An OpenCL-based Solution for Portable Bodyscan SAR Processing on Multicore Platforms

Miloš Rašković
An OpenCL-based Solution
for Portable Bodyscan SAR Processing
on Multicore Platforms

Master’s Thesis in Embedded Systems

Parallel and Distributed Systems Group
Faculty of Electrical Engineering,
Mathematics, and Computer Science
Delft University of Technology

Distributed Sensor Systems Department
and Radar Department
TNO Defence, Security and Safety
The Hague

Supervisors

Henk Sips
Ana Lucia Varbanescu

Maarten Ditzel
Wouter Vlothuizen

Miloš Rašković

14th June 2010
Summary

Multicore systems have become an indispensable part of our everyday life. They represent a viable alternative for increasing processor performance without hitting the memory and power walls. However, the shift from traditional programming to multicore programming has a critical influence in three dimensions: the applications, the software tools, and the hardware platforms.

In this thesis, we focus on the application dimension, and we investigate the performance potential of a 3D bodyscan SAR processing algorithm running on several multi-core processors. To allow for the processor variety, the solutions are based on OpenCL, a language that enables inter-platform portability for a large set of multicores. To allow for performance, we choose to mainly evaluate and optimize for NVIDIA GPUs, but also for Intel GPPs and ATI GPU.

Our solutions design and implementation follow a step-by-step strategy. First, we analyse the application to determine its functionality, data characteristics, and performance requirements. Next, we design and implement a sequential reference solution, and we evaluate its performance. We use this particular solution to design a set of possible parallel solutions, which are also evaluated in terms of performance and platform utilization, using a generic prototyping platform. Further, we select the most promising parallel solution for a first portable OpenCL implementation. All platform-agnostic basic optimizations, are applied now, towards a new application, with increased performance and no decrease in portability. After these basic optimizations, we show how several platform-specific optimizations can be applied. Basically, in this step, we trade portability for performance. For this work, we specifically target NVIDIA GPUs, and we show the potential performance impact of data and memory-related optimizations on the case-study application.

Finally, we gather all these steps and findings into a generic, empirical strategy that enables programmers to reason about OpenCL applications in a systematic manner, letting them decide the level of the trade-off between the application portability and performance.

Our main conclusions are twofold. First, we conclude that OpenCL is a promising standard (and language) for enforcing the implementation of portable multicore applications. Second, we conclude that the bodyscan application is a good fit for running on multicore platforms, and we recommend the GPUs as the target platform for such an application.

For future work, we propose to advance in two directions. First, as generic research, we propose to focus on the strategy validation and refining, as well as on a more abstract way to derive the parallel OpenCL versions. Second, on the application/technical side, we plan to focus on finding and implementing hardware-dependent optimizations of the OpenCL solution on various hardware platforms.
Contents

1 Introduction 1
  1.1 Background and Related Work ........................................... 2
  1.2 Research questions ......................................................... 3
  1.3 Contributions ................................................................. 4
    1.3.1 Research Contributions .............................................. 4
    1.3.2 Technical Contributions ........................................... 5
  1.4 Thesis Organization .......................................................... 5

2 Multicore Systems 7
  2.1 Parallelism ................................................................. 8
  2.2 Hardware Platforms ......................................................... 10
    2.2.1 General-purpose Processors ....................................... 10
    2.2.2 High-performance Processors ..................................... 10
    2.2.3 Digital Signal Processors ......................................... 11
  2.3 Software Tools .............................................................. 11
    2.3.1 Compiler-based Approach .......................................... 11
    2.3.2 Language-based Approach ......................................... 11
    2.3.3 Model-based Approach ............................................. 12
    2.3.4 Software Development Kits ....................................... 12
    2.3.5 Comparison ............................................................ 12
  2.4 Applications ................................................................. 13

3 State of the Art 15
  3.1 Hardware Platforms ......................................................... 15
    3.1.1 Intel® Core™ i7 Processor Extreme Edition ..................... 15
    3.1.2 NVIDIA® Tesla™ 20-series ........................................ 16
    3.1.3 Texas Instruments® OMAP™ 4 ....................................... 18
    3.1.4 Cell Broadband Engine Architecture ............................ 19
    3.1.5 Summary ............................................................... 21
  3.2 Software Tools .............................................................. 21
    3.2.1 OpenCL ................................................................. 22
    3.2.2 Compute Unified Device Architecture (CUDA) .................. 22
    3.2.3 Ct ................................................................. 24
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>A Strategy for OpenCL Application Design and Implementation</td>
<td>97</td>
</tr>
<tr>
<td>8.1</td>
<td>Strategy</td>
<td>97</td>
</tr>
<tr>
<td>8.2</td>
<td>Summary</td>
<td>102</td>
</tr>
<tr>
<td>8.3</td>
<td>Bodyscan Case-study</td>
<td>103</td>
</tr>
<tr>
<td>9</td>
<td>Conclusions and Future Work</td>
<td>107</td>
</tr>
<tr>
<td>9.1</td>
<td>Research Questions Revisited</td>
<td>109</td>
</tr>
<tr>
<td>9.2</td>
<td>Future Work</td>
<td>110</td>
</tr>
<tr>
<td>A</td>
<td>Application Computational Intensity</td>
<td>117</td>
</tr>
<tr>
<td>B</td>
<td>NVIDIA OpenCL Bugs</td>
<td>119</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Multicore systems have emerged as an alternative solution to increasing performance. First, transistor size is reaching its theoretically smallest size. Second, even with the current transistor sizes, the power consumption has reached alarming values. Multicore systems have arise to address these problems. They utilize the chip in a different way. Instead of having single large, and complex cores, multicores use several (multi) smaller and simpler cores. In this way, performance is gained with parallel applications. Furthermore, power consumption is more controllable and an energy management system can use only the needed number of cores, putting other cores in stand-by mode.

One of the key problems generated by multicore hardware platforms is the drastic shift in programming concepts. In order to utilize multiple cores efficiently, applications need to run tasks in parallel. This ultimately requires a paradigm shift from traditional, sequential programming, to the more cumbersome parallel programming. While parallelism is by no means new, it is still lagging behind in terms of tools and user-support for extracting application parallelism. Furthermore, parallelism used to be a luxury of eccentric scientists. Now, with the super-fast increase of multicore platforms in both numbers and performance, it has to become mainstream.

Another problem that comes with the advances in multicore technology is the increased variety of platforms. Applications need special parallelisations, optimizations, and tuning strategies for each platform. Therefore, the guaranteed portability of the sequential algorithms is also lost.

In this thesis we focus on investigating the performance of a case-study application when implemented using OpenCL, a framework that targets portability over multiple hardware platforms. An OpenCL application can, in principle, run on hardware platforms consisting of general-purpose processors, graphics-purpose processors and similar processors, without code modifications. In general, any processor that provides support for the OpenCL application programming interface can be used. We envision that the OpenCL framework will be one tool able to respond to the portability challenges of modern multicore hardware in the near
The motivation for investigating multiple multicore platforms is a bodyscan application, whose performance needs improvement from the previous parallel solution. The bodyscan uses small radars and a three-dimensional synthetic aperture radar (SAR) principle to generate the image of a body. Usual real-life application of this technique is in the detection of possibly hidden weapons. Briefly, bodyscan scans the human body and collects millions of samples, called observations. The output is formed by coherently adding these observations for all the output positions of the interest, which is in our case the human body.

At the first glance, the bodyscan application seems inherently parallel, since all the output samples can be processed in parallel. However, the very large data sets and irregular accesses pose obstacles to generic optimizations. Despite these difficulties, we were able to design and implement a generic OpenCL solution for the bodyscan application, which we present extensively in this thesis. We show how the application is portable and performs much better than hand-tuned code on GPPs (using OpenMP) or on the Cell/BE (using the software development kit). Further, we show how, with an assumed drop in portability, we can increase performance using data and hardware-dependent optimizations. Finally, we develop a strategy for systematic development of the OpenCL solutions as a set of step-by-step guidelines.

1.1 Background and Related Work

Our work is a spin-off from the previous work on the 3D bodyscan SAR project, where the bodyscan algorithm proposed in [75] has been implemented on the IBM Cell processor.

In general, any work on 3D SAR processing is on-going research and is mostly related to the airborne SAR. Moreover, the 2D SAR processing has been well researched [18]. Although 3D SAR processing is much more demanding in terms of the measurements compared to the 2D SAR processing, the same SAR processing algorithms can be used for both 2D and 3D SAR processing.

Traditionally, SAR algorithms are based on convolutions in the frequency domain [15, 17, 66]. However, we base our work on the approach proposed in [75], where time-domain is used. Time-domain algorithms require much more computing power, which was hardly accessible before the new era of multicore systems. Different approaches have been used to tackle the problem (for example, see [21]). This thesis proposes several parallel implementations of the algorithm presented in [75].

In conclusion, although 2D SAR processing has been well researched and there is on-going research on 3D airborne (long-range) SAR processing, the author is not aware of any similar on-going or published work on 3D bodyscan (short-range) SAR processing and the type of algorithms used in this thesis. The in depth analysis of the design alternatives, the implementation (with various optimizations), the
performance analysis, together with the use of OpenCL, make our approach unique to this date.

1.2 Research questions

This section lists and briefly explains the important research questions that we answered in this thesis. They are:

1. What are the current multicore systems trends, in both hardware and software?
   We show the current multicore system trends in three dimensions: hardware, software, and applications. We make an overview of the current state of the art developments and compare them to the past trends. This leads us to envision the multicore systems of the near future.

2. What is the structure of the application? How does the main algorithm look like? What is its sequential (baseline) performance?
   We analyse the application in order to understand its structure and requirements. We devise a sequential algorithm and we develop a sequential reference solution that serves as a base for further development. We measure performance, determine bottlenecks, and set our performance goals.

3. How can the algorithm be parallelized on a homogeneous, shared-memory multicore? What is the most suitable parallel algorithm for this problem, in terms of performance?
   We aim to develop a parallel solution that could be run on homogeneous, shared-memory multicores (general-purpose processors). We determine the efficient solutions and analyse the new application performance and bottlenecks.

4. How can the parallel algorithm be transformed to suit graphics-processing units (GPUs)? Can the solution be portable, both in terms of design and implementation?
   When the implementation for a general-purpose processor is settled, we look at the options to develop other solutions that can be run on different type of multicore systems, namely GPUs. We select OpenCL as a potential suitable software framework, as it addresses both portability and performance concerns.

5. Is OpenCL truly portable, both in terms of code and performance?
   Once our OpenCL solution is in place, we investigate its portability. OpenCL implementations are claimed to be portable over different types of multicores. Therefore, we explore the performance of the same application when running on different platforms.
6. Are there any generic optimizations? What is the impact of platform-specific optimizations on the application performance when running on a particular family of processors? What is the impact of data-dependent optimizations on the performance of the OpenCL solution? After exploring portability, we target different kinds of optimizations in order to evaluate how they influence application performance. We are interested in both general optimizations that preserve the portability and in hardware-specific optimizations.

