). As can be learned from Chapter 2.3.5 (Amdahl's law), these parts have to be concentrated on, to optimize for parallelization and achieve a significant speedup.
Other less significant, but notable steps, are the tapering functions for the stresses and velocities. The other parts, take less than 1% of the total runtime, summed up a 6% of the total runtime.

To realize serious time reduction, it is clear that the more significant parts have to be the primary target. However all the steps are being implemented on the GPU, because otherwise a lot of extra communication overhead would be generated. If this is not taken in account the amount of overhead generated by extra communication could become significant. This is why all steps are integrated on the GPU in order to realize a total solution.

4.4 Realization

In this paragraph the realization of the implementation of the GPU is discussed. The steps defined in de profiling are analyzed and ported to the GPU. Implementation results are analyzed and discussed.

4.4.1 Sending data

One function is created to send all the essential data to the GPU. This function is called before the main loop and has to be executed once. This is possible, because all calculation are being performed on the GPU. Hereby a great deal of communication overhead can be avoided that otherwise would be generated by data, swapped between the host and the GPU.

4.4.2 Receiving data

There are three functions created to receive data from the GPU:

- Intermediate results.
- Beam results.
- Final results.

These functions are called on different occasions in time, providing the data required at that moment. Hereby communication overhead can be reduced as much as possible.

4.4.3 Parallelization of the calculation of the beam and tapering

The main function of calculating and tapering of the beam, velocities and stresses, consists of a double loop, without dependencies between the input and the output. Therefore all the elements in the double loop can be performed in parallel. If the size of the problem is O(N*M), the above calculations, with P(N*M) processing elements, can be performed in T(1) time steps. This makes these functions very appropriate for parallelization. The velocities are calculated in two double loops the stresses in three and the beam only one. Each double loop is mapped on the available processing elements of the GPU and is executed in parallel for each direction. The data required for the calculations is fetched from the global memory. The conversion of the code was mainly pointing the memory pointers to the right place of the GPU global memory. Also some code to calculate the planar space position, by converting the block- and thread-id to a x-y coordinate, had to be added.

4.4.4 Parallelization of the boundaries and mirroring

The calculations for the boundaries of the velocities and stresses and the mirroring for the stresses are all processed in a similar way. The calculation consists of 4 single loops, so that each loop loops through in a direction on the plane. The results of the calculation do not depend on the input or output and so it can be parallelized. The problem size is O(2*N + 2*M) and can be calculated with P(2*N + 2*M) processing elements in T(1) time. The velocities and stresses

require a calculation in respectively two and three directions. The boundary calculations for velocities and the stresses cannot be done concurrently and has to be done sequentially. The data required for the calculations is fetched from the global memory. The conversion of the code was mainly pointing the memory pointers to the right place of the GPU global memory. Also some code to calculate the planar space position, by converting the block- and thread-id to a x-y coordinate, had to be added.

4.4.5 Parallelization of insert sources points and record data

The insert source points consist of a single for-loop with no dependencies between the calculations. So, the calculations can be performed with P(N) number of processors in T(1) time instead of T(N) time. The same thing can be done with the record data step. The data required for the calculations is fetched from the global memory. The conversion of the code was mainly pointing the memory pointers to the right place of the GPU global memory. Also some code to calculate the planar space position, by converting the block- and thread-id to a x-y coordinate, had to be added.

4.4.6 Realization results

The implementation of UMASIS is performed on a CUDA-enabled GPU from NVIDIA.
In Table 5 the specifications of the GPU are shown. The GPU is selected for its high-end performance in combination with relatively low costs.

From the results in Table 9 and Figure 48 can be determined, that the most time consuming parts are showing the highest speedup, where the beam calculation (247x) shows the most impressive performance increase. The speedup for calculation and tapering of the velocities (28x, resp. 26x) and stresses (40x, resp. 24x) are good figures to start with. Some parts are currently running slower on the GPU then on the host. This is the trade-off, made between the communication overhead between GPU and the host computer and the execution time of smaller, possibly less parallel optimal, parts on the GPU. A major part (34%) of the runtime consists of the store data part. This is the time that is required to save all data to the hard disk. A minor part (~4%) is communication overhead from the GPU to the host, which is acceptable. The runtime, required to transfer the data to the GPU, is negligible and therefore not taken in account.
The implementation of the calculate velocities and stresses steps; do result in a speedup of respectively 28 and 40 times. The stresses calculation profits more than that of the velocities, since it has an extra direction, resulting in an increased amount of calculations. In the way the memory is fetched, is still for both calculations not very efficient. Some points are redundantly fetched multiple times. This overhead will be reduced in the optimization step by using the shared memory.
The optimization of calculate beam and tapering are resulting in a speedup of respectively 247 and 24 times. The speedup result of 247 times is more than theoretically expected (Chapter 2.4.3) there can be multiple reasons for this, for example if this step is not running efficient on the reference PC. Both calculations are very similar, but the beam calculation is much more compute intensive, resulting in a much higher speed up for the beam calculation. This is very similar to what we seen at the stresses and accentuates that parallel processing profits most from parallel compute intensive calculations. The calculation of the beam and tapering do not leave much room for further optimization, because both are very straight forward to parallelize and implement. The reason for this is the high degree of independency of operations within the loop.

The calculation of the boundaries for the velocities and stresses resulted in a speedup of respectively 0.9 and 1.2 times. The runtime is for both calculations almost the same as the reference PC, both calculations are suffering from an efficient way of reading out vertical lines from memory. The compute intensity is also not very high, making parallel execution less optimal. Still it is better to perform these calculations on the GPU, because otherwise the data has to be swapped between the GPU and the host computer, resulting in an intensive communication overhead.
The speedup of the insert source points is 0.7 times, the runtime is higher than the runtime of the reference PC. The reason is that this step is not very compute intensive and therefore missing the profitability of parallel execution.

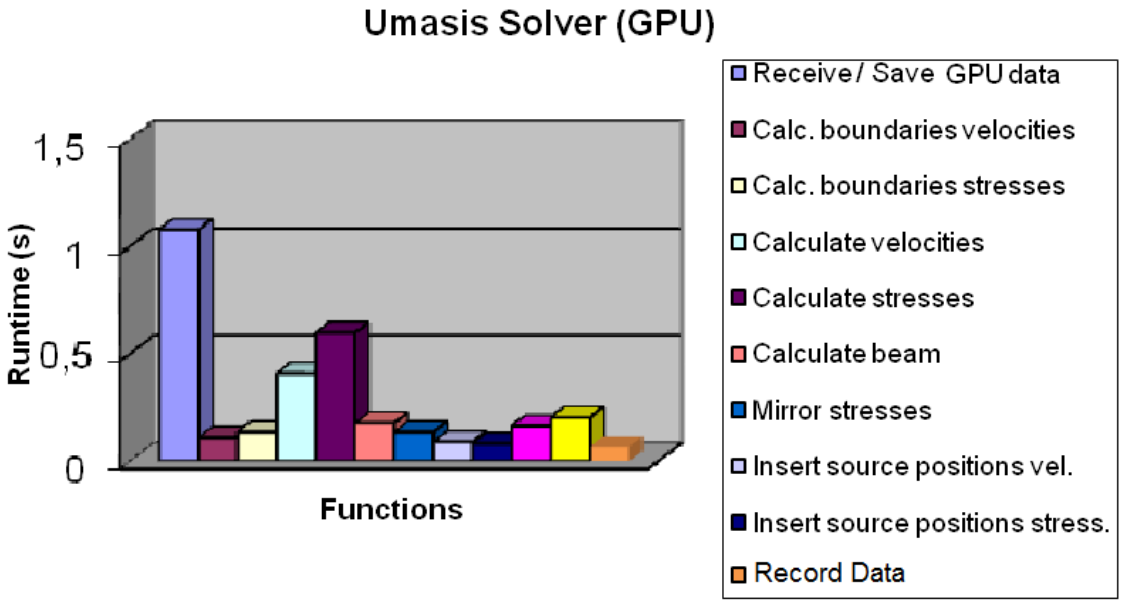

Figure 48. UMASIS realization results

Table 9. UMASIS results. The speedup is the host runtime divided by the GPU runtime

UMASIS solver loop			
Function name	runtime PC (s)	runtime GPU (s)	Speedup
Calculate velocities	11.323	0.408	28
Calculate boundaries velocities	0.100	0.109	0.9
Insert source points velocities	0.054	0.090	0.6
Apply taper velocities	4.226	0.163	26
Calculate stresses	23.608	0.598	40
Calculate boundaries stresses	0.166	0.135	1.2
Insert source points stresses	0.057	0.082	0.7
Apply taper stresses	4.884	0.203	24
Mirror stresses	0.042	0.084	0.5
Calculate beam	43.639	0.177	247
Record data	0.960	0.067	15
Store data	1.032	1.078	0.9
Total	90.091	3.194	28.2

4.5 Optimization

In this paragraph the optimizations of the implementation of the GPU are discussed. The steps are analyzed and optimized, to gain performance. Optimization results are analyzed and discussed.

4.5.1 Optimization calculate velocities and stresses

For the calculations of a single point on a plane, multiple points are required. These points are fetched from the global memory. This is the case for the velocities and the stresses. Performance would increase if parts of the global memory were copied to a faster memory space. The shared memory can be accessed much faster, but has a size limitation. Therefore the whole space has to be partitioned in blocks. These blocks copies parts of the memory to the shared memory. So, calculations could be performed on the data provided in the shared memory block and will so reduce the amount of global memory fetches, resulting in a speedup.

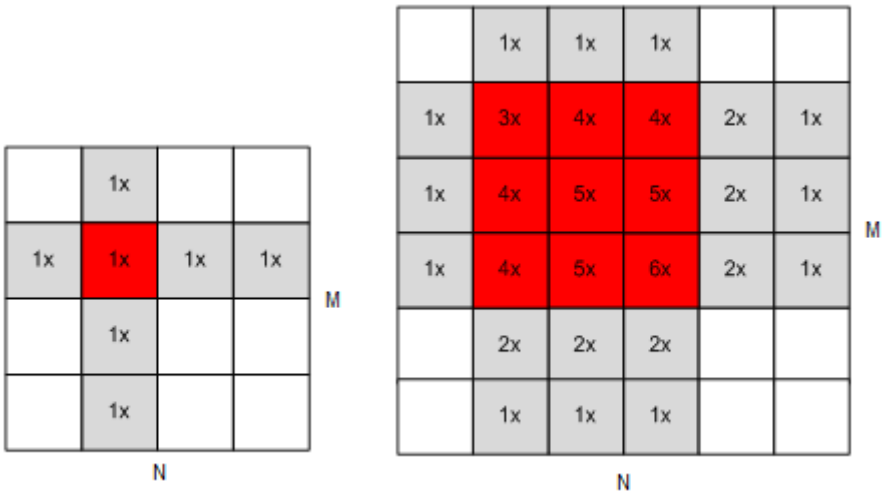

Figure 49. left: glob. mem. fetch single calculation, right: glob. mem. fetch 3x3 calculations

Figure 49 shows for single point calculations seven points which have to be fetched from the global memory. By creating blocks of threads this can be reduced. Each thread fetches a single point in the global memory and stores it to an array in shared memory, this is allocated to store all required points for all threads. The GPU aligns automatically all the horizontal aligned threads to fetch the memory as a horizontal lines. Therefor the size of M is less important than the size of N, N should not be chosen to small. For the UMASIS implementation a block size of NxN is chosen, this to reduce extra complexity in the code. A limiting factor is the maximum size of the shared memory. The CUDA Occupancy Calculator [39] can be used to define the optimal block size. Some points are still fetched multiple times from the memory, this is required on the boundaries of the blocks. Figure 50 shows how the blocks are organized and how the overlapping data is retrieved. It is clear that the amount of data to be fetched is reduced drastically having only up to three fetches on the boundaries of the block. A block size bigger is than three is needed to reduce the total memory reads as much as possible.