7. Can we derive a strategy for developing OpenCL solutions? How shall we approach portability and performance in OpenCL?
After developing several solutions and performing different kinds of analysis, we summarize our experiences in a set of guidelines. The guidelines attempt to give a practical set of steps that should be followed in order to efficiently map an application to the OpenCL. Furthermore, they address both portability and performance issues.

1.3 Contributions

In this section we list our major research and technical contributions.

1.3.1 Research Contributions
The major research contributions of this thesis are the following:

- We survey current trends of multicore systems in terms of hardware and software, predicting possible trends in the near future.

- We analyse a state of the art radar application, namely 3D bodyscan SAR processing. We propose several parallel solutions and we select the most promising ones, in terms of portability and performance. Further, we implement and benchmark them, proving that the application has high potential for parallel execution, thus being very suitable for multicore processors.

- We analyse OpenCL, a novel software framework that focuses on inter-platform portability. We evaluate its usability, portability, performance, and its available features, as well as its missing ones for an application of the size and complexity of SAR processing.

- We develop a strategy, as a practical set of step-by-step guidelines, that describes an efficient design and implementation of OpenCL applications. We show how should the application be analysed, how should the efficient parallelisations be explored and finally, how are the portability and performance addressed in OpenCL.
1.3.2 Technical Contributions

To tackle both the portability and the performance issues of the bodyscan SAR application, we have developed the Bodyscan SAR Processing Software package, which consists of three parts: the framework, the solution collection, and the environment. We argue that this package in itself is a major technical contribution, due to its flexibility, its completeness, and its user-friendly interface.

- The framework alleviates to programmer to focus on the key algorithm used for the 3D SAR processing, by providing a set of consistent data elements and methods that can convert different types of measurements to the unique data set. It is simple, flexible, portable, and fault-proof.

- The solution collection present implementations of the 3D bodyscan algorithm. Almost all the solutions discussed and analysed in the thesis are ported to the framework and included in this collection. Furthermore, the collection is easily extendible with additional solutions (guidelines provided).

- The bodyscan environment is developed to unite the framework and the solution collection, and provides the user with a set of parameters for direct tweaking of the solutions.

Overall, we believe that the framework presents a valuable tool for fast and flexible development of SAR processing algorithms. Furthermore, on longer term, the framework might be extended to support other (similar) applications for multicore processors, as it provides a flexible environment for prototyping OpenCL solutions for multicore applications.

1.4 Thesis Organization

The thesis is organized as follows. Chapters 2 and 3 present multicore systems: the basics, available hardware platforms and software tools, and applications. Chapter 2 gives a general overview on multicore systems. It presents the base concepts of multicores by discussing different forms of parallelism available both at the hardware and at the software level. Then, it gives a classification of the multicore hardware platforms and software tools. The chapter is concluded with a short list of applications that are naturally suitable for multicores. Chapter 3 continues the previous chapter with an overview of the state of the art developments in the field of multicore systems. We survey several state of the art hardware platforms, software tools, and applications. The chapter is summarized with a “look in the future”, where based on the history of multicore developments and current trends, we try to predict how will the multicore systems look like in the future.

In Chapters 4 and 5 we introduce the bodyscan application. In Chapter 4 we look at the radar principles and the details specific to the Synthetic Aperture Radar
Next, we present and analyse the SAR processing algorithm, the central algorithm in the thesis. While Chapter 4 looks at the SAR from the radar perspective, Chapter 5 looks at the SAR processing algorithm from the computing perspective. In particular, in Chapter 5 we present specifications of the SAR bodyscan application. We introduce the important bodyscan parameters from the aspect of processing, and we look at the application memory requirements. Moreover, we reveal the details of the test cases used.

Chapters 6 and 7 describe all the implemented solutions. First, we discuss the experimental setup, the platforms, and the used metrics. Then, we present the reference solution. After evaluating the reference solution, we turn to parallel solutions. We develop OpenMP solutions for general-purpose processors and portable OpenCL solutions. The OpenCL solutions are the major part of the thesis. After implementing the reference OpenCL solution and evaluating its portability we do a series of optimization and analysis with the goal to increase the application performance on the GPUs. We do general optimizations (that do not influence the portability), data-dependent optimizations, and hardware-specific optimizations. Finally, we briefly summarize our findings on performance vs. portability and productivity for different solutions. Chapter 6 describes the general implementations issues, presents the reference solutions and the OpenMP solutions, while Chapter 7 focuses on OpenCL implementations.

Chapter 8 attempts to collect our experiences into a strategy that describes how to systematically develop OpenCL solutions. The strategy is developed as a set of step-by-step guidelines. We start from the top-level application where we analyse work, data, and performance requirements. By careful analysis, with the reference sequential solution and the generic parallel solutions, we choose the right parallelisation and we aim at developing OpenCL solutions. Our first focus is on portability, after which we focus on performance and optimizations at various levels.

Finally, Chapter 9 concludes the thesis. It summarizes our work and findings, and proposes directions for future work.
Chapter 2

Multicore Systems

After several decades of pursuing higher performance with the increase of the clock frequency, current technology is reaching its limits [11, 13]. On the one hand, the transistor size is approaching the smallest size, theoretically possible. On the other hand, even with the current transistor sizes, power usage has become an important issue. Engineers have been eager in following the Moore’s law [10] according to which the number of transistors on a chip is doubled every 18 months (see Figure 2.1). However, power does not scale down linearly with transistor size. If we consider that silicon scaling improves transistor density by 50% per generation, the power is reduced only by 20% per generation [23].

Figure 2.1: Moore’s law with respect to the Intel processors [76].
Aside from the technological problems just described, a broad range of applications do not need high performance in terms of clock frequency, but measure their performance in other metrics. For instance, high-throughput performance is becoming a dominant requirement by many important applications, such as image and video processing, radar processing, weather modelling, etc. Multicore systems come as an alternative solution addressing the key problems mentioned previously. Multicore systems are parallel machines that have several (multi, many) independent processing units (cores) that are able to run programs in parallel. Each core can either run a completely independent program or can run a part of a larger program, executing one thread of the program execution. The idea behind the multicore systems is to utilize the product between the number of transistors and cores in a different way compared to the traditional processor architecture. A multicore processor has many simpler cores (with less transistors), while a traditional processor has one large core (with more transistors), thus meaning that the product between the number of cores and transistors is effectively the same.

On the one hand, the main disadvantage of multicores compared to traditional processors is that they have lower programming flexibility. On the other hand, there are numerous advantages, many of them answering modern pressing issues in processor architectures. For instance, power consumption in multicore systems is more controllable. Smart power management systems can be implemented that can put some of the cores to stand-by when the workload is low. Overall, having multicore systems gives new wings to the field of the high-performance computing.

The fact that multicore systems have become a mainstream tool of all processor vendors to increase performance has put a burden on programmers to develop applications for such parallel machines. The programming model of multicore systems conceptually differs from the sequential programming model that is deeply settled in the industry. Not only efficient tools and software environments need to be developed, but also programmers need to learn how to think in parallel, often a non-intuitive and non-straightforward process.

In the following sections we will take a closer look of some important parameters and developments of the field of multicore computing. We will first explain many types of parallelism that are exploited nowadays both within hardware and software (Section 2.1). Afterwards, we will turn our focus on the main aspects of multicore systems, namely: hardware platforms (Section 2.2), software tools (Section 2.3) and applications (Section 2.4). After making a general overview in this chapter, the next chapter will detail about some of today’s state of the art developments.

2.1 Parallelism

We can broadly distinguish two major types of parallelism: hardware and software parallelism. Under hardware parallelism we consider the processor hardware capabilities that allow parallel execution of the program. From the coarser to the finer level of parallelism, we distinguish:
• Core-level parallelism - A processor has more than one core, where each core has its own instruction and data memory, meaning that each core is capable of running an independent program. This parallel model is often referred to as multiple-instruction multiple-data (MIMD) model.

• Data-level parallelism - A processor has one core that has only one instruction memory, but has several data memories, meaning that one program can run over different data sets. This parallel model is often referred to as single-instruction multiple-data (SIMD) model.

• Simultaneous multithreading (SMT) - SMT permits multiple independent threads of execution to better utilize the resources provided by modern processor architectures. Usually, in this way, by switching threads, a processor is able to hide memory latencies.

• Pipeline parallelism - The finest level of parallelism where program instructions are issued in a pipeline, thus better utilizing the available processor resources.

Usually, these hardware parallelisms capabilities are combined. For instance, often we have several cores (core-level parallelism) where each core has either one instruction memory and several data memories (data-level parallelism). In addition, almost all modern processor include some kind of pipeline parallelism.

Software parallelism [53] is defined as a technique utilized at the application programming level to exploit parallelism in a program. We distinguish the following levels of software parallelism:

• Task-level parallelism - Program execution is divided among several tasks and the communication between the tasks is established with special synchronization constructs.

• Loop-level parallelism - The execution of the loops is parallelised so that different parts of the loops can be executed by different threads of the execution. This is also a form of data level parallelism, since the loop iterates over the same instructions with different data, but we use the term loop-level parallelism to distinguish it from the hardware parallelisation named data-level parallelism.

The exploitation of software parallelism is tightly related to the underlying hardware platform. For instance, we cannot run several tasks at once if we have only one thread of execution available.

Ideally, the main task of the programmer is to specify the high-level parallelism available at the application level, while system software (compilers, drivers, operating systems, etc.) should be more involved in the finer levels of the parallelisation process. For example, task-level and loop-level parallelism can be extracted by the programmer in a great degree with a special programming language, while
mapping from the software parallelism to the hardware parallelism should be completely handled by the appropriate software tools (compilers, for instance).

2.2 Hardware Platforms

Processors (or more general, computers) have become a ubiquitous and indispensable part of our everyday life. Many types and families exist. Generally, based on their target application, we classify processors in three categories [13]:

- General-purpose processors (GPPs);
- High-performance processors (often referred to as high-performance computers - HPCs);
- Digital signal processors (DSPs).

In this sections we give an overview of general features of each class, while in Section 3.1 one particular representative of each class is described in more detail.

2.2.1 General-purpose Processors

GPPs are usually intended for the server and desktop market where power consumption is usually not a major concern. These systems are designed for the best average performance on most workloads. GPPs usually have few cores, high clock frequencies, low throughput, and high power consumption. They mostly exploit task-level parallelism by the means offered by the operating system and classical software tools can be used here. They usually provide a great level of flexibility. As an example of the GPP we selected the Intel® Core™ i7 Processor Extreme Edition, which is described in Subsection 3.1.1.