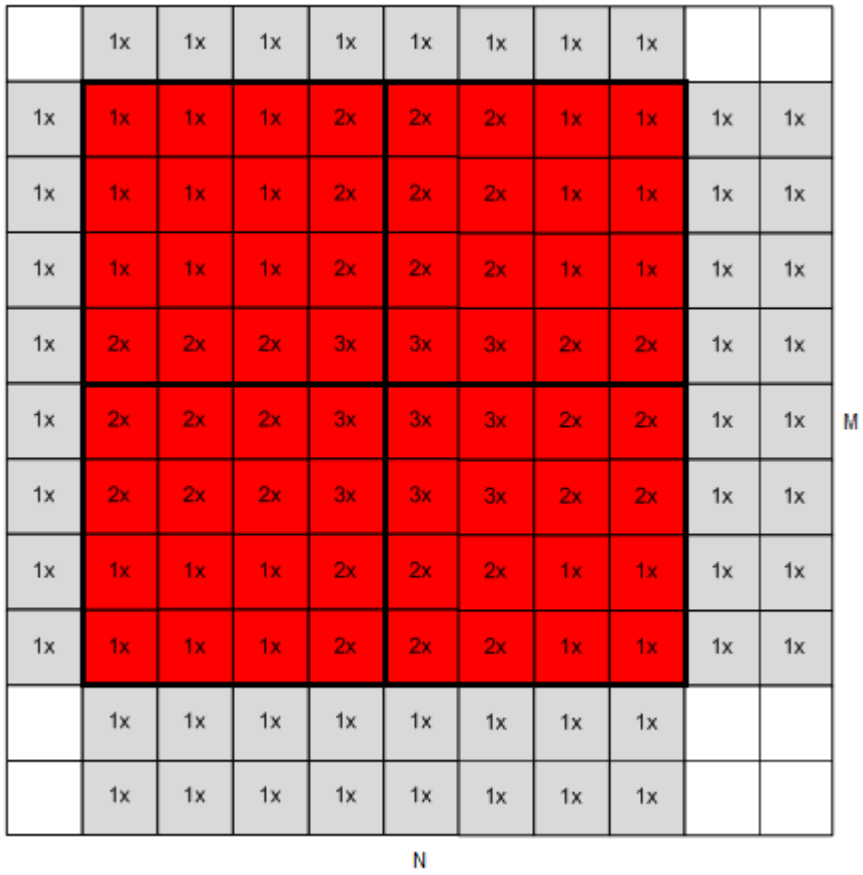

Figure 50. Global memory fetch for 4 blocks of 4x4 calculations

Another optimization performed, is the way memory is assigned. In the first implementation the shared memory variable was defined as a 2d-array, e.g. A[x][y]. This is not optimal for the performance, because of the way memory is arranged. To achieve coalesced memory reads/writes it is better to define the array as 1d-array, e.g. A[y*width + x]. The data can be fetched as coalesced read/write, resulting in a single read/write in the shared memory, so that more data points can be fetched at once. So, the amount of memory reads and writes are reduced.

4.5.2 Optimization calculate beam and tapering

The optimizations for the beam calculation and tapering are very similar. First the input values are fetched and placed into the core register, continuing until all fetches are performed. Then results are calculated and saved to their corresponding places in the global memory.

4.5.3 Optimization boundaries velocities and stresses

The optimization for the boundary calculations of both the velocities and stresses are mainly done by combining kernels. The previous calculations were performed in two kernels each for a different direction (horizontal, vertical). Combining these directions from two to a single kernel, some parts of the code can be merged and have to be calculated once. Some values are fetched multiple times from the global memory. The performance was boosted by sharing variables among other threads.

4.5.4 Optimization insert source points and record data

The optimization for the insert source points and also for the record data lies in minimizing the amount of operations. Some values were fetched multiple times from the global memory. By reducing these redundant operations a speedup gain was realized from these steps.

4.5.5 Optimization results

Table 10 and Figure 51, show that the optimizing step result in an overall improved speedup. The results from the optimization compared to realization show that the overall speedup is smaller for the optimization step (max. 2.6x), however almost all steps are faster than their serial counterparts (except for the store data). From the results it can be determined that the least profitable parts from the first implementation gains most from the optimization process.
The optimization of the calculate velocities and stresses steps are resulting in a speedup of respectively 1.6 and 1.5 times. It seems that the communication overhead, which this optimization mostly reduced, was not the most significant part of the steps. The optimization introduced additional code being executed to arrange the optimization and resulting in a larger kernel. Also the allocation of the block size was altered, to benefit from the optimization and resulting in less optimal usage of available threads, these threads are only used for reading the borders. It is expected that there is still some room to organize the optimization in other ways, resulting in a more profitable optimization.
The results from the optimization calculate beam and tapering are resulting in a speedup of respectively 1.4 and 1.5 times. This seems small, but is quiet some speedup on top of the high speed up gained from the first implementation. Also the optimization changes made for these steps were small because there was less room to gain extra performance.
The optimization of the calculation of the boundaries for the velocities and stresses resulted in a speedup of respectively 2.4 and 2.6 times. These were the highest speedups gained from the optimization. In the previous implementation, both steps have similar performance as the reference PC, now the runtime is shortened and significantly faster than the reference PC. The optimization for the boundary calculations of both velocities and stresses, are mainly done by combining kernels. Therefore some variances only have to be fetched from the memory once.
The speedup generated from the optimization of insert source points, in comparison to the previous implementation, is 2.4 times. That way, the total runtime of this step is below the runtime of the reference PC. This result is mainly generated by reducing the number of memory fetched, but there is not much room to realize further optimization. Because of the fact that this step is not very compute intensive it is therefore more difficult to profit from a parallel execution on the GPU. For this purpose a CPU is able to execute the code very fast in serial and this is also the case for the boundaries calculations.
The most significant step is the store to disk data step taking ~45% of the total runtime of the UMASIS solver. It will not be easy to reduce this step any further as it is mostly write time to the hard disk. Reduction should be found in compression of data, requiring extra computations, or improved hardware, e.g. a faster (solid state) hard disk.

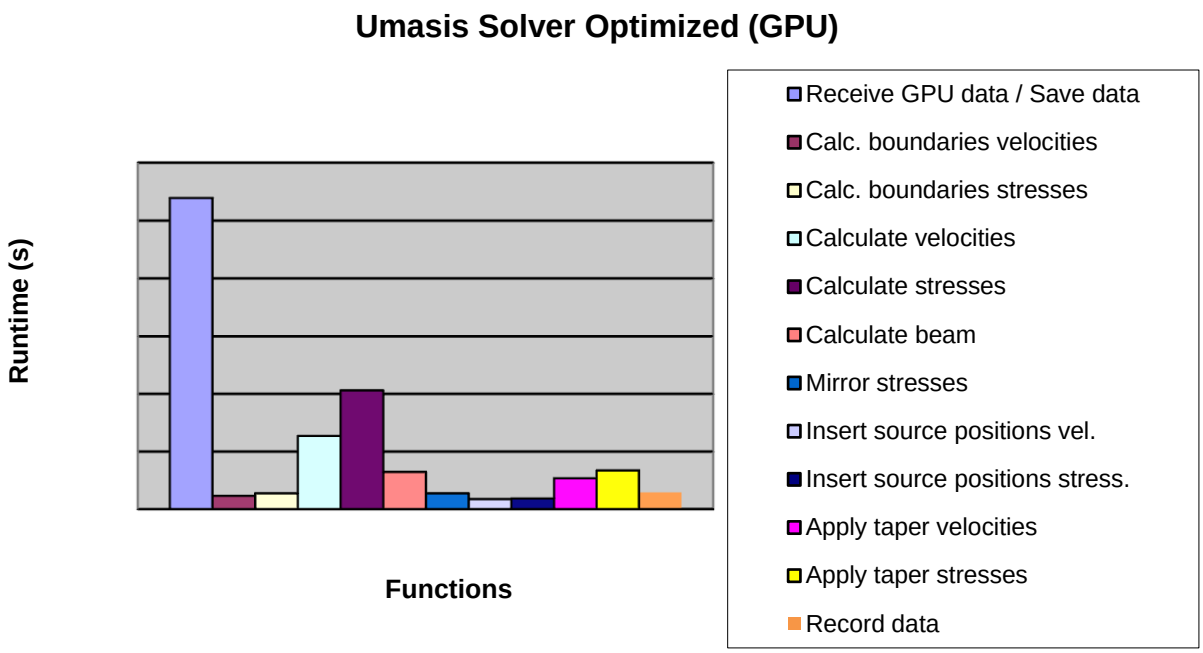

Figure 51. Optimization results UMASIS

Table 10. Opt. Results. The Tot. speedup is the host runtime divided by the GPU runtime. The Opt. speedup is the GPU runtime of the previous step divided by the GPU runtime of the optimization step

UMASIS solver loop				
Name	runtime PC (s)	runtime GPU (s)	Opt. speedup	Tot. speedup
Calculate velocities	11.323	0.254	1,6	48
Calculate boundaries velocities	0.100	0.046	2,4	2.2
Insert source points velocities	0.054	0.035	2,6	1.5
Apply taper velocities	4.226	0.107	1,5	40
Calculate stresses	23.608	0.412	1,5	57
Calculate boundaries stresses	0.166	0.055	2,5	3
Insert source points stresses	0.057	0.037	2,2	1.5
Apply taper stresses	4.884	0.134	1,5	37
Mirror stresses	0.042	0.032	2,6	1.3
Calculate beam	43.639	0.129	1,4	338
Record data	0.960	0.058	1.2	17
Store data	1.032	1.078	1	0.9
Total	90.091	2.377	1.4	38

4.6 Analyses

To evaluate the results, a comparison is made between the results and the specifications of the NVIDIA GTX 285 (Table 5). The beam function is used for this comparison; it is the most significant step of the UMASIS solver. The GTX 285 can, according to its specifications, deliver up to 933 Gflops for single precision floating point calculations and deliver an estimated memory bandwidth of 158.9 GB/s. The reference PC (Table 3) can deliver in theory up to 6 Gflops, for a single core, and have a theoretical memory bandwidth of 16 GB/s. So, the theoretical speedup in performance for floating point calculations on the GPU versus the reference is at best 155 times, as its the memory bandwidth is 10 times wider.