2.2.2 High-performance Processors

HPCs are more specialized architectures designed for high-throughput applications (which are the primary focus in this thesis). They either have a moderate number of high-level cores or a large number of simpler cores. Typical HPCs are characterised by moderate clock frequencies, high throughput, and high power consumption. While GPPs put an accent on the clock frequency (Hz), HPCs emphasize the throughput (FLOPS - floating point operations per second). Usually HPCs target task and loop parallelism at the application level and often some vendor-specific tools are required, beside classical tools that can be used. They have significant amounts of special-purpose hardware, customized for the special application needs. The most notable examples are found in graphics applications where many hardware components are specialized for image and video processing. As an example we selected the NVIDIA® Tesla™ 20-series graphics processing unit (GPU), detailed in Subsection 3.1.2.
2.2.3 Digital Signal Processors

DSPs present even more specialized hardware architectures. They have a limited number of cores (sometimes even just one), most of them being specially-dedicated functional units. The DSPs usually have low clock frequencies, low throughput and very low power consumption. They have the least flexible programming model and are specialized for particular application fields, having most of the features already implemented in hardware. Very specialized software tools are needed to program the processors. As an example we selected the Texas Instruments® OMAP™ 4, described in more details in Subsection 3.1.3.

2.3 Software Tools

The gap between various hardware architectures and many different applications needs to be bridged with efficient software tools [11]. We need to port, optimize and tune applications for different hardware architectures taking into concern the time-to-market requirements. Software tools aim to ease programming and to efficiently map the programs to the hardware architectures, thus enabling to programmers to focus on high-level aspects of the application without the need to know all the details of the underlying hardware architecture.

There are four fundamental approaches we can use for writing parallel programs [67]:

- Compiler-based approach;
- Language-based approach;
- Model-based approach;
- Software development kits (SDKs).

In the following subsections we give an overview of the listed approaches and briefly compare them.

2.3.1 Compiler-based Approach

A compiler-based approach puts the burden of parallelising an application on the compiler. A sequential program is given to a compiler that automatically parallelises it for the given hardware target platform. However, compilers can only extract limited parallelism from the application [67].

2.3.2 Language-based Approach

In a language-based approach, one can use an appropriate language to explicitly define parallelism in the code. On the one hand, we can use languages with special extensions that give indications to compilers how to exploit parallelism. On
the other hand, we can use special libraries, so-called application programming inter-
faces (APIs), that provide parallelisation, communication, and synchronization
functions. Most of the languages that support parallelism come with both base
language extensions (using some base sequential language like C) and APIs.

2.3.3 Model-based Approach

A model-based approach uses predefined programming models that abstract the
low-level details of implementations. For instance, two fundamental models are
finite-state machine (FSM) models, suited for control-oriented applications, and
actor-based models, designed for computation-oriented applications. In this ap-
proach, the application is described with some high-level language in terms of a
model. If the application suits the model well, then model-based approach offers
an efficient development path, because we can easily implement the application
with high-level constructs. These are then efficiently mapped to the hardware by
the supporting tools. However, if the application does not correspond directly to
the model, it might be very difficult to describe the application in terms of a model.
Finally, the model-based approach has not been widely accepted in practice, mainly
due to the fact that there are not so many applications that correspond very well to
particular models [25]. For more details about the model-based parallel program-
ing approach, we refer the reader to [25, 67].

2.3.4 Software Development Kits

Software development kits are special development environments that combine the
elements of the previous approaches and are usually provided by the hardware
vendors. This is especially true for the specialized hardware platforms like DSPs,
whereas vendors try to alleviate the use of the platforms with appropriate software
tools, usually in the form of an SDK. The loss in generality and portability when
developing with a vendor-specific SDK is compensated with the gain in perfor-
mance for the specific hardware platform.

2.3.5 Comparison

Since the compiler-based approach suffers from the limitations in extracting paral-
lelism, the model-based approach limits us to particular models, thus not providing
the required generality in developing multicore applications, and SDKs do not give
us enough generality, our focus is mainly on language-based approaches. How-
ever, this approach puts major responsibility on the programmers. In order to effi-
ciently develop multicore software, a programmer needs to carefully use language-
ensions and libraries.

Section 3.2 reveals some of the state of the art developments in the field of
language-based software tools.
2.4 Applications

We can freely say that nowadays the great majority of the processors are multicore processors\(^1\). Therefore, almost all of today’s applications will be executed on these machines, independent of whether the applications are parallel in nature or just simple sequential programs. However, our primary interest is in the high-performance computing applications, which naturally fit to multicore systems. Therefore, we limit our examples to such applications. Here we list some examples from different application fields, which naturally exhibit a lot of parallelism (mainly data-parallelism), while in the section 3.3 we describe in more detail some particular algorithms.

- Bio-informatics - For instance, DNA analysis includes millions of sequences that need to be analysed and compared in order to extract useful information. One of the most representative examples from this field is sequence alignment. This algorithm aims to find regions in one sequence that are similar or identical to the regions in another sequence (these regions can represent genes, for example). The sequences can be very long arrays of data which naturally lend themselves to data-parallelism.

- Finance - Risk calculation is just one inherently parallel example that requires simulations that use many parameters and large amounts of data (Monte Carlo simulation, for example).

- Image processing - Digital images are represented with pixels. Often, different algorithms are applied to an image to extract information or to improve quality. Furthermore, this processing can often be done in parallel on groups of pixels. One example of such an algorithm is detailed in section 3.3.

- Graphics processing - Although to some extent this has similarities with image processing, we can look at it as a special field. The gaming market is a representative example. To support CAD (computer-aided design) and high-resolution 3D games, powerful graphics processing is needed.

- Weather modelling - Weather prediction is based on a range of different parameters that lead to very data-intensive calculations. Tsunami modelling is one of the examples.

The list of application fields definitively does not end here, but it is apparent that multicore systems have found their use in many of today’s application areas.

\(^1\)We exclude embedded processors, which are usually very simple, application-specific processors.
Chapter 3

State of the Art

This chapter has a purpose to give an overview on the current multicore state of the art developments. Three key aspects are considered: hardware platforms, software tools and major application fields. Three hardware platforms, three software tools and several application fields are depicted as some of the key state of the art developments.

3.1 Hardware Platforms

Based on the classification given in the introduction (Section 2.2), for each class, one state of the art hardware platform is depicted. Three selected platforms are: Intel® Core™ I7 (general-purpose processor), NVIDIA® Tesla™ 20-series (high-performance processor), Texas Instruments® OMAP™ 4 (digital-signal processor). In particular, we take a look at the three important aspects: processor model (architecture and microarchitecture), memory model, and interconnect model. In addition, we take a closer look at Cell Broadband Engine processor, because we will compare our work to the work previously done, where Cell processor has been used as the implementation platform.

Although these hardware platforms are aimed for very different application fields and are therefore difficult to compare, Table 3.1 compares them against several important parameters.

3.1.1 Intel® Core™ i7 Processor Extreme Edition

Intel Core I7-960 [33, 40] is a 45nm technology general-purpose processor that does "everything" well. It targets high-end desktop, server, and workstation systems.

Concerning the computational power, it has 4 cores with four-way out-of-order execution model and can use up to two threads per core, thus enabling fairly good multitasking and multithreading. With a maximal frequency of 3.33GHz, it can deliver up to 55GFLOPS. The high price for large programming flexibility is paid
with power consumption of maximal 130W. The core architecture is standard Intel 64 architecture [37], while the microarchitecture is based on Intel Microarchitecture (Nehalem) [36]. The main architectural features (that Intel is very proud of) are the scalability (dynamical scalability and design-scalable microarchitecture) and energy/performance efficiency. Block diagram of the processor is presented in Figure 3.1.

Many levels of memory (memory hierarchy) are used to hide memory latencies. It has three levels of cache (L1-32KB of instruction and 32KB of data cache, L2-256KB for data and instructions, L3-8MB) and integrated memory controller that can support memory bandwidth of 25GB/s, which is also the maximal bandwidth supported by the interconnect. The amount of supported DRAM is outstanding 24GB [40].

The communication among the cores and between the cores and DRAM is done via Intel QPI point-to-point link capable of providing 25GB/s.

### 3.1.2 NVIDIA® Tesla™ 20-series

NVIDIA Tesla C2070 [59, 60] is particularly interesting hardware platform because it presents an attempt to unify both graphics and general-purpose processing [50], thus leading to the rise of the field of general-purpose processing on graphics processing unit (GPGPU). The main emphasis of the platform is on high-performance computing, in particular, high-throughput applications.

The "revolutionary" NVIDIA Fermi™ architecture [57, 69] presents an important step towards unified computing. The hierarchical platform organization is carefully customized to extract parallelism via multithreading and multitasking [50].

With 512 basic (CUDA¹) processors divided over 16 streaming multiprocessors

---

¹CUDA - "Compute Unified Device Architecture". Go to subsection 3.2.2 for more details about
(SMs) and throughput power of 630GFLOPS of double precision floating point numbers, it has remarkable throughput capabilities. Block diagram presenting the streaming processor is shown in Figure 3.2. Each streaming processor manages two groups of 16 CUDA processors, where each CUDA processor is fully pipelined integer arithmetic-logic unit (ALU) and floating point unit (FPU). An SM uses the so-called single-instruction multiple-thread (SIMT) architecture. In other words, threads are organized in groups of 32, called warps. Each SM manages 24 warps, while two warps can be active at the same time. More details on the programming model are given in the next section, where CUDA software platform is described.

A great step towards unified computing are the load/store units (LD/ST) that can access data in both cache and DRAM. Furthermore, the 40-bit unified address space with an extensive C++ support gives a very convenient programming model. As in the general-purpose processors, this amazing throughput power comes at the price of maximal $225W$. Finally, modest maximal core frequency is $1.4GHz$.

To handle large amounts of data there is a great memory demand which is solved by CUDA.
with two level caches (L2- 768KB) and outstanding 6GB of dedicated memory. Another convenience is that 64KB of memory per SM can be configured either as 16KB of L1 cache and 48KB of shared memory, or 48KB of L1 cache and 16KB of shared memory. An overview of Fermi architecture is shown in Figure 3.3.

![Figure 3.3: Top-level overview of Tesla architecture.](image)

### 3.1.3 Texas Instruments® OMAP™ 4

Texas Instruments OMAP 4 mobile applications platform [32] is developed for the latest Smartphones and Mobile Internet Devices (MIDs). It is a very heterogeneous architecture, with five processors and many more special-purpose peripherals that can support various range of advanced digital signal processing at very low power cost. Figure 3.4 gives an overview of the architecture. The low-power ARM® Cortex™-A9 MPCore™ processor [9] is the heart of the system. Other processors and peripherals are there to enhance image/video/audio processing.

ARM Cortex-A9 MPCore [8] is a RISC processor with the popular ARMv7 architecture. Block diagram of the architecture is shown in Figure 3.5. The processor is very configurable and can be adopted for a wide range of digital-signal processing applications, thus allowing developers to balance between peak-performance and power-consumption. It can have a variable number of cores, from 1 to 4, where OMAP 4 has a two-core processor. The maximal frequency is up to 1GHz with 2.50DMIPS/MHz\(^2\). The processor is out-of-order multi-issue superscalar with 8-stage pipeline.