By profiling the performance of the beam function is measured, this is done by calculating the total amount of floating point operations for the beam function divided by the runtime. The overall memory throughput is 130 GB/s where the estimated number of flops is 71.7 Gflops. The results of the measured memory bandwidth are in line with the theoretical specification (~82%). The performance expressed in flops is using only ~7% of the total theoretical specification. The reference PC delivers 0.21 Gflops (~3.5%) and a memory bandwidth of 0.38 GB/s (~3%).

The GPU and the reference PC are performing both far below the theoretical maximum amount of flops. This does, however, not imply that both implementations are underperforming for this specific case. The problem with expressing performance in flops is that a number of parameters are not taken in account. The amount of memory reads/writes and branching by conditional statements influences the operation performance. These parameters are not included in the measurements but can have a great influence on the performance.

The memory bandwidth throughput of the GPU implementation is close to the maximum performance which shows that the memory is more efficiently used by the GPU. It can be expected that new hardware with a higher memory throughput and lower memory latencies will very likely increase the execution speed significantly. The speedup between the GPU and the reference PC is 338 times and this is substantially more than the maximum theoretical speedup. It seems that the GPU is executing the algorithm more efficiently than the reference PC.
The combination of parallelized processing and a more efficient memory throughput delivers the result of this speedup.

4.6.1 Limiting factors

One of the limiting factors can be the precision of single floating point calculations. In Figure 52 the beam results of the solver versions are shown. Left shows the result of the standalone PC version written in C and at the right the GPU version. The specifications of the PC are shown in table 3.

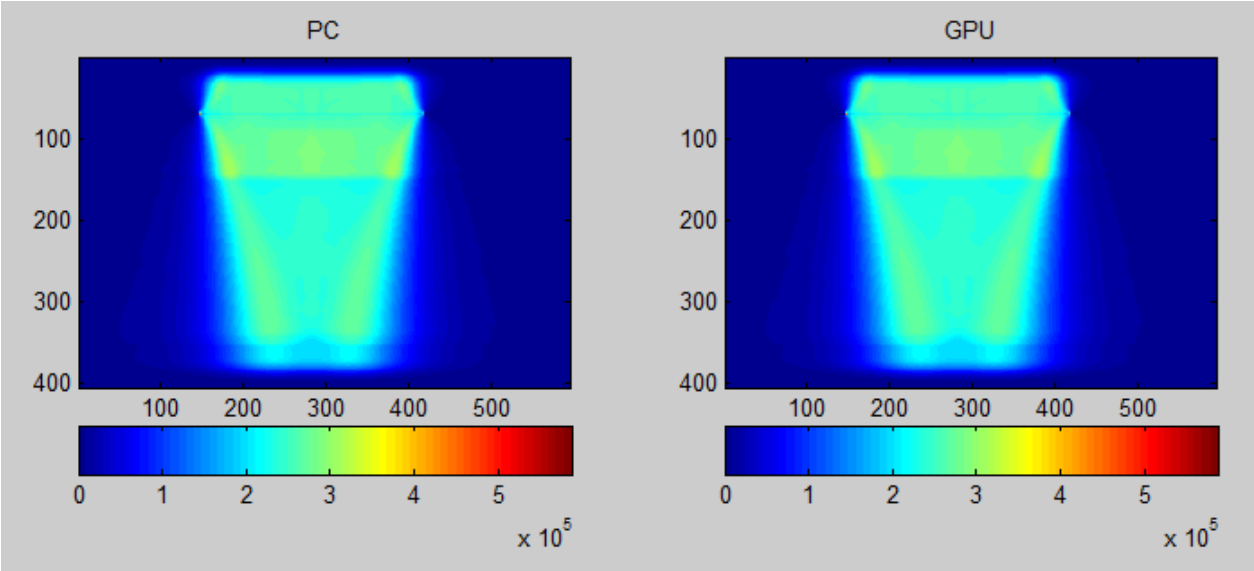

Figure 52. Beam results from PC (left) and GPU (right)

At first sight there are no noticeable differences between the results. In Figure 53 the differences between the beam results of the PC and the GPU are shown. Here it can be determined that there are some differences in output. The differences for the CUDA implementation are within the precision tolerance limits for this case. However for future research it is interesting to investigate the differences in precision for cases with a greater amount of time steps.

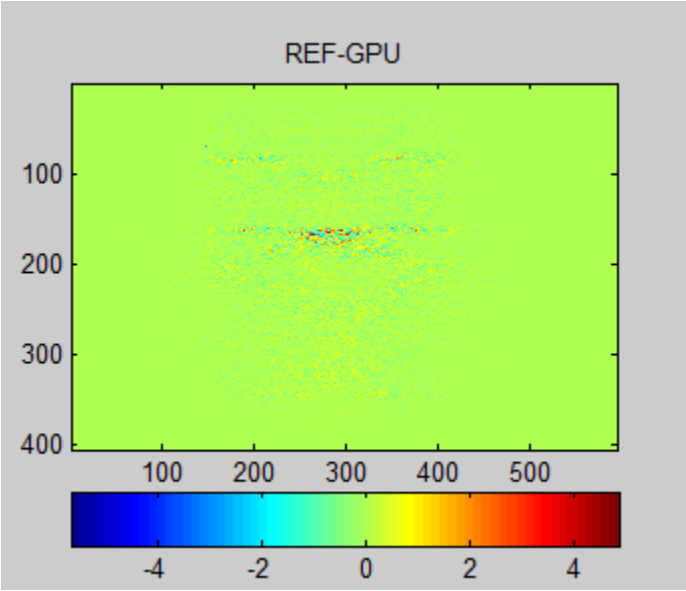

Figure 53. Differences in precision between PC and GPU

Another limiting factor is the onboard memory of the GPU. If the size of the simulation is too big to fit in to the global memory, the simulation has to be divided over multiple GPU blocks. This will generate extra communication overhead and the implementation will be more complex. Proper propagation of velocities and stresses at the edges of the blocks needs to be communicated between the blocks.

4.6.2 Future optimization

It is not expected that much higher speedup will be derived by tweaking the code any further. It is also expected that the time consumed by further optimization in relation to the speedup will not pay off effectively.
Faster runtimes may be achieved easier by using faster hardware or by using multiple GPUs. Multiple GPUs can be mapped as grid where each GPU in grid takes a part/block of the simulation. This approach could be beneficial, especially for large simulations where the amount of memory on the GPU is limiting the simulation performance. If the simulation data fits on the global memory of one GPU it is better to keep the data locality as high as possible and avoid using multiple GPUs.

4.6.3 Development process

In Figure 54 the development of the UMASIS implementation on the GPU is shown.
The first phase (I), where the code was analyzed and converted to provide the right data for the GPU, is taking ~16% (19 h.) of the total development time. This phase is in comparison to the other phases the smallest phase. The length of this phase highly depends on the size and complexity of the code. The code size was moderate (1000+ lines of code) and the complexity in terms of parallelization was not very high, due to the amount of independency between iterations in the double for-loops. In the second phase (II), a great speedup was derived from the reference version. Almost 75% of the total gain was achieved in this phase. The reason for this can be found in the parallelizability of the algorithm, especially for the calculation of the Beam, Tapering, Stresses and Velocities, consisting mainly in point and convolution calculations. This phase took ~41% (49 h.) of the total implementation time. In the final phase (III), the implementation was further analyzed and functions optimized. This resulted in a speedup of almost 25%, but the total gain as result of optimization was less successful than in the previous phase, while the total development time was almost equal ~42% (50h.). The reason for the this was, that optimization had less impact because of the easiness of parallelization of certain parts (Beam etc.). Therefore the implementation of some functions were almost optimally during the first implementation, leaving less room for further optimization in the next phase. Also some optimization reduced communication overhead but complicated at the same time the code, resulting in a modest speedup. The total development time of the CUDA-enabled UMASIS implementation took 118 hours and gained a speedup of 40 times in relation to the reference PC. This is much lower than the theoretical expected speedup (155x), but this is difficult to achieve over the whole application. The beam function exceeded the theoretical maximum almost two times (338x), showing that a great speedup can be achieved by applying CUDA on certain steps of an application.

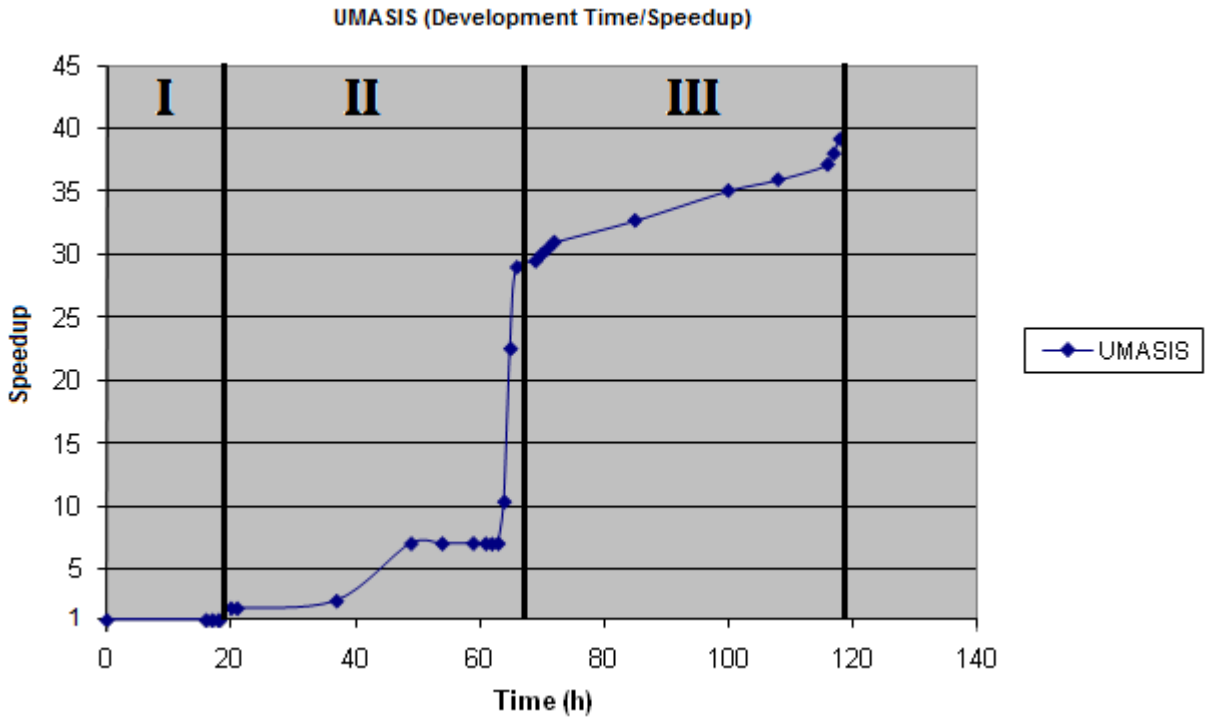

Figure 54. Graph divided in development phases

4.7 Summary

In this chapter the implementation of UMASIS on the GPU was discussed. Results show an overall speedup of almost 40 times (Figure 54). The total development process took up to 120 hours divided into three phases. Each phase was analyzed to determine the steps for the next stage and to evaluate the entire process. Two limiting factors were analyzed: The calculation precision and the GPUs global memory size. A possible solution was mentioned to gain an extra speed up by applying multiple GPUs. Finally the development process of the UMASIS case was discussed.