\(^2\)We could not find the information about GFLOPS, probably because for this kind of processors,
The processor memory is consisted of two-level cache hierarchy (L2 can be up to 2MB). Interconnect is a bus (Advanced Bus Interface Unit) supporting industry-standard AMBA interface [7]. Finally, the power consumption is very low and it is even reported to be 1W [13] for some configurations.

### 3.1.4 Cell Broadband Engine Architecture

Cell Broadband Engine [29, 28, 19] is a supercomputer on a chip, jointly developed by IBM®, Sony®, and Toshiba®, with a purpose to give outstanding performance, particularly for game and multimedia applications.

---

that is an irrelevant metric.
Architecture of the Cell is a heterogeneous architecture consisted of nine cores. One core is a general-purpose processor designed with high clock frequency, with space and power efficiency in mind: Power Processor Element (PPE). Other eight cores are high-performance processors: Synergistic Processor Elements (SPE). Basically, PPE acts as a resource manager, while SPEs are there to do (intensive) data processing.

PPE is a 64-bit RISC Processor (PowerPC [27]) with 2-way hardware multi-threading, 2-level cache and 3.2GHz clock speed. At the heart of the SPE is Synergistic Processing Unit (SPU), a dual-issue, up to 8-way 32-bit SIMD, with its own local memory. Cell architecture and layout are presented in Figure 3.6. At the clock speed of 3.2GHz, each SPE can deliver up to 25.6GFLOPS, which leads to approximately 200GFLOPS in total (for all 8 SPEs). The Cell’s power consumption is 100W [13].

![Cell Broadband Engine Abstract Overview](image)

Figure 3.6: Basic block diagram (on the left) and layout (on the right) of the Cell.

Element Interconnect Bus (EIB) presents four 16B coherent data rings to which processors and memories are attached through special interface modules. Peak EIB Bandwidth is 96B/cycle, which results in more than 300GB/s and over 100 outstanding requests. Supported DRAM is XDR DRAM, which is connected via 25GB/s interface to the EIB.

PPE has a 2-level cache memory (L1- 32KB instruction memory and 32KB data memory; L2- 512KB) with a coherent load/store mechanism. Each SPE has its own local memory of 256KB (there is no cache) and its own DMA. By putting local memories and thus, by giving to programmer the control over the memory content, Cell designers wanted to allow predictable memory communication.

Since released, Cell processor has found its way in many applications, from gaming to high-performance computing. However, the most widely know Cell application is the Sony Playstation® 3 (PS3) [12]. Some example algorithms that are implemented with a great success on the Cell are given in [26, 22, 75, 17]. In particular, the reference implementation that is going to be used in this paper is
also implemented on the Cell and is given in [75].

### 3.1.5 Summary

We made an overview over four representative state of the art hardware platforms. Due to different application fields that these platforms target, they are built with different aims in mind. Nevertheless, to get a feeling of different application requirements and the latest state of the art hardware platforms, Table 3.1 summarizes the main features of the previously described hardware platforms.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Intel Core i7-975</th>
<th>NVIDIA Tesla C2050/2070</th>
<th>Texas Instruments OMAP4440</th>
<th>IBM Cell</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Application area</strong></td>
<td>high-end desktops, servers</td>
<td>high-performance computing</td>
<td>mobile application platforms</td>
<td>high-performance computing</td>
</tr>
<tr>
<td><strong>Technology</strong></td>
<td></td>
<td></td>
<td></td>
<td>45nm</td>
</tr>
<tr>
<td><strong>Number of cores</strong></td>
<td>4</td>
<td>32</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td><strong>Power (max)</strong></td>
<td>130W</td>
<td>190W</td>
<td>few watts</td>
<td>100W</td>
</tr>
<tr>
<td><strong>Frequency (max)</strong></td>
<td>3.33GHz</td>
<td>1.4GHz</td>
<td>1GHz</td>
<td>3.2GHz</td>
</tr>
<tr>
<td><strong>Throughput (max)</strong></td>
<td>55.36GFLOPS</td>
<td>630GFLOPS</td>
<td>N.A.</td>
<td>200GFLOPS</td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td>L3 (8MB)</td>
<td>L2 (768KB)</td>
<td>L2 (up to 2MB)</td>
<td>Subsection 3.1.4</td>
</tr>
<tr>
<td></td>
<td>max 24GB DRAM</td>
<td>6GB DRAM</td>
<td>N.A.</td>
<td></td>
</tr>
<tr>
<td><strong>Interconnect</strong></td>
<td>Intel QPI</td>
<td>N.A.</td>
<td>bus</td>
<td>ring</td>
</tr>
</tbody>
</table>

Table 3.1: State of the art hardware platforms.

### 3.2 Software Tools

From the variety of software tools available, three representative state of the art software tools are briefly explained: Open Computing Language (OpenCL), Compute Unified Device Architecture (CUDA) and Ct. Essentially, these tools present language extensions [67], providing a set of APIs and extensions to the C programming language for efficient representation of parallelism. However, while OpenCL targets heterogeneous platform model and CUDA targets NVIDIA GPUs, Ct does not make any assumptions about the underlying platform. We detail about these three software platforms in three aspects: platform model, programming model and memory model. Finally, Table 3.2 summarizes the key features of the software platforms.
3.2.1 OpenCL

The OpenCL [24], developed by the Khronos group, is a new open industry standard for programming a heterogeneous collection of CPUs, GPUs, and other discrete computing devices organized into a single platform [49]. The first stable release was published in December 2008, while vendors released the OpenCL implementation for their hardware few months later. For instance, NVIDIA released the first OpenCL implementation in June 2009.

The platform model assumes a host and one or more OpenCL devices (Devices). Devices are further divided into one or more compute units, which are even further divided into (SIMD) processing elements. The host is responsible for executing an application and assigning tasks to Devices. Figure 3.7 shows the platform model.

![OpenCL Platform Model](image)

Figure 3.7: OpenCL platform model.

The OpenCL program has two parts: host program and kernels. Host program is responsible for managing the kernel (which are similar to C functions) execution on Devices. Kernel itself is executed over index space, where each item in the index space is called *work-item*. Furthermore, work-items are organized into *work-groups*.

Finally, memory model (Figure 3.8) separates four different regions: global memory (accessible to all work-items), constant memory (constant region of global memory), local memory (accessible to a work-group), and private memory (accessible to a work-item).

The OpenCL provides a complete framework consistent of *OpenCL Platform layer*, *OpenCL Runtime* and *OpenCL Compiler*.

3.2.2 Compute Unified Device Architecture (CUDA)

The CUDA project [56] was introduced in November 2006 by NVIDIA. The CUDA architecture is hardware supported by the "CUDA cores" integrated into all modern NVIDIA GPUs. The architecture is shown in Figure 3.9 [58]. The key el-
elements of the CUDA architecture are the three lower (green) boxes presented in Figure: *CUDA Parallel Compute Engine inside NVIDIA GPUs* (CUDA cores), *CUDA Support in OS Kernel* (support for hardware initialization, configuration, etc.), and *CUDA Driver* (drivers that provide device-level APIs for developers).

What is convenient about the CUDA is that, although it is restricted to NVIDIA platforms it provides self-contained complete software development environment with support for different applications, but natively, applications written with CUDA
Driver API (which essentially presents a set of extensions to the C language).

The CUDA platform model is consisted of the Host and the Device [55], where Device is an NVIDIA CUDA GPU that runs as a coprocessor to the Host. Host should run the sequential program and leave the parallel work to Device.

The programming model is described in the following. The main programming instance is kernel (basically a C function). Kernel is executed over threads, which are hierarchically organized and can be grouped in thread blocks and thread blocks can be grouped in grids. The number of threads per block is restricted by the memory resources of a processor core [55]. While the thread blocks within a grid need to be independent, threads inside thread blocks can communicate and synchronize between each other. In addition to kernels, it is assumed that there is an appropriate host program that creates and enqueues kernels to Device.

Finally, the memory model. Each thread has a private local memory. Each thread block has its own shared memory visible to all threads of the block. There is also a global memory accessible to all threads. In addition to these three memory levels, all threads can also access to special read-only memories, which are optimized for special use. Both thread and memory hierarchy models are summarized in Figure 3.10.

3.2.3 Ct

Ct, C for Throughput Computing [35, 34], developed by Intel, is a deterministic general-purpose parallel programming model [23] that provides a set of APIs and extensions to the C++ language. It is intended to be a flexible parallel programming model for highly parallel architectures and a “new” C language for parallel computing.

Although it does not explicitly assume any particular underlying multicore platform model [67], it is primarily intended to support future Intel scalable multicores (Figure 3.11) [23, 41, 52].

Its SPMD (single program multiple data) programming model [52] basically presents a set of APIs and extensions to C++ object-oriented language. The extensions are focused on types and operators that efficiently exploit parallelism, mainly data-parallelism (SIMD). For instance, the main new data type is TVEC, which can allocate a vector of specified size in the memory. The operators used on this type are Ct operators, which include element-wise operators, some basic communication operators, etc. Since this high-level language hides the underlying hardware, the power of Ct lies in the Ct compiler and runtime, which goal is to minimize threading overhead and make efficient use of memory bandwidth.

As there is no assumed platform model, memory model is also not assumed. In fact, this software platform is still in the development phase and probably more details will be soon published.
3.2.4 Summary

It is immediately noticeable that OpenCL and CUDA are very similar in many ways. First of all, the platform model is very similar, although CUDA is targeted solely for NVIDIA GPUs, while OpenCL targets heterogeneous platforms, in general. Nevertheless, NVIDIA Tesla described previously (and GPUs in general) almost perfectly matches the OpenCL platform model. Concerning the programming model, the two platforms again use the same principles, although different terminology (NDRange against thread blocks and grids). The same holds for the memory model. Finally, NVIDIA is strongly supporting OpenCL development [61].

On the other hand, Ct is a general-purpose parallel programming model. It does
not take any assumption about the underlying parallel platform model and implements the programming model through the high-level APIs and C++ extensions, with a support of powerful Ct compiler and runtime.

We can see that all of these state of the art software tools in some way extend the widely-used C programming language, probably to have a good acceptance from the programmers, who are used to traditional C. Table 3.2 gives an overview of the previously described software tools.

<table>
<thead>
<tr>
<th>Software tool</th>
<th>Khronos group</th>
<th>NVIDIA CUDA</th>
<th>Intel Ct</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
<tr>
<td></td>
<td>Khronos group</td>
<td>NVIDIA CUDA</td>
<td>Intel Ct</td>
</tr>
</tbody>
</table>
|               | Khro
3.3.1 **Synthetic Aperture Radar (SAR) processing**

Synthetic aperture radar (SAR) is a radar that emulates long antenna length (that gives high spatial resolution) with small antenna length (that gives low spatial resolution) by synthetically combining radar observations from many different observations. The key idea is to move the small antenna over the area of interest. By processing all the taken observations, we can generate a high-resolution image for which we would need much larger static antenna. Since the SAR is the application of interest in this thesis, Chapter 4 explains general principles of radars and goes in more detail on SAR.