Chapter 5: Discussion

In this chapter both case studies are being discussed. The contents are as following.
In Chapter 5.1 a comparison is made between the development processes of both cases.
In Chapter 5.2 an overview of CUDA-enabled GPUs are given and compared utilizing the UMASIS implementation. Parallelization results from OpenMP are compared with that of CUDA in Chapter 5.3. In Chapter 5.4 the maturity of CUDA is analyzed. In Chapter 5.5 the overall analysis of the whole process is discussed and conclusions are presented based on the results of both use cases.

5.1 Comparison between the two cases

Comparing the two cases shown in Figure 55, it can be determined that there are great differences in development time and performance gain. The development time of UMASIS is almost 3 times smaller than that of the object detection, while the performance gain is a factor 10 times larger.

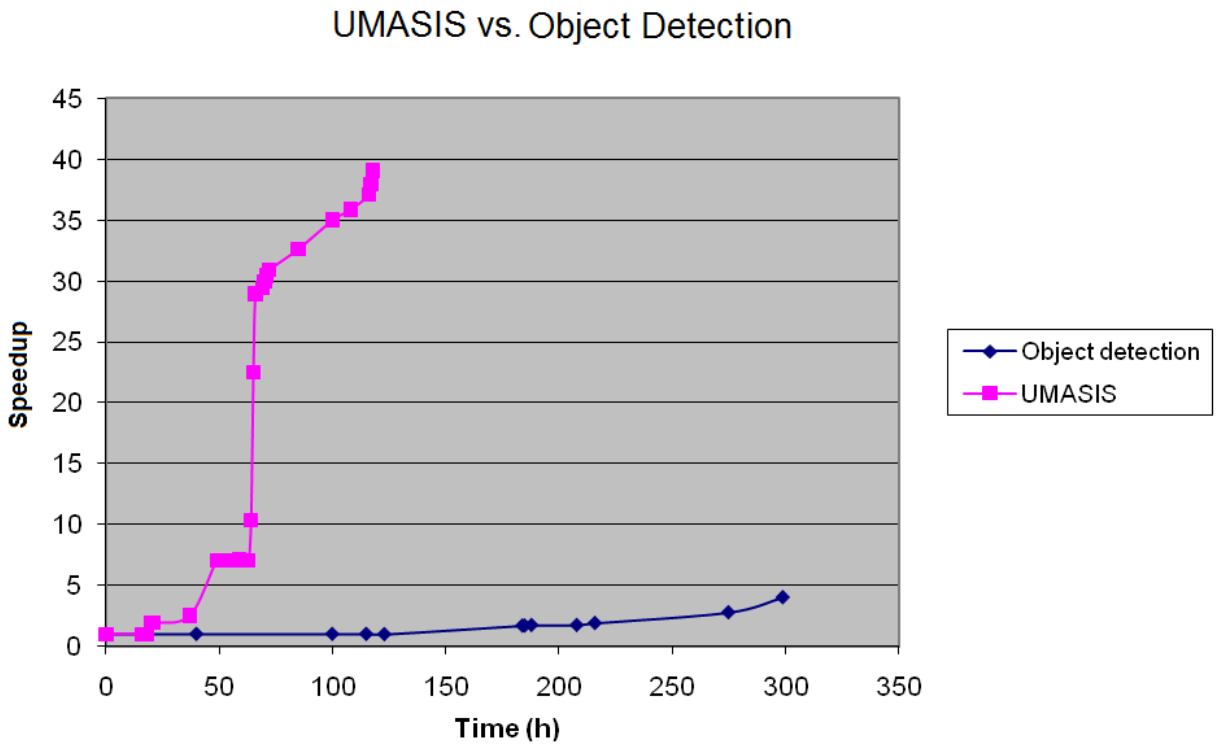

Figure 55. Speedup vs. development time of object detection and UMASIS

The reason that UMASIS is more capable to be implemented for CUDA is due to a series of factors, influencing the successfulness of an implementation greatly.

- The first, most important, factor is the requirement for high independency between input and output within loops. The grade of independency determines the parallelizability of the algorithm. In general most matrix operations are very appropriate for parallelization on the GPU, UMASIS executes convolutions and matrix point calculations.
- Another important factor is the degree in which the code is able to decompose in fine grained parts. This way the kernel of each part can be kept as small as possible, reducing the total number of registers, required to execute the kernel.
- Locality of data is also very important. Keep the data on the GPU as close as possible to the processing elements, make use of the global memory, shared memory and registers. Avoid swapping redundant data between the host and the GPU, this way the amount of communication overhead can be reduced greatly.
- Avoid the use of loops within the kernel functions, the cores are not optimized for branching in comparison to general purpose processor cores. It will be difficult to fill up the full warp size resulting in less optimal core occupation.
- Use the memory bus to its full extends, by aligning read/write data. This will increase the memory throughput and avoids redundant memory latencies.

Most of the mentioned factors are not applicable to the full extend on the object detect case. This is the reason why the results are far from optimal. The algorithm has to be seriously revised, to gain and meet the factors. For future CUDA developments it will be wise to estimate beforehand the successfulness of a CUDA implementation with the mentioned factors.

5.2 Performance comparison between various CUDA-enabled GPUs

There is a broad range of CUDA-enabled GPUs, which vary in price and specifications.
The performance and capabilities of a selection of CUDA-enabled GPUs are analyzed.
The selection criterion has been set to the price range of the GPU. Three kinds of GPU classes are defined, being low-end, mid-range and high-end. The low-end GPU is the class of GPUs with a market value less than €100. The mid-range is defined between €100 and €200 and the high-end is defined from €200 and above.

5.2.1 Specification

For the performance comparison the UMASIS case is utilized. The three GPUs specified in Table 13 and a reference computer specified in Table 11 is used for the comparison. In the reference computer a single core of the processor is used.

Table 11. Comparison between GPUs

Class	High-end	Mid-range	Low-end
Card type	GTX285	8800GTX	9600GT
Architecture	GT200	G80	G94
Hardware version	1.3	1.0	1.1
I/O Interface	PCI-E 2.0 x16	PCI-E 2.0 x16	PCI-E 2.0 x16
Number of cores	240	128	64
Core clock	648 MHz	575 MHz	600
Shader clock	1476 MHz	1350 MHz	1500
Memory clock	1242 MHz	900 MHz	800 Mhz
Memory bandwith	158.9 GB/s	86.4 GB/s	57.6 GB/s
Memory size	1024 MB	768 MB	512 MB
Memory interface	512 bit	384 bit	256 bit
Single precision	933 GFLOPS	518 GFLOPS	312 GFLOPS
Double precision	78 GFLOPS	not supported	not supported
Consumption (TDP)	~183 Watt	~177 Watt	~59 Watt
Price (Euro) (4-3-2010)	250	150	80

5.2.2 Comparison

The results in Figure 56 and Table 12 show all CUDA-enabled GPUs used for this comparison perform better than the reference computer. The GPUs are not performing as expected, conform their specifications in their class. There is no linear relation between the number of cores and the resulting performance. The mid-end delivers 3 times the performance of the low-end where the high-end performs almost 11 times better. This is not only because the memory clock, bandwidth and interface are higher for each class but because there are also architectural differences between the CUDA-enabled GPUs [24]. Therefore it is important to evaluate the GPUs architecture, for example the high-end architecture is optimized for CUDA resulting in a higher performance.

A new important parameter is energy consumption. Comparing the energy consumption between various classes, it can be derived that the consumption of the mid-range and high-end class is almost equal. The low-range class consumes 3 times less energy compared to the other two classes. This class however performs almost 11 times less compared to the high-end class. This makes the high-end GPUs the most energy efficient.

Currently only the high-end class is able to calculate double precision floating point calculations. In the future it is expected that all the other classes will be able to perform these calculations.

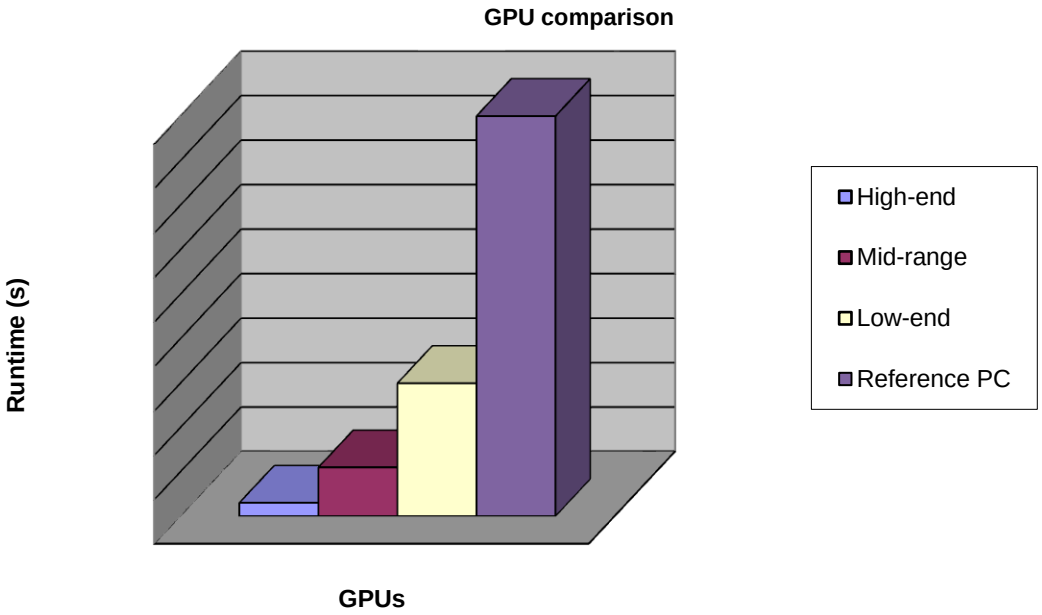

Figure 56. Comparison between GPUs and reference PC, lower is better

Table 12. Comparison between GPUs and reference PC

UMASIS Solver		
Name	Runtime (s)	Speedup from reference PC
High-end	2.4	38
Mid-range	10	9
Low-end	30	3
Reference PC	90	1

5.3 OpenMP vs. CUDA

Current generations of general purpose processors consist of multiple processor cores. OpenMP can be used to utilize all the cores. Both use cases are extended with OpenMP functionality.
In the next paragraphs the performance gained from the application of OpenMP is discussed. The comparison is made between a single, a quad core processor and the CUDA implementation. The implementation time required for utilizing OpenMP is compared to CUDA. The specifications of the reference PC are shown in Table 13 and that of the GPU, used for the CUDA implementation, in Table 5.

Table 13. Specification of the reference PC used for profiling

Processor	Intel Q9550
CPU Speed	2.83 GHz
Number of Cores	4
L2 Cache Size	12 MB
Memory Size	4GB
Bus Speed	4x333MHz
Operating System	Microsoft Windows 7 32bit
Compiler	Microsoft Visual Studio 2008

5.3.1 Object detection

The most significant part of the total runtime of the object detection application is the feature extraction function, which is called within the object detect step, see Figure 19.
The for-loop around the feature extraction function can be parallelized by adding the following syntax before the for-loop:

```
#pragma omp parallel for num_threads(max_threads) schedule(dynamic)
```

Hereby the for-loop is divided in multiple threads, where each thread is automatically assigned to the available processor cores. All threads have to finish their detection before the resulting detection is composed. This is why all the detection results of each thread has to be temporarily stored, done by creating a dynamic array for each thread, where the position and scale of a detection is stored. After the detection the results of all threads are merged to gain the resulting detection. This OpenMP implementation was realized within 4 hours.

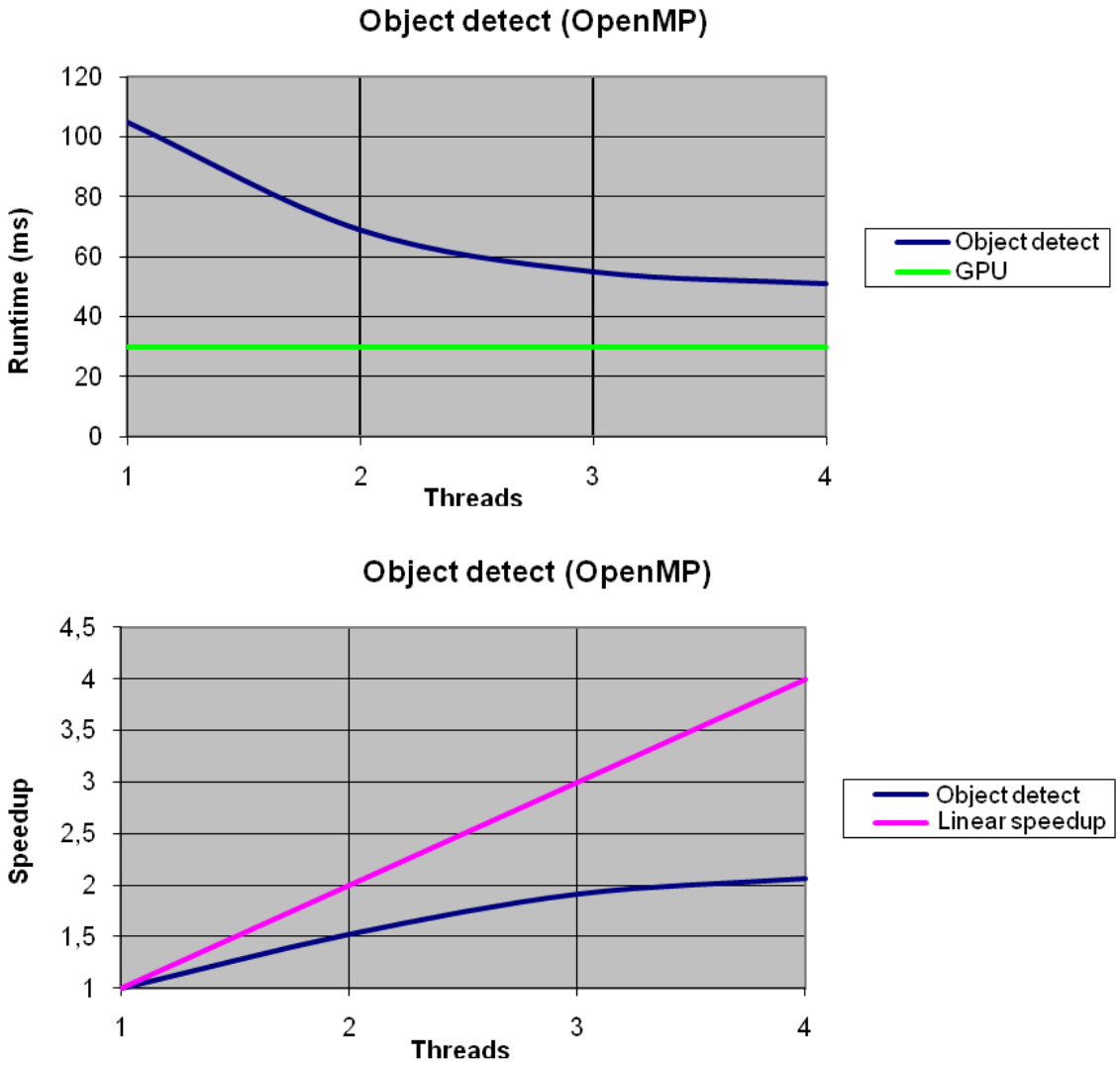

Figure 57. Comparison OpenMP vs. GPU for object detection

In Figure 57 the speedup and runtime of the OpenMP implementation is shown. The results show the speedup and runtime up to 4 threads for the object detection application. Also the linear speedup and the runtime of the GPU are plotted. The speedup is calculated by taking the runtime of the single-thread object detection, divide by the runtime of the multi-thread execution. From the figures it can be determined that the speedup does not follow the ideal speedup, which is the linear speedup. With two threads enabled a speedup of 1.5 is achieved instead of the expected 2 times. Utilizing more than three threads did not have any significant effect on the performance.

5.3.2 UMASIS

The most significant parts, i.e. calculate beam, velocities, stresses and tapering are parallelized by adding the following syntax before the outer for-loop:

```
#pragma omp parallel for shared(e.g. A, B) private(x,y) schedule(dynamic)
```

This way the double for-loop is divided in multiple threads. The threads are automatically scheduled in a dynamic order to the available processor cores. An unique x and y value are set to each thread by applying private(x,y). Within shared(…) variables are defined that can safely be shared among other threads. This OpenMP implementation was realized within 1 hour.

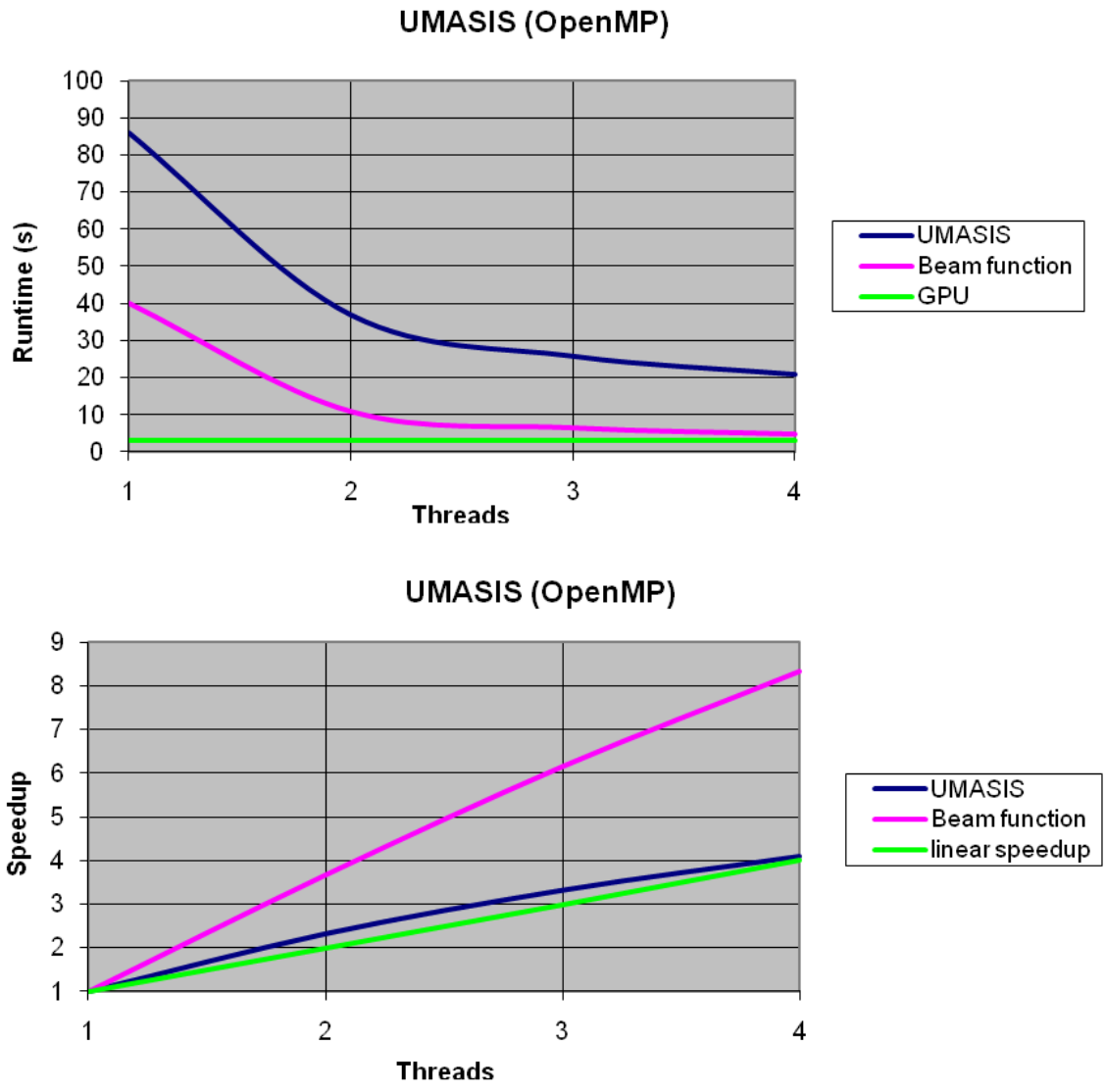

Figure 58. Comparison OpenMP vs. GPU for UMASIS

In Figure 58 the speedup and runtime of the OpenMP implementation are shown.
The results show the speedup and runtime up to 4 threads for the UMASIS and beam function implementation. Also the linear speedup and the runtime of the GPU are plotted. The beam function is the most significant step in terms of runtime within the UMASIS application; this is why this function is presented. The speedup is calculated by taking the runtime of the single-thread UMASIS implementation, divided by the runtime of the multi-thread UMASIS OpenMP implementation. From the figures, it can be determined that the UMASIS application is almost equal to the linear speedup, which is the ideal speedup, having N numbers of processor elements resulting in a speedup of N. The speedup for 2 and 3 threads is slightly larger than respectively 2 and 3 times. This phenomenon is called super linear speedup [40]. Also the beam function is showing a speedup far greater than the linear speedup. An explanation of this phenomenon can be found by the extra caches each core accumulates to the total cache size. Therefore less memory i/o to slower memory spaces is required, resulting in a speedup greater than linear speedup. This ended up advantageous for this specific application, but is not automatically the case for other applications. This case is not an unique case for FTDT as can be read in a published paper on IEEE [37], where a super linear speedup is achieved with the parallelization of FTDT by making use of MPI.