This radar idea, where small and portable antenna is used to observe areas for which we would need very large static antennas would be impossible without the large processing power, namely large throughput. In order to generate a complete image of the observed area from many radar observations, we need to process large amounts of data. In particular, for each voxel for which we want to calculate radar reflection, we need to go through all the observations to account them into the calculation of the voxel of interest. Since output voxels can be calculated independently, data-parallelism comes as a natural solution to the great amount of processing that is needed.

In this thesis we look over two important application areas of SAR: *air-to-ground surveillance* and *bodyscan*, whereas later application is the principal topic in this thesis. In air-to-ground surveillance, SAR is attached to an airplane that flies over the area of interest, while in the bodyscan application, SAR is used for scanning the human body by moving many small radars around it.

3.3.2 **Local Adaptive Contrast Enhancement (LACE) Analysis**

The need to properly adjust image contrast can vary depending on the application area. Our focus is on the military applications where contrast is adjusted in such a way that darker parts of the image are enlightened in order to get more information from darker parts, which will hopefully reveal new information.

The input for the algorithm is a grayscale (monochromatic) image and the output is also an image. The processing is done, as in many other image processing algorithms, by taking pixel by pixel and by looking at its neighbourhood in order to appropriately adjust the contrast of the pixel in consideration. Since processing can be done independently on each pixel, this algorithm is very suitable for data-parallelisation and thus for high-performance computing.

As mentioned, independent processing of pixels or group of pixels is possible in many image processing algorithms, often making high-performance computing platforms a very good match for applications from this area.
3.3.3 Summary

There is a very wide range of applications that are inherently parallel and listing the most important ones would take books for itself [70]. Using high-performance processors that have high throughput instead of traditional processors with lower throughput and higher clock frequency is particularly suitable for such kind of applications. Moreover, there are many more unexplored application areas that would significantly benefit from the modern high-throughput processors and many new ideas can nowadays be actual implemented when having so much throughput power. One such example is the SAR processing algorithm, which we explore in this thesis.

3.4 Look in the Future

After the overview of some of the major state of the art multicore developments, it is appropriate to discuss in this place how might the future of multicore systems look like, taking in concern historical developments and current technology trends.

It is obvious that popular trend of going towards higher frequencies is being replaced with the concept of parallelism, where performance is gained by using many cores/processors. Increased number of supporting software tools and developments that should ease parallel programming is another testimonial of that trend. Even the very application-specific processors, like Texas Instruments OMAP, are having more cores to satisfy higher computing demands.

Furthermore, from the current state of the art developments we can see that the architectures are becoming more heterogeneous. On the one hand, general-purpose computers are introducing more special-purpose hardware, to satisfy special (often signal-processing) demands. On the other hand, GPUs are adding more capabilities for general-purpose computing, which can be seen in a good way in the previously described NVIDIA Tesla processor. Finally, even the supporting software tools are becoming increasingly focused on these heterogeneity issues.

According to these trends, a new trend of general-purpose computation on graphics hardware (GPGPU) [1] is rising. It makes the graphics hardware, which is inherently parallel and has many cores available, programmable with classical programming techniques and utilities. It is also a trend that links the gap between general-purpose processors and high-performance processors.
Chapter 4

Synthetic Aperture Radar (SAR)

The focus of this thesis is on the Synthetic Aperture Radar (SAR) application, in particular on SAR processing.

The basic radar principles and terms, together with a list of radar application areas are explained in section 4.1. After introducing the basic radar principles we dive in the particularities of the SAR, in section 4.2. There, some elementary SAR features are described. In addition, in section 4.3 we take a closer look at the SAR application areas, with the emphasis on the air-to-ground surveillance and the bodyscan. Finally, in section 4.4, the gist of this paper, SAR processing, is detailed and the algorithm used in the following chapters is derived.

4.1 Radar Principles

Radar is an active electromagnetic system with the main function to detect and to measure distance to an object. Depending on the purpose of the radar in use, an object can be an aircraft, a ship, person or even a natural environment [72]. The term radar stands for *RAdar Detection And Ranging*, describing the primary radar purpose: measuring distance (range) to an object.

Essentially, radar transmits signals and measures reflection of the transmitted signal (Figure 4.1). All information about the object is determined from the echo. By measuring the time it takes a reflection to come back, we can calculate the distance. Combined with the angle of the received echo we can determine the location of the object. From successive echoes, we can also determine if the object is moving and at what speed. Unfortunately, in practice, the reflection always contains some undesired signals that need to be filtered, such as reflections from other objects, thermal noise, and interference from other transmitters. Filtering these undesired signals from the echo is the job of many radar processing algorithms.

Since range is the primary information obtained from the radar observations, we are interested in the range resolution of the radar. Radar range resolution is defined as the ability of the radar to distinguish between two close targets: the closer the targets are, the higher resolution is required for successful distinct detections. If,
Figure 4.1: Radar principle: Detecting and measuring distance from an aircraft.

for example, the distance between two objects is $1\,m$ and the range resolution is $5\,m$, we will not be able to distinguish these two objects.

Radar frequencies range from $100\,MHz$ to $35\,GHz$ [72]. Lower frequencies provide low resolution, but are suitable for long ranges, while high frequencies give a high resolution and short range.

Another important concept utilized in radars is the Doppler effect, which is used for detecting moving objects. Because of the frequency change that occurs when an object is moving (Doppler effect), we can determine the speed of the object. Together with the location and the range we can effectively determine the trajectory of the moving object.

Being capable to operate independently of weather conditions, day and night, radar has become a valuable tool in many application areas, a few of them listed below:

- **Military** - Military applications have been the main driver of radar development. One of the most common applications is in air, ground, and sea surveillance.

- **Law enforcement** - Speed radars are an essential tool for police traffic control. The main use of the speed radar is to measure speed of the vehicle based on the Doppler effect. Besides application in police, speed radars can be used for many other purposes, such as in sports.

- **Remote sensing** - Although the radar itself can be considered a remote sensor, in this context we think on environment sensing, such as:
  - Meteorology (weather forecast, observation of severe weather such as thunderstorms and tornadoes);
– Planetary observation (planet Earth is regularly being observed by space radars);
– Short-range below ground probing (radars can map the composition of the Earth’s crust);
– Mapping of sea ice (by knowing ice thickness efficient ship routing can be planned).

- Air traffic control - Radars are the main tool for traffic control and surveillance at every larger airport. Even in bad weather conditions, air traffic control can safely guide aircraft take-off and landing.

- Aircraft safety and navigation - Many airborne vehicles have their own radar to observe the surrounding for the presence of other aerial vehicles, to monitor the weather, etc.

- Ship safety - By using radars, especially during bad weather or night, ships can avoid collisions and observe navigation buoys, shore, and islands.

4.2 SAR Principles

Many different types of radars have been developed. Each type suits a particular purpose and has different parameters, such as type and frequency of emitted signal, type of antenna, etc.

Synthetic Aperture Radar [18, 71] is a type of radar that emulates a long (big) antenna length by moving a radar with a short (small) antenna length. In general, the longer the antenna is, the higher resolution of the observations we can get. However, SAR synthetically achieves a high resolution without using a large antenna. The antenna length emulated in this way can be up to several kilometres long. Figure 4.2 graphically represents the SAR principle. The synthetic length of the antenna is calculated from the point where data-collection starts to the point where the data collection ends. By coherently combining all the acquired data, we can achieve a high resolution.

As every radar, SAR is used to "range" objects. Therefore, resolution is a key performance measure. SAR resolution is expressed in two terms along two axes:

- Range (cross-track) resolution: the resolution along the range axis (cross-track). See Figure 4.3. It is calculated in the same way as the range resolution of any radar.

- Azimuth (cross-range) resolution: the resolution along the azimuth axis (along-track). The key benefit of SAR processing lies exactly here - many observations made along azimuth axis are coherently integrated and contribute to a high resolution radar reflection image.
There are several modes in which SAR can operate. The two most common modes are: strip map (swath mode) and spot (spotlight) mode. These two modes are symbolically represented in Figure 4.4.

In the strip map mode, successive observations are taken along the path antenna is moving, resulting in high-resolution images of the observed area that would, in the case of a static antenna, require a large antenna. In the spot mode, the antenna beam is steered to one point, and we can achieve a very high resolution of the
observed spot.

Until now, we have considered constructing 2D radar reflection images from the observations taken on a 1D path. However, we can also synthetically generate 3D radar reflection images from the observations taken from a 2D space. This can be done either by having more antennas moving along parallel paths or by having just one antenna moving in 2D space.

### 4.3 SAR Applications

Attractive SAR features described in the previous section give rise to many different application fields. Essentially, a SAR can be used whenever we need a small radar to generate high-resolution observations that are not possible to take with the antenna size of the given radar. Here we discuss two SAR applications central to this paper: the air-to-ground surveillance and the bodyscan. Both applications use the same SAR processing algorithm, and are conceptually the same. However, while for the air-to-ground surveillance the distance between the object and the radar is very long (in order of kilometres), in the bodyscan the distance between the radar and the human body is short (in order of decimetres).

SAR is a vital component in air-to-ground surveillance [75], where a SAR system is attached to an aerial vehicle. Figure 4.5 shows the *Uninhabited Aerial Vehicle Synthetic Aperture Radar* on a NASA Gulfstream III aircraft.

![Uninhabited Aerial Vehicle Synthetic Aperture Radar](image)

**Figure 4.5:** *Uninhabited Aerial Vehicle Synthetic Aperture Radar.*

Small unmanned aerial vehicles (UAV) for air-to-ground surveillance present a specially interesting field where the SAR has been successfully applied. The object of interest (target) is the Earth’s surface and the assumed mode is the strip map mode. This type of SAR is very attractive for military purposes, but can be used in many civil applications as well, such as short-range below ground probing.
As one example of a state of the art UAV we take a look at the ScanEagle SAR project [54]. ScanEagle is a small low-cost, long-endurance autonomous unmanned aerial vehicle [14] developed by Boeing and The Insitu Group. Its specification are given in [54]. SAR system specifically designed for the ScanEagle is an ultra small SAR system called NanoSAR, which specifications are given in [31]. Figure 4.6 shows both the ScanEagle UAV (on the left) and the NanoSAR (on the right).

![ScanEagle and NanoSAR](image)

Figure 4.6: Scan Eagle (on the left) and NanoSAR (on the right).

ScanEagle has been successfully used in many military missions, which include intelligence, surveillance, and reconnaissance (ISR), special services operations, escort operations, sea-lane and convoy protection, and protection of high-value and secure installations [54].

Another example of a SAR application is the bodyscan, where we scan the human body using a SAR system. Many information can be extracted from radar observations of the human body, most notably the presence of possibly concealed weapons. For example, nowadays bodyscanners are used as a part of the airport security checking procedure. Figure 4.7 shows two state of the art bodyscanners based on the use of radars, which have been used at several airports: ProVision™ Advanced Imaging Technology body scanner from L3 communications [16] (on the left) and Rapiscan Secure 1000 Single Pose from Rapiscan systems [73] (on the right).