5.3.3 Conclusion

General purpose processers are evolving to many-core processors in the near future.
Therefore, parallelism is important to make full profit of these new architectures.
By applying OpenMP, this can be realized in a fast and convenient way. The speedup versus development time ratio is far better than the CUDA implementation. OpenMP was implemented in UMASIS and object detection in respectively one en four hours. The speedups gained were respectively 4 and 2 times on a quad-core processor. This is in comparison with the speedup of 40 and 4 times on the GPU much more advantageous, considering that the development time of both cases was over hundreds hours. The only disadvantage is that multi-core processors are still having a limited amount of processing cores in comparison of GPUs; therefore it is still not possible to achieve the amount of parallelism that can be performed on a GPU. A correlation between the speedup of the GPU and OpenMP can be found for both cases. The speedup of the object detect application is for the GPU and OpenMP less advantageous than for the UMASIS application. This is why it could be interesting to perform first an OpenMP implementation to determine whether it is profitable to perform a CUDA implementation, since the OpenMP implementation requires much less time.

5.4 Maturity of CUDA

One of the most important questions is the maturity of the CUDA platform. Is it profitable to invest in further development of CUDA-enabled applications? Is there enough ground and assurance that current CUDA applications will be future proof? How has CUDA developed thus far and can predictions for the future be made? All these questions are related to the question of the CUDA maturity.

5.4.1 Performance

Both use-cases, object detection and UMASIS, are utilizing CUDA version 2.3 and are executed on CUDA hardware version 1.3. To determine whether a gain in performance is realized between the successive versions, the first stable version 1.0 of CUDA is compared to the most recent version 2.3. For this comparison the UMASIS solver is used to compare the performance differences between the two releases. The solver is compiled with the provided tools from version 1.0 of CUDA. The binaries are executed and the runtime of the solver is measured. The results are being compared with the results as described in Chapter 3.4, which is based on version 2.3. It appeared that the differences between the runtimes of both versions can be considered to be negligible. So, it can be concluded that the GPU hardware has much more impact on the performance than the version of CUDA, see Chapter 5.2.

5.4.2 Feature developments

All CUDA versions are evaluated from the first -stabile- version to the most recent one. Throughout the versions new features are being introduced. Some of the new features are GPU architectural dependent. The GPU independent features are for example improved debugging and evaluation of system functioning. The hardware dependent features can only be used from a certain architectural type. By making use of new hardware dependent features, the CUDA application is not 100% backwards compatible anymore. These new architectural dependent features can be enabled by setting a compiler flag. An example of an architectural dependent feature is ability to perform double precision calculations.

5.4.3 Future developments

NVIDIA has announced a new architecture called the Fermi architecture. The Fermi architecture introduces new features which are not available on older CUDA-enabled GPUs and is compatible with older CUDA-enabled applications. However, to make full benefit of the new architecture the original CUDA code probably has to be reviewed.
An example of a new feature is the move from Single Instruction Multiple Thread (SIMT) to Multiple Instruction Multiple Thread (MIMT) architecture. This implies that multiple kernels are able to be run concurrently on the GPU, enabling for example the use of recursive functions. Also the performance of double precision calculations has been improved with a factor of 8. Another feature is the introduction of two cache levels. The global memory on the GPU can be greater than 4 GB and is error corrected.
These new features enable certain applications, which were in the past, less capable for execution on the GPU. The applications of the object detection and UMASIS can be improved greatly by using these new features. The extra caches, double point calculation improvement and MIMT can boost the performance greatly for the object detection application. The object detection uses double point calculations which are slower on older CUDA-enabled GPUs. The possibility of running multiple concurrent kernels (MIMT) can be usable in the object detection case. The caches reduce the latency involved with each memory fetch, where the MIMT architecture improves the core occupancy.

5.4.4 Conclusion

The performance of CUDA-enabled applications is not highly dependent on the version of the CUDA compiler. The GPU hardware has much more impact on the performance, see Chapter 5.2. If it comes to features, CUDA can be considered as still being in a development phase, considering the future perspectives and developments.
The compatibility is assured due to the fact that new features have to be enabled by setting extra compiler flag options. This disables compatibility for older non-feature compatible GPUs. However older CUDA-enabled applications are still compatible with newer generations. This is an important feature that saves a lot of time and energy, thereby reducing maintenance costs.
OpenMP can become an opponent to CUDA in the near future as general purpose processors evolve from multi-core processors to many-core. The development time required for a CUDA implementation is the greatest weakness for CUDA; it is however possible that in the future the development time will be reduced by introducing new hardware features (e.g. branch prediction, prefetching) or by introducing a better/smarter compiler . It is expected that CUDA will, performance-wise, be more efficient than OpenCL, however the risk of vender lock-in is much lower with OpenCL and OpenMP compared to CUDA.
Results show for the UMASIS implementation, that CUDA can already be used to get a significant speedup (40x). This depends heavily on the nature of the algorithm used, as seen in the object detection case (4x). New hardware features introduced for CUDA may result in greater speedups for less CUDA-appropriate applications in the near future.

5.5 Summary

In this chapter the development process of both use-cases were discussed. The development results turned out to be significantly different for the object detection and UMASIS case. This is mainly due to differences in the very nature of the algorithms. The UMASIS algorithm is able to achieve a high grade of independency between code, resulting in a fine grained decomposition keeping the data communication between host and GPU as low as possible. These factors are important for a successful implementation on the GPU. So, it is important to investigate these factors in advance to estimate the grade an algorithm will run on a CUDA-enabled GPU. Also the development time of CUDA-enabled applications can be better predicted.
OpenMP is applied to parallelize the use-cases for general purpose multi-cored processors. This application resulted in a speedup of 2x and 4x for, respectively, the object detection and UMASIS case on a quad core processor. OpenMP can be used as an indicator to predict the parallelizability of an algorithm. The main advantage of OpenMP is the ease of implementation into an algorithm. Finally the maturity of CUDA is evaluated, important for the decision on whether it is wise to apply CUDA accelerating applications. Due to the backwards compatibility of current and future CUDA-enabled GPUs, a CUDA accelerated application will be able to run on all the current and next CUDA-enabled GPUs. Therefore it is not required to invest extra time to use CUDA accelerated applications on future devices as performance may increase each generation. The main reason, making CUDA to be an interesting platform, is the great performance you can achieve (UMASIS).

6 Conclusions

A new trend in computing is the use of multi-core processors and Graphics Processing Units (GPUs) for general purpose high speed parallel computing. Therefore, TNO wants to respond to the current trend of parallel computing and investigate on which architectures would be the best to apply and parallelize their applications on. This thesis discusses the implementation of two applications used by TNO onto a CUDA-enabled GPU and a multi-core processor.
The applications chosen for the implementation are object detection and ultrasound simulation (UMASIS). The development time and performance gain were measured during the implementation of both cases. With these measuring results an estimation of future projects can be made.

6.1 Summary

In Chapter 2 the background is given for this thesis. The current developments in processing and future developments were discussed. The two main architectures, multi-core and NVIDIA CUDA-enabled GPU, were shortly introduced. Types of parallelism were explained and samples of parallelization of code were given. Finally CUDA, OpenMP and OpenCL were discussed and some code examples shown.

In Chapter 3 the implementation of object detection on the GPU was discussed. Results show an overall speedup of 4 times (Figure 28). The total development process took 300 hours, divided in three phases. Each phase was analyzed to determine the steps for the next stage and to evaluate the whole process. The algorithm was analyzed to find the limiting factors. Three reasons were found: In the use of registers, decision tree and scattered reads. For this problem a solution is given to avoid the limitations and achieve a higher speedup. This by changing the way the algorithm is parallelized. The main drawback of this solution is the time required to apply the changes. Also other optimizations were discussed, estimated to require less development time. Finally the development process of the object detection case was discussed.

In Chapter 4 the implementation of UMASIS on the GPU was discussed. Results show an overall speedup of almost 40 times (Figure 52). The total development process took up to 120 hours divided into three phases. Each phase was analyzed to determine the steps for the next stage and to evaluate the entire process. Two limiting factors were analyzed: The calculation precision and the GPUs global memory size. A possible solution was given to gain an extra speed up by applying multiple GPUs. Finally the development process of the UMASIS case was discussed.

In Chapter 5 the development process of both use-cases were discussed. The development results turned out to be significantly different for the object detection and UMASIS case. This is mainly due to differences in the very nature of the algorithms. The UMASIS algorithm is able to achieve a high grade of independency between code, resulting in a fine grained decomposition keeping the data communication between host and GPU as low as possible. These factors are important for a successful implementation on the GPU. So, it is important to investigate these factors in advance to estimate the grade an algorithm will run on a CUDA-enabled GPU. Also the development time of CUDA-enabled applications can be better predicted.
OpenMP is applied to parallelize the use-cases for general purpose multi-cored processors.

This application resulted in a speedup of 2x and 4x for, respectively, the object detection and UMASIS case on a quad core processor. OpenMP can be used as an indicator to predict the parallelizability of an algorithm. The main advantage of OpenMP is the ease of implementation into an algorithm. Finally the maturity of CUDA is evaluated, important for the decision on whether it is wise to apply CUDA accelerating applications. Due to the backwards compatibility of current and future CUDA-enabled GPUs, a CUDA accelerated application will be able to run on all the current and next CUDA-enabled GPUs. Therefore it is not required to invest extra time to use CUDA accelerated applications on future devices as performance may increase each generation. The main reason, making CUDA to be an interesting platform, is the great performance you can achieve.

From the results of this thesis the following conclusions can be made:

- **Great differences in results between CUDA implementations, in terms of speedup and development time.**
- **High speedups can be achieved by making use of CUDA, but the results are highly dependent on the algorithm.**
- **It is strongly advised to evaluate the algorithm before a CUDA implementation is performed.**
  Some indicators for CUDA successfulness are:
  - high degree of independency between operations
  - ability to make a fine grained decomposition
  - locality of data
- **It is difficult to exploit the full theoretical capabilities of CUDA-enabled GPUs.**
  Latency and memory bandwidth are one of the limiting factors.
- **CUDA is still in a development phase, but is mature enough to be used for implementations.**
  This due to the fact high speedups can be achieved, the way new features are implemented and compatibility is assured on newer architectures.
- **OpenMP is a fast and convenient way to parallelize applications for general purpose multi-core processors**.
- **The results from an OpenMP implementation can be used as indicator for the successfulness of a CUDA implementation.**
  This in terms of expected performance gain, not development time.
- **OpenCL is an interesting future development, in terms of compatibility between various architectures.**
  But code still needs to be adapted to make full use of a specific architectures.

6.2 Contributions

The main issue addressed by this thesis was the analysis of the performance gain and development time on a CUDA-enabled GPU and multi-core processor.

The contributions of this thesis are:

- **Implementation of object detection on a CUDA-enabled GPU:** The object detection application was successfully implemented on a CUDA-enabled GPU. The performance gained from this implementation was 4 times compared to the reference computer.

- **Implementation of UMASIS on a CUDA-enabled GPU:** The UMASIS application was successfully implemented on a CUDA-enabled GPU. The performance gained from this implementation was almost 40 times compared to the reference computer.
- **Analysis of the development process for CUDA based on two implementations:** The development time and performance gain were recorded during the implementation of both use cases. With this, data performance and time to develop estimations, can be done for future projects. This by comparing both cases with the to be implemented case. To ease this process, some important factors, related to the success rate of the implementation, were identified.

6.3 Future Suggestions

This section concludes this thesis by suggesting some ideas for future works and improvements on the use of CUDA-enabled GPUs.

Some of these suggestions for future works are the following:

- **Comparison between OpenCL and CUDA:** A new introduced API for many-cored processors is OpenCL (Chapter 2.5.3). The question is, how does this new API perform in comparison to CUDA, in means of implementation, performance and compatibility. This new API is also interesting as it is not only compatible with NVIDIA branded GPUs but also compatible with other GPUs and other multi-cored architectures. How is the portability and what are the performance differences between multiple architectures?
- **Investigate the precision of CUDA-enabled GPUs:** In the UMASIS case differences were found between calculation results of a CUDA-enabled GPU and the reference computer. The differences were within acceptable limits for UMASIS case. However, in some cases it may result into faulty results. This is why it is advisable to analyze the precision over multiple cases. Are there ways to increase or decrease precision and to what costs?
- **Investigate the use of multiple CUDA-enabled GPUs:** In this research only one CUDA-enabled GPU was used. There are ways to make the current applications faster using multiple GPUs. Interesting questions are: how is the scalability and what is the development effort?

Appendix A: NVIDIA GPU Architecture (CUDA)

CUDA (Compute Unified Device Architecture) is a parallel computing architecture developed by NVIDIA [24]. CUDA enables programmers to use GPUs for non-graphics calculations. GPUs with their highly parallel architecture are capable of processing over more than a thousand threads at a very high speed. This is why, propelled by recent development on CUDA and GPU computing, an increase of interest from the industry and academics has been created. In Table 11 specifications of some CUDA-enabled GPUs are shown.

Kernel, grid, threads and Blocks

In general portions of code, an application can be rewritten to functions able to be executed in parallel. This type of function is called a kernel and can be executed, one at a time, on the GPU. (see Figure 59 for a schematic view). The kernel consists of a grid with a programmer defined amount of blocks. Each grid can have up to 3 dimensions and 65,536 blocks. Inside the blocks there is a programmed amount of threads. Blocks can have up to 3 dimensions and contain up to 512 threads. Threads can communicate inside blocks, but not between blocks.

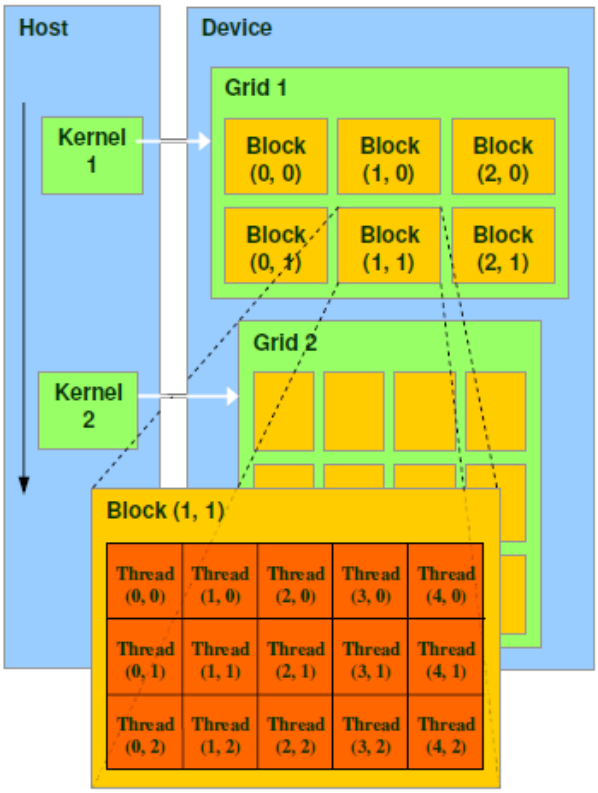

Figure 59. CUDA organization of kernels, grids, blocks and threads

Streaming Multiprocessors

The Tesla architecture (Figure 60) is built around a scalable array of multithreaded Streaming Multiprocessors (SMs), which consists of:

- Eight Scalar Processor (SP) cores
- Two Special Function Units (SFU)
- Multithreaded Instruction Unit (MT issue)
- On-chip shared memory

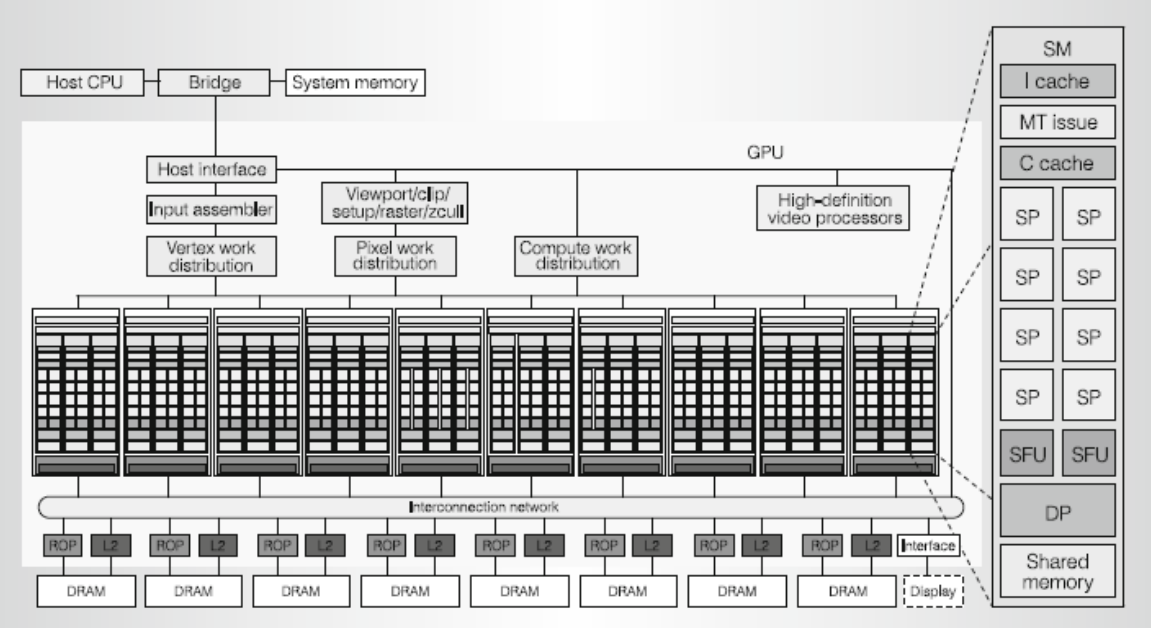

Figure 60. NVIDIA CUDA architecture

When a CUDA program on the host CPU invokes a kernel grid, the blocks of the grid are enumerated and distributed to multiprocessors with available execution capacity.
The threads of a thread block execute concurrently on one multiprocessor. As thread blocks terminate, new blocks are launched on the vacant multiprocessors.

The multiprocessor creates, manages, and executes concurrent threads in hardware with zero scheduling overhead. A barrier synchronization can be initialized in a single instruction by calling __syncthreads(). Therefore it is possible to have threads with a small problem size.

SIMT

The multiprocessor uses a new architecture SIMT (single-instruction, multiple-thread), to manage hundreds of threads running several different programs. This SIMT architecture is similar to SIMD (Single Instruction, Multiple Data). SIMT instructions specify the execution and branching behavior of a single thread. Unlike SIMD, SIMT enables programmers to write thread-level parallel code for independent scalar threads and data-parallel code for coordinated threads. You do not have to make use of the SIMT behavior; however a serious speedup can be achieved by taking SIMT behavior in account.

Warp

The multiprocessor SIMT unit creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps. A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path. If threads of a warp come to a data dependent conditional branch, then the warp serially executes each branch path taken¸ till the paths are completed. The threads go back to the same execution path when all the paths are completed. This only occurs within a warp, other warps execute independently even when they are executing common or disjointed code paths.

Individual threads, part of a SIMT warp, start together at the same program address but are free to branch and execute independently. When a multiprocessor is given one or more thread blocks to execute, it splits them into warps getting scheduled by the SIMT unit. The way a block is split into warps is always the sam. Each warp contains consecutive threads. The SIMT unit selects a warp ready to execute and issues the next instruction to the active threads of the warp.

Memory

Each multiprocessor has on-chip memory of the four following types (Figure 61):

- One set of local 32-bit registers per processor.
- Parallel data cache or shared memory, shared by all scalar processor cores.
- A read-only constant cache, shared by all scalar processor cores speeding up reads from the constant memory, which is a read-only region of device memory.
- A read-only texture cache, shared by all scalar processor cores, speeding up reads from the texture memory space, which is a read-only region of device memory. Each multiprocessor accesses the texture cache via a texture unit that implements the various addressing modes and data filtering.