Both scanners make 2D images of the human body with the aim to detect illegal objects. However, the main application of this thesis is 3D body scanner. Making a 3D image is significantly more difficult than making 2D image, because we need a radar that moves in two dimensions instead of one. At present, these types of SAR are still a part of active research and development.

### 4.4 Azimuth Compression (SAR Processing)

In general, SAR processing consists of the following steps:

- Pulse compression;
- Motion compensation;
• Azimuth compression;
• Autofocus;
• Geometric correction;
• Intensity scaling for display;

The heart of the SAR system lies in the azimuth compression where observations are coherently added to form a complete image. From now on, under the term SAR processing, we exclusively mean azimuth compression. For a particular spot in the observed space, radar reflectivity is calculated by combining the radar reflectivity of the spot from all the observations that include that spot.

Since the amount of data that needs to be processed is large, many algorithms have been developed to efficiently perform the calculation in terms of execution time and power consumption. Traditionally, some form of Fast Fourier Transformation (FFT) is used. Frequency domain is mainly used due to the simple arithmetic. However, in order to perform calculations in the frequency domain, it is required that data is organized in special data layouts. Data layouts obtained with SAR measurements usually have improper organization and extensive reordering is often required (motion compensation). As computational power significantly increased over time, it has become possible to do computations in the time-domain, without doing the usual approximations with frequency domain. Beside the fact that these calculations are very accurate, this offers some other benefits as well. For example, motion compensation is inherently included in the processing, thus meaning that data-reordering is not required.

Therefore, we will use a time-domain “brute-force” algorithm [75] that looks as follows:

Let us define the $k$-th antenna position observation or radar observation with $s_k[r]$, where $r$ is a range (distance) of a particular output spot in the current antenna...
position observation. Each antenna position observation is taken from a specific (known) position, which is stored as three-dimensional antenna position $\vec{a}_k$, associated with each antenna position observation. Furthermore, each antenna position observation itself presents an array of complex, pulse compressed samples, meaning that each element represents one observed sample (indexed with the range $r$), simply called an observation. Figure 4.8 tries to graphically present these variables, which are for the sake of simplicity represented on the air-to-ground surveillance application example.

![Figure 4.8: SAR algorithm basic variables.](image)

Range is calculated from the antenna position, $\vec{a}_k$, and the position in the space for which we want to calculate radar reflectivity, $\vec{w}$.

$$r_k(\vec{w}) = |\vec{w} - \vec{a}_k|$$

Knowing the range, we need to calculate one-way phase shift ($\phi_k(\vec{w})$), that is used for calculation of the two-way phasor ($\theta_k(\vec{w})$):

$$\phi_k(\vec{w}) = 2\pi \frac{r_k(\vec{w})}{\lambda}$$

$$\theta_k(\vec{w}) = e^{2j\phi_k(\vec{w})}$$

where $\lambda$ is the wavelength of the transmitted wave.

The equation that calculates the radar reflectivity of the spot (voxel) from one antenna position observation looks as follows:

$$v_k[\vec{w}] = s_k[r_k(\vec{w})] \times \theta'_k(\vec{w})$$
Finally, by summing up the radar reflectivity of the spot, for the desired spot, from all the observations, we get that the overall radar reflectivity of the spot is:

\[ v [\vec{w}] = \sum_{k \in K_{\vec{w}}} v_k [\vec{w}] \]

where \( K_{\vec{w}} \) represents the set of all relevant observations, observations that include the desired spot (in the world, for air-to-ground application, or at the human body, for the bodyscan application).

For more details about SAR processing and SAR in general, the interested reader is refereed to [18].
Chapter 5

Application

In this chapter we present the bodyscan application in detail and analyse data sets. In the first part we introduce the overall system requirements, while in the second part we describe the test environment.

5.1 System Requirements

The application, namely the bodyscan, scans the human body in order to generate a 3D human body representation.

The bodyscan works on the SAR principle collecting observations of the human body from several antennas. Theoretically, one antenna that moves in the 3D space around the body would be enough, but this imposes several implementation difficulties. For instance, that antenna should move both around the body and along the path parallel to the body, which is not practical to implement. Moreover, the measurement time would be unacceptably long, taking at least a few minutes to make the complete measurement. In order to do correct real-time measurements, the system needs to be able to make the measurements in a few seconds.

We will see in the test environment how the very simple measurements can be done. It is also important to notice that for the sake of simplicity we will consider that an antenna is both transmitting and receiving the signal. This is a valid situation, but in practice we will use two antennas: transmitter and receiver. More precisely, in practice, the body scan can be implemented by using two independent arrays of antennas: a vertical linear array and a horizontal circular array (Figure 5.1). The vertical linear array consists of antennas moving vertically, while the horizontal circular array consists of antennas moving around the human body. By combining the observations from these measurements, with the SAR principle described in Section 4.4, we can obtain a complete 3D image of the human body.

Let us assume the most simple case, where we have only one antenna. In order to achieve the required output resolution, we need to move the antenna in certain step sizes, in both vertical and circular direction. The measurement process looks as follows: the antenna is moving in circular direction ($\phi$) in certain step sizes, $\Delta \phi$, ...
Figure 5.1: In order to practically perform 3D body measurements using the SAR principle, two arrays of antennas are used: vertical linear array and horizontal circular array.

and in each position the antenna also traverses the vertical path \( H \) in certain step sizes, \( \Delta H \). The overall number of \textbf{antenna positions}, \( |a| \), is equal to the number of antenna positions along the vertical path \( |H| \) times the number of antenna positions along the circular path \( |\phi| \):

\[
|a| = |H| \times |\phi|
\]  

(5.1)

Besides \( |H| \), \( \Delta H \), \( |\phi| \), and \( \Delta \phi \), there are several other important parameters that characterize the data sets. Other two important antenna parameters are the number of observations that antenna collects in each position, \( |R| \), and the proper step size of each observation, \( \Delta R \). These parameters are determined according to the antenna bandwidth. These six parameters completely define the input data set in terms of memory requirements and performance analysis. We consider that an \textbf{input sample} is equivalent to an observation, \textit{obs}. Therefore, the \textbf{number of observations}, \( |\text{obs}| \), is equal to the number of antenna positions times the number of observations at each antenna position:

\[
|\text{input}| = |\text{obs}| = |a| \times |R| = |H| \times |\phi| \times |R|
\]  

(5.2)
The output data set, i.e., the radar reflections obtained from the radar observations, is determined according to the output that we want to observe. The coordinate system that we use for representing output samples is the Cartesian $3D$ coordinate system (Figure 5.2), compared to the polar coordinate system that naturally describes the antenna positions. Since we need to have a single reference system, in our calculations we choose the Cartesian $3D$ coordinate system. This means that we need to convert polar coordinates to $3D$ Cartesian coordinates. To define the output area in three dimensions, we need three parameters for each dimension: $min$ - the first sample, $max$ - the last sample, and $step$ - the step size.

![Figure 5.2: The 3D Cartesian coordinate system of human body that is also the referent system used for all the calculations.](image)

According to these three parameters, we will use the following variables to uniquely represent the observed output area:

- **X axis**: $X_{min}$ (minimal value of $X$), $X_{max}$ (maximal value of $X$), and $\Delta X$ (step size of $X$);
- **Y axis**: $Y_{min}$ (minimal value of $Y$), $Y_{max}$ (maximal value of $Y$), and $\Delta Y$ (step size of $Y$);
- **Z axis**: $Z_{min}$ (minimal value of $Z$), $Z_{max}$ (maximal value of $Z$), and $\Delta Z$ (step size of $Z$).

As an **output sample**, $out$ we consider a radar reflection at a certain position. The overall number of output samples, $|out|$ is:

$$|out| = \frac{X_{max} - X_{min}}{\Delta X} \times \frac{Y_{max} - Y_{min}}{\Delta Y} \times \frac{Z_{max} - Z_{min}}{\Delta Z}$$  \hspace{1cm} (5.3)
The key parameter that determines the number of both input and output samples is the output resolution that we want to achieve, together with the size of the observed area of interest.

In this section we introduced the key variables that define the input and the output data set. These variables are used throughout this thesis. In the following section we look at some concrete numbers to estimate application memory requirements.

5.2 Data Set Analysis

The output (observed) area needs to be large enough to accommodate the general dimensions of a human body. Therefore, the output area is chosen to be $2m$ high and with $1.5m$ diameter (see Figure 5.1). If we consider the referent 3D coordinate system of the human body (Figure 5.2), we choose a resolution of $0.005m$ in all three dimensions. This resolution is one of the key system requirements. In order to achieve that resolution we need sufficiently small antenna step sizes. From the interviews with the SAR experts, we found out that following antenna step sizes are sufficient to achieve the required output resolution:

$$
\Delta H = 0.0025m \\
\Delta \phi = 0.1^\circ \\
\Delta R = 0.005m
$$

Table 5.1 summarizes these antenna parameters.

<table>
<thead>
<tr>
<th>Input parameters</th>
<th>Observed area</th>
<th>Step size</th>
<th>Number of samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>$H$</td>
<td>$2m$</td>
<td>$0.0025m$</td>
<td>800</td>
</tr>
<tr>
<td>$\text{Phi}$</td>
<td>$360^\circ$</td>
<td>$0.1^\circ$</td>
<td>3600</td>
</tr>
<tr>
<td>$R$</td>
<td>$1.5m$</td>
<td>$0.005m$</td>
<td>300</td>
</tr>
</tbody>
</table>

Table 5.1: Required input parameters for obtaining $0.005m$ resolution in each dimension of the output area.