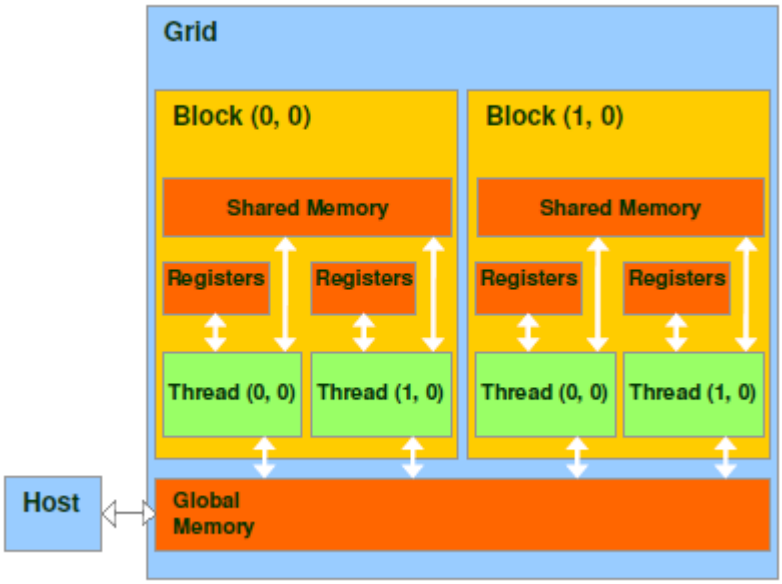

Figure 61. CUDA architecture

The global, constant, and texture memory are optimized for different memory usages. Texture memory also offers different addressing modes, as well as data filtering, for some specific data formats. The global, constant, and texture memory spaces are persistent across kernel launches by the same application.

The multiprocessor registers and shared memory are split among the threads of the blocks.
If there are not enough registers or shared memory available per multiprocessor to process at least one block, the kernel will fail to launch.

CUDA threads may access data from multiple memory spaces during execution.
Each thread has private local memory and access to the global memory. Also two additional read-only memories are accessible by all threads, namely the constant and texture memory. Each block has a shared memory, visible to all threads of the block and with the same lifetime as the block.

Bibliography

1. John Owens, "GPU architecture overview", ACM SIGGRAPH 2007 courses, August 05-09, 2007, San Diego, California

2. M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Yao Zhang, V. Volkov, "Parallel Computing Experiences with CUDA," Micro, IEEE In Micro, IEEE, Vol. 28, No. 4. 2008, pp. 13-27.

3. D. C. Pham, T. Aipperspach, D. Boerstler, M. Bolliger, R. Chaudhry, D. Cox, P. Harvey, P. M. Harvey, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Pham, J. Pille, S. Posluszny, M. Riley, D. L. Stasiak, M. Suzuoki, O. Takahashi, J. Warnock, S. Weitzel, D. Wendel, K. Yazawa, "Overview of the architecture, circuit design, and physical implementation of a first-generation cell processor," Solid-State Circuits, IEEE Journal of, Vol. 41, No. 1. 2006, pp. 179-196.

4. Cray Inc., "Cray XD1 Specifications", http://www.hpc.unm.edu/~tlthomas/buildout/Cray_XD1_Datasheet.pdf, 2004

5. Viola and Jones, "Rapid object detection using boosted cascade of simple features," Computer Vision and Pattern Recognition, 2001, pp. 511-518

6. B. Catanzaro, N. Sundaram, and K. Keutzer, ‘‘Fast Support Vector Machine Training and Classification on Graphics Processors,’’ Proc. 25th Ann. Int’l Conf. Machine Learning, Omnipress, 2008, pp. 104-111.

7. B. He et al., ‘‘Relational Joins on Graphics Processors,’’ Proc. ACM SIGMOD 2008, ACM Press, 2008; www.cse.ust.hk/catalac/papers/gpujoin_sigmod08.pdf.

8. M. Schatz et al., ‘‘High-Throughput Sequence Alignment Using Graphics Processing Units,’’ BMC Bioinformatics, vol. 8, no. 1, 2007, p. 474; http://dx.doi.org/10.1186/1471-2105-8-474.

9. S. Manavski and G. Valle, ‘‘CUDA Compatible GPU Cards as Efficient Hardware Accelerators for Smith-Waterman Sequence Alignment,’’ BMC Bioinformatics, vol. 9, suppl. 2, 2008, p. S10; http://dx.doi.org/10.1186/1471-2105-9-S2-S10.

10. V. Volkov and J.W. Demmel, ‘‘LU, QR and Cholesky Factorizations Using Vector Capabilities
of GPUs,’’ tech. report UCB/ EECS-2008-49, EECS Dept., Univ. of Calif., Berkeley, 2008.

11. R.H. Ni, ‘‘A Multiple Grid Scheme for Solving the Euler Equations,’’ Proc. AIAA 5th Computational Fluid Dynamics Conf., AIAA Press, 1981, pp. 257-264.

12. S.S. Stone et al., ‘‘How GPUs Can Improve the Quality of Magnetic Resonance Imaging,’’ Proc. 1st Workshop General Purpose Processing on Graphics Processing Units, 2007.

13. D. Frenkel and B. Smit, Understanding Molecular Simulations, Academic Press, 2002.

14. J.A. Anderson, C.D. Lorenz, and A. Travesset, ‘‘Micellar Crystals in Solution from Molecular Dynamics Simulations,’’ J. Chemical Physics, vol. 128, 2008, pp. 184906-184916.

15. John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips., "GPU Computing," Proceedings of the IEEE, 96(5):879–899, May 2008.

16. J.P. Farrugia, P. Horain, E. Guehenneux, Y. Allusse, "GPUCV: A framework for image processing acceleration with graphics processors", CDROM proc.of the IEEE International Conference on Multimedia & Expo (ICME 2006), July 9-12, 2006, Toronto, Ontario, Canada.

17. J. Fung and S. Mann. "Using graphics devices in reverse: GPU-based image processing and computer vision," In IEEE Int'l Conf. on Multimedia & Expo, 2008, pp. 9--12

18. Yuancheng Luo, Ramani Duraiswami, "Canny edge detection on NVIDIA CUDA," cvprw, 2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, 2008, pp.1-8

19. OpenCV, “OpenCV 1.1 C++ Reference”, OpenCV, 2009, http://opencv.willowgarage.com/documentation/cpp/index.html

20. Simbiosys, Inc.“The fast and the furious: compare Cell/B.E., GPU and FPGA”, Simbiosys Blog, 2008,http://www.simbiosys.ca/blog/2008/05/03/the-fast-and-the-furious-compare-cellbe-gpu-and-fpga/

21. Moore, G.E, “Progress in digital integrated electronics,” Electron Devices Meeting, 1975 International, Volume: 21, 1975, pp. 11- 13

22. Barillet R et al, "Limitation of the Clock Frequency Stability by the Interrogation Frequency Noise," Experimental Results, IEEE Transactions HYPERLINK \l ""on Instrumentation and Measurement, IEEE Inc. New York, US, vol. 42, No. 2, Apr. 1, 1993, pp. 276-280

23. D. Anderson, B. C. Dewitt, R. L. Eisele, M. G. Gallup, Y. W. Ho, S. C. Mcmahan, D. Marquette, B. Martin, K. C. Scheuer, L. C. Sood, T. S. Spohrer, "The 68040 32-b monolithic processor," Solid-State Circuits, IEEE Journal of In Solid-State Circuits, IEEE Journal of, Vol. 25, No. 5. 1990, pp. 1178-1189.

24. NVIDIA Corporation, "NVIDIA CUDA Programming Guide," whitepaper, http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf, ver. 2.3.1, 2009

25. Amdahl, G. "The validity of the single processor approach to achieving large-scale computing capabilities". Proceedings of AFIPS Spring Joint
Computer Conference, Atlantic City, N.J., AFIPS Press, 1967, pp. 483–85.

26. John L. Gustafson, "Reevaluating Amdahl's Law", Communications of the ACM 31(5), 1988, pp. 532–33

27. Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. "Cilk: An Efficient Multithreaded Runtime System," Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 1995 pp. 207–216.

28. Leonardo Dagum, Ramesh Menon, "OpenMP: An Industry-Standard API for Shared-Memory Programming," Computing in Science and Engineering, vol. 5, no. 1, January-March, 1998. pp. 46-55

29. Snir, Marc; Otto, Steve; Huss-Lederman, Steven; Walker, David; Dongarra, Jack, "MPI: The Complete Reference," MIT Press Cambridge, MA, USA. 1995, ISBN 0-262-69215-5

30. Intel, "Intel(R) Threading Building Blocks", Reference Manual, ver. 1.16, http://www.threadingbuildingblocks.org/uploads/81/91/Latest%20Open%20Source%20Documentation/Reference.pdf

31. Alastair Donaldson, Anton Lokhmotov, Colin Riley, Andrew Cook. "Auto-parallelisation of Sieve C++ Programs," In Proceedings of the Euro-Par Workshop Highly Parallel Processing on a Chip (HPPC'07), August 2007. Lecture Notes in Computer Science 4854, 2007.

32. Cray Inc., "Cray's Chapel," http://chapel.cray.com/spec-0.785.pdf, whitepaper

33. Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., Sam Tobin-Hochstadt, " The Fortress Language Specification," Sun Microsystems inc., ver. 1.0, 2008, http://projectfortress.sun.com/Projects/Community/browser/trunk/Specification/fortress.1.0.pdf

34. Vijay Saraswat, "Report on the Programming Language X10," IBM Research, Ver. 2.0, 2009 http://dist.codehaus.org/x10/documentation/languagespec/x10-200.pdf

35. Aaftab Munshi, "The OpenCL Specification version 1.0," Khronos OpenCL Working, ver. 1.0, rev. 48 Group, 2009, http://www.khronos.org/registry/cl/specs/opencl-1.0.48.pdf

36. Kane Yee, "Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media,". Antennas and Propagation, IEEE Transactions on 14, 1966, pp. 302–307.

37. Ciamulski, T.; Sypniewski, M., "Linear and superlinear speedup in parallel FDTD processing, " Antennas and Propagation Society International Symposium, 2007 IEEE, Volume , Issue , 9-15 June 2007, pp. 4897 - 4900

38. Bob Warfield, "Multicore Crisis", 2007, http://smoothspan.wordpress.com/2007/09/06/a-picture-of-the-multicore-crisis/

39. NVIDIA Corporation, “CUDA Occupancy Calculator”, 2008,
http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

40. Helmbold, D. P. and McDowell, C. E., "Modeling Speedup(n) greater than n,"
International Conference on Parallel Processing Proceedings, 1989, Volume III, pp. 219-225

41. Alan R. Levander, “Fourth-order finite-difference P-W seismograms,”
Geophysics, November 1988, vol. 53. no. 11, pp. 1425.-1436

Curriculum Vitae

J.A. Huisman was born on 28 September 1982 in Sliedrecht, the Netherlands. He completed his secondary education at the C.S. Oude Hoven in Hardinxveld-Giessendam in 1999. From 1999 to 2003 he attended the intermediate vocational education in Rotterdam. In 2003 he started at the Rotterdam University of Applied Sciences where he achieved his Bachelor of Information and Communication degree in 2006. He continued his studies in 2006 at the TU Delft computer engineering department where he is currently working on to receive his Master of Science degree. Besides his education and between the degrees he worked in software and hardware design for different companies. His main interests include embedded systems and computer architecture.