Let us consider the memory requirements of the input data set. Each antenna position can be represented as a single-precision floating point number (float). The number of bytes required for a float is 4B [30]. Therefore, to store all the antenna positions (we have three parameters per position), we need:

$$
\text{mem}(|a|) = |H| \times |\phi| \times 3 \times \text{sizeof(float)} \\
= 800 \times 3600 \times 12B \\
\approx 33\text{MB}
$$
Since we have two antennas (receiver and transmitter), we need twice as much memory. Thus, the final memory required for all the antenna positions, is approximately 66MB. Moreover, since per one antenna position we get |R| observations, where each observation is a complex number that can be represented as two floats, the overall memory requirement for the input is:

\[
mem(|\text{obs}|) = |a| \times |R| \times 2 \times \text{sizeof}(\text{float}) \\
= |H| \times |\phi| \times |R| \times 2 \times 4B \\
= 800 \times 3600 \times 300 \times 8B \\
\approx 6.5\text{GB}
\]

Let us now take a look at the output memory requirements. As stated before, an output sample (radar reflection) is a processed radar observation at a certain position, in the human body coordinate system. The processed observation is calculated according to the SAR processing formulas given in section 4.4. The number of output samples depends on the output area we want to observe and the resolution we want to achieve. Consider the "worst-case" scenario where we want to observe a window of sizes 2m times 1m times 1m (which is a window where most of the humans fit), with the resolution of 0.005m in all three dimensions. This means that we want a scan of the complete human body. The memory requirement is the amount of memory needed to store one 3D position plus the radar reflection. For each dimension, we need one single-precision floating point number (three floats in total), while the processed output is represented as a complex float (two floats), meaning that five floats are needed in total. Therefore, the memory requirements for the output are:

\[
mem(|\text{out}|) = \left(\frac{X_{\text{max}} - X_{\text{min}}}{\Delta X}\right) \times \left(\frac{Y_{\text{max}} - Y_{\text{min}}}{\Delta Y}\right) \times \left(\frac{Z_{\text{max}} - Z_{\text{min}}}{\Delta Z}\right) \times 5 \times \text{sizeof}(\text{float}) \\
= \frac{200 \times 200 \times 400 \times 20B}{0.005m} \\
\approx 306\text{MB}
\]

It is easy to see that the input memory requirements are considerably higher than the output memory requirement:

\[
\left|\frac{\text{input}}{\text{output}}\right| = \left(\frac{\frac{X_{\text{max}} - X_{\text{min}}}{\Delta X}}{X_{\text{max}} - X_{\text{min}}}\right) \times \left(\frac{\frac{Y_{\text{max}} - Y_{\text{min}}}{\Delta Y}}{Y_{\text{max}} - Y_{\text{min}}}\right) \times \left(\frac{\frac{Z_{\text{max}} - Z_{\text{min}}}{\Delta Z}}{Z_{\text{max}} - Z_{\text{min}}}\right) \times 5 \times \text{sizeof}(\text{float})
\]
With the real system memory requirements in mind, in the next section we select representative test cases.

### 5.3 Test Environment

The test environment is an experimental test bench pictured in Figure 5.3. The test environment is simpler than the real system and uses smaller data sets. This leads to shorter measurement and processing time and thus easier testing and development.

![Test Environment Diagram](image)

Figure 5.3: The experimental test environment is composed of two antennas (receiver and transmitter) and the object under observation. The object rotates around its central axis and for each object position, transmitting antenna traverses a vertical path (parallel to the object), taking radar observations in each step. The receiving antenna is standing still.

The test bench contains one transmitting antenna (transmitter) and one receiving antenna (receiver). The transmitter can move along a vertical axis, while the receiver is standing still. To simulate the rotative motion of the receiver, the object is rotated around its central axis. The measurements are performed in such way that during each object position, the transmitter (the moving antenna) traverses a vertical path, which effectively has the same effect as if the antenna was moving along the vertical axis and around the test object. A model of the test bench is represented in Figure 5.4. The figure also presents the coordinate systems used: polar...
coordinate system of the antenna is the \((H, \phi)\)-coordinate system, while the 3D Cartesian coordinate system of the test object is the \((x, y, z)\)-coordinate system, with the 0 point in the centre of the measurement area.

![Figure 5.4: Model of the body-scanner with the test object. Two coordinate systems are represented: polar coordinate systems for antenna positions and Cartesian 3D coordinate system for the test object.](image)

Finally, two different measurement setup are used. First measurement setup, Setup 1, has fewer radar observations with lower precision, while the second measurement setup, Setup 2, has very high number of observations with high precision. The first setup is used for simpler processing, while the second setup is used for more complex processing. The purpose of different measurement setup will be clear in the following chapters where we discuss specific implementations. The measurement setup are given in tables 5.2 and 5.3.

<table>
<thead>
<tr>
<th>Input parameters</th>
<th>Observed area</th>
<th>Step size</th>
<th>Number of samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>(H)</td>
<td>(\approx 0.5m)</td>
<td>0.0045</td>
<td>109</td>
</tr>
<tr>
<td>(\phi)</td>
<td>80.5(^{\circ})</td>
<td>0.5(^{\circ})</td>
<td>161</td>
</tr>
<tr>
<td>(R)</td>
<td>1.5m</td>
<td>0.00125m</td>
<td>1200</td>
</tr>
</tbody>
</table>

Table 5.2: Input parameters of the measurement setup 1.

According to the parameters given in the Table 5.2, the memory requirements for the input data, for Setup 1, are:

- Number of antenna positions: \(109 \times 161 = 17549\). Memory requirement for
Table 5.3: Input parameters of the measurement setup 2.

<table>
<thead>
<tr>
<th>Input parameters</th>
<th>Observed area</th>
<th>Step size</th>
<th>Number of samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>$H$</td>
<td>0.64m</td>
<td>0.001</td>
<td>64</td>
</tr>
<tr>
<td>$\phi$</td>
<td>$\approx$ 360$^\circ$</td>
<td>0.25$^\circ$</td>
<td>1441</td>
</tr>
<tr>
<td>$R$</td>
<td>3m</td>
<td>0.0009375m</td>
<td>3200</td>
</tr>
</tbody>
</table>

storing positions of one antenna: $17549 \times 12B \approx 205$KB. Since there are two antennas, the total memory requirement is $\approx 410$KB.

- Number of input samples: $109 \times 161 \times 1200 = 21058800$. Memory requirements for all the observations: $21058800 \times 8B \approx 161$MB.

The memory requirement for Setup 2 (Table 5.3) are:

- Number of antenna positions: $64 \times 1441 = 92224$. Memory requirement for storing positions of one antenna: $92224 \times 12B \approx 1081$KB. Since there are two antennas, the total memory requirement is $2161.5$KB.

- Number of input samples: $64 \times 1441 \times 3200 = 295116800$. Memory requirements for all the observations: $295116800 \times 8B \approx 2252$MB.

Now, when the measurement setup is explained, let us look at the test objects, output areas of interest, and output memory requirements. The test object used for Setup 1 is a metal cylinder, which can be seen in Figure 5.3. Its model is presented in Figure 5.5. In addition, in Figure 5.6 we show the physical representation of all the output parameters introduced in the previous section on the cylinder test object. The test object used for Setup 2 is "Eva", a metal model of human body in natural dimensions.

In order to appropriately observe the whole cylinder we need to observe the appropriate output area. The dimensions of the cylinder are: $1.35m \times 0.16m \times 0.16m$. However, in reality we do not know the exact test object dimensions, and we always need to observe the larger area. Taking that into concern, Table 5.4 summarizes the output parameters required for appropriately observing the whole cylinder. The table also presents the memory requirements for the output data.

Comparing the test input data and test output data requirements from the table we get that input data is $\left|\frac{\text{input}}{\text{output}}\right| = 18.80$, which is close enough to the real ratio, 21.6.

Concerning the second test object, Eva, the output requirements are as for the real case (table 5.1), since Eva represents a model of human body in natural dimensions. The only difference compared to the real case is that we can observe smaller area since we precisely know the dimensions in all directions.

However, since to process these amounts of data (even the most powerful CPUs of today require tens of hours) we need to limit ourselves to simpler test cases that would allow us reasonably fast testing. Therefore, we constraint ourselves to four
Figure 5.5: The model of the cylinder used as a test object with the area being observed.

Figure 5.6: Physical representation of the important output parameters of the cylinder test object.

test cases: two test cases for simpler processing (processing on CPUs) and two test cases for more powerful processing (processing on CPUs and GPUs). Their specification, together with the memory requirements, is given in tables 5.5 and 5.6. Test cases 1, 2 and 3 use Setup 1, while test case 4 uses Setup 2. Especially important is test case 4 since it presents the most accurate model of the real situation.

In all the test cases, except the test case 3, the overall memory requirements are dominated by the input data, similar to the real case (where input is approximately 6.5GB and the output is approximately 306MB). Table 5.7 summarizes the input and output memory requirements for the test cases and gives the ratio between the
Table 5.4: The output area that we need to observe in order to appropriately capture the complete cylinder. We choose a resolution of 0.005m.

<table>
<thead>
<tr>
<th>Test case 1</th>
<th>Test case 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>-0.1m</td>
</tr>
<tr>
<td>Y</td>
<td>-0.1m</td>
</tr>
<tr>
<td>Z</td>
<td>-0.7</td>
</tr>
<tr>
<td>out</td>
<td>448000</td>
</tr>
<tr>
<td>mem(out)</td>
<td>448000 × 20B ≈ 8.55MB</td>
</tr>
</tbody>
</table>

Table 5.5: The observed output areas used for test cases 1 and 2 and their memory requirements.

<table>
<thead>
<tr>
<th>Test case 3</th>
<th>Test case 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>-0.2m</td>
</tr>
<tr>
<td>Y</td>
<td>-0.2m</td>
</tr>
<tr>
<td>Z</td>
<td>-0.5</td>
</tr>
<tr>
<td>out</td>
<td>20000000</td>
</tr>
<tr>
<td>mem(out)</td>
<td>20000000 × 20B ≈ 381.5MB</td>
</tr>
</tbody>
</table>

Table 5.6: The observed output areas used for test cases 3 and 4 and their memory requirements.

<table>
<thead>
<tr>
<th>Test case 1</th>
<th>Test case 2</th>
<th>Test case 3</th>
<th>Test case 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input data</td>
<td>410KB + 161MB ≈ 162MB</td>
<td>2162KB + 2252MB ≈ 2253MB</td>
<td></td>
</tr>
<tr>
<td>Output data</td>
<td>125KB</td>
<td>3750KB</td>
<td>381.5MB</td>
</tr>
<tr>
<td>Overall</td>
<td>162MB</td>
<td>166MB</td>
<td>543.5MB</td>
</tr>
<tr>
<td></td>
<td>1316.17</td>
<td>43.87</td>
<td>0.3</td>
</tr>
</tbody>
</table>

Table 5.7: Memory requirements of the test cases. |Input|Output| shows the ration between input test data and the output test data.

On the one hand, we have test cases 1 and 2. Test case 1 is a very small test case
in terms of processing requirements and we can also see that its \( \frac{\text{Input}}{\text{Output}} \) ratio is far from what we expect in reality (1316.17 vs. 21.6). Test case 2 represent much more demanding test case in terms of processing compared to the test case 1 and its ratio is close enough to what is expected in reality (43.87 vs. 21.6). On the other hand, we have test cases 3 and 4, which require substantially higher processing power compared to the test cases 1 and 2. Test case 3 does not correspond very well to the real case, but it is used as an extreme exercise of the processor throughput power. Finally, test case 4, which have considerably larger input data set (due to the measurement setup 2) and the output set that is even smaller than in test case 3, represents the test case that corresponds the best to the real case. The four test cases are used in the following chapters for evaluating and comparing the solutions.
Chapter 6

Generic Sequential and Parallel Solutions

This chapter presents the design and implementation of several solutions for the application described in Chapter 5. We give an overview of the common algorithm, we describe the solution evaluation, together with its underlying hardware and software details. Further, we discuss each of the solutions in detail.

Thus, Section 6.1 presents the platforms and metrics used for the experiments along this thesis. In section 6.2, we take a look at the sequential implementation (reference solution) of the algorithm, using it as a prototype to define application requirements in terms of the computation throughput and memory bandwidth. In Section 6.3, we parallelize the sequential version using the OpenMP model; these parallel solutions are described.

6.1 Platforms and Metrics

In this section we present the target hardware platforms and the metrics used to evaluate and compare various implementation variants, called solutions.

6.1.1 Experimental Setup and Platforms

In order to measure algorithm performance bottlenecks, each solution has been benchmarked. We inspect platform utilization and performance for each solution. In this way, we can compare solutions and determine the potential bottlenecks.

Several different machines are used for running and profiling the solutions. In particular, we have used the following host processors: Intel® Core™ 2 Duo 6300 [42, 43, 46], Intel® Xeon™ Processor E5320 [48, 47], and Intel® Core™ i7-920 Processor [45, 44]. Table 6.1 lists some of the main processor features together with the random-access memory (RAM) details. Processor features are collected from the datasheets, while the DRAM bandwidth has been calculated with the STREAM tool [51].
<table>
<thead>
<tr>
<th>Processor model</th>
<th>Intel Core 2 6300</th>
<th>Intel Xeon E5320</th>
<th>Intel Core i7-920</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Superscalar, out of order</td>
<td>Superscalar, out of order</td>
<td>Superscalar, out of order</td>
</tr>
<tr>
<td>Core count</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Thread count</td>
<td>2 ((2 \times 1))</td>
<td>8 ((4 \times 2))</td>
<td>8 ((4 \times 2))</td>
</tr>
<tr>
<td>Frequency</td>
<td>1.86GHz</td>
<td>2.67GHz</td>
<td></td>
</tr>
<tr>
<td>Cache size</td>
<td>2MB</td>
<td>8MB</td>
<td>8MB</td>
</tr>
<tr>
<td>Throughput</td>
<td>14.88GFLOPS</td>
<td>29.76GFLOPS</td>
<td>42.56GFLOPS</td>
</tr>
<tr>
<td>DRAM bandwidth</td>
<td>2922MB/s</td>
<td>4547MB/s</td>
<td>13419MB/s</td>
</tr>
</tbody>
</table>

Table 6.1: Host processors and RAM specifications.

Furthermore, we use several different GPUs: NVIDIA® Tesla™ C1060 [64], NVIDIA® GeForce™ GTX 280 [63], and ATI® Radeon™ HD 5870 Graphics [5]. Specifications of the GPUs are given in Table 6.2.

<table>
<thead>
<tr>
<th>Processor model</th>
<th>NVIDIA Tesla C1060</th>
<th>NVIDIA GeForce GTX 280</th>
<th>ATI Radeon HD 5870</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core count</td>
<td>30</td>
<td>30</td>
<td>20</td>
</tr>
<tr>
<td>Thread count</td>
<td>240</td>
<td>240</td>
<td>N.A.</td>
</tr>
<tr>
<td>Processor frequency</td>
<td>1.3GHz</td>
<td>1.3GHz</td>
<td>850MHz</td>
</tr>
<tr>
<td>Local memory</td>
<td>16KB</td>
<td>16KB</td>
<td>32KB</td>
</tr>
<tr>
<td>Global memory</td>
<td>4GB</td>
<td>1GB</td>
<td>256MB</td>
</tr>
<tr>
<td>Throughput</td>
<td>933GFLOPS</td>
<td>933GFLOPS</td>
<td>544GFLOPS</td>
</tr>
<tr>
<td>Memory bandwidth</td>
<td>102GB/s</td>
<td>141.7GB/s</td>
<td>153.6GB/s</td>
</tr>
<tr>
<td>Memory frequency</td>
<td>800MHz</td>
<td>1107MHz</td>
<td>1.2GHz</td>
</tr>
<tr>
<td>Communication with the host</td>
<td>PCI Express x16 Gen2</td>
<td>16GB/s</td>
<td>/</td>
</tr>
</tbody>
</table>

Table 6.2: GPUs.

Given the available target platforms, the following sections present a reference sequential implementation of the target application, as well as multiple OpenMP parallel solutions.

6.1.2 Metrics

Each experiment consists of the solution under consideration, one target platform and one test case. There are three major solutions: the reference sequential solution, OpenMP solutions, and OpenCL solutions, six different platforms consisting of GPPs and GPUs, and four test cases that largely differ in parameters and sizes. In order to compare the results and evaluate different solutions on different hardware platform, we use four metrics.

1. Execution time, $T$, as the main optimization criteria from the industrial/prag-
matic point of view. We measure the execution time of the main processing part by using the "wall clock".

2. Speedup, $S$, for relative and scalability comparisons:

$$ S = \frac{T_{\text{sequential}}}{T_{\text{parallel}}} \quad (6.1) $$

We also use a "normalized" speedup, $S_n$, where the speedup is divided by the number of cores. Different types of multicore systems significantly differ in the complexity and the number of cores. For instance, GPPs have fewer, complex cores, while GPUs have many, simple cores. With this in mind, we use this metric to give a better insight into the speedup when different multicores are compared.

3. Throughput, for absolute comparisons. We use two throughput metrics:

(a) Computational throughput, FLOPS: the application arithmetic complexity calculated as the number of floating-point operations (FLOPs) per second:

$$ \text{FLOPS} = \frac{\text{FLOPs}}{T[s]} \quad (6.2) $$

(b) Memory throughput, MBw: the application memory bandwidth calculated as the number of total memory reads from the main memory (download) and memory writes to the main memory (upload), in bytes per second:

$$ \text{MBw} = \frac{\text{download}[B] + \text{upload}[B]}{T[s]} \quad (6.3) $$

6.2 Reference Solution

In this section we discuss the structure, implementation, and performance of a reference sequential implementation of the bodyscan application.

In the previous work [75], a basic sequential implementation of the application for real-time brute force SAR processing of radar observations taken from an aircraft was parallelised on a Cell Broadband Engine Architecture (for details on Cell, refer to subsection 3.1.4). The implementation gave good speedup compared to the sequential solution and the possibility of real-time processing. Moreover, with
minor changes, the same implementation has been used for the bodyscan SAR processing. However, due to a very platform-dependent trimming of a sequential solution for the Cell processor, and due to the fact that the application modelling for more generic multicore systems has not been performed, the implementation is not portable. Moreover, the real-time requirements of the bodyscan application require lower execution time and thus more processing power than the Cell can offer. For this reason we reversed engineered the current Cell implementation in order to obtain the reference solution, which could be used as the base solution for different parallel implementations. We removed all the platform-dependent code from the implementation in order to produce a reference sequential implementation. In this way we can analyse the algorithm and find guidelines for further parallelization and application modelling.

The high-level implementation structure of the reference implementation is presented in Figure 6.1. In the **Pre-processing**, variables are initialized, the input file with observations is read into the memory (**Read observations**), the antenna positions are calculated from the input file (**Read antenna positions**) and the output vector is initialized (**Initialize output**). The main data processing (presented in the **Processing** part of Figure 6.1) is all done by a function called **Compress**. The function iterates through the output vector. In each iteration it calculates one output sample (one radar reflection) by iterating through all the antenna positions and taking relevant observation from each antenna position. The process of calculating a contribution to an output sample from an observation is done in the function **contrib** according to the formulas presented in Section 4.4.

This means that we have two nested loops:

- The main (outer) loop: iterates through the output vector;
- The inner loop: iterates through the antenna positions.

This can be represented in pseudo code with the following code segment:

```pseudo
def compress(output, inputs):
    for i in range(N):
        for j in range(M):
            output[i] += contrib(inputs[j])
```

Figure 6.2 graphically presents the data layout of the application. In each antenna position we make a number of observations - antenna position observations. Antenna position observation is an array of observations indexed by the distance from the observed output sample. Antenna positions and antenna position observations present the input data. Each output sample is represented with two values: a position in the 3D Cartesian coordinate system and a radar reflection (calculated using one observation from each antenna position observation).

### 6.2.1 Benchmarking and Profiling

Profiling is a technique where we measure the execution time of different parts of a program in order to estimate their contribution to the overall execution time.
In this way, we can determine the program bottlenecks and invest our efforts in optimizing the most time-consuming program parts. This is usually an efficient way for improving the overall program performance.

Profiling is done on an Intel Core 2 CPU 6300 (see Table 6.1) with the GNU
gprof profiler [2]. The timer resolution is 0.01 seconds. Tables 6.3 and 6.4 show timing information for test cases 1 and 2 (described in subsection 5.3). The columns in the table represent the following:

- Function contribution - The percentage of the total running time of the program used by this function;
- Individual time - The number of seconds accounted for by this function alone;
- Cumulative time - A running sum of the number of seconds accounted for by this function and those listed above it;
- Number of calls - The number of times this function was invoked;
- Individual time/calls - The average time spent in this function per call;
- Cumulative time/calls - The average time spent in this function and its descendents per call.

<table>
<thead>
<tr>
<th>Function name</th>
<th>Function contribution [%]</th>
<th>Individual time [s]</th>
<th>Cumulative time [s]</th>
<th>Number of calls [calls]</th>
<th>Individual time/calls [s/call]</th>
<th>Cumulative time/calls [s/call]</th>
</tr>
</thead>
<tbody>
<tr>
<td>contrib</td>
<td>67.71</td>
<td>21.79</td>
<td>21.79</td>
<td>115138989</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Compress</td>
<td>7.41</td>
<td>2.39</td>
<td>24.18</td>
<td>1</td>
<td>2.39</td>
<td>31.70</td>
</tr>
</tbody>
</table>

Table 6.3: Profile of the reference solution for the test case 1.

<table>
<thead>
<tr>
<th>Function name</th>
<th>Function contribution [%]</th>
<th>Individual time [s]</th>
<th>Cumulative time [s]</th>
<th>Number of calls [calls]</th>
<th>Individual time/calls [Ks/call]</th>
<th>Cumulative time/calls [Ks/call]</th>
</tr>
</thead>
<tbody>
<tr>
<td>contrib</td>
<td>67.83</td>
<td>679.44</td>
<td>679.44</td>
<td>3454169670</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Compress</td>
<td>6.73</td>
<td>67.42</td>
<td>746.86</td>
<td>1</td>
<td>67.42</td>
<td>987.11</td>
</tr>
</tbody>
</table>

Table 6.4: Profile of the reference solution for the test case 2.

The tables only present the timing information for the most time-consuming functions, which are the functions that do the main processing. The next most time-consuming functions are the ones from the standard library that operate on complex numbers. They are not included in the list because they are irrelevant from the optimization point of view (no optimization will be performed on those functions). Therefore, our primary effort for parallelisation should be focused on functions Compress and contrib. Function contrib is the processing heart of the system and does arithmetic operations on the input data, in order to compute output data. This means that it is suitable for low-level parallelisation. Function Compress, the iterative part of the application, built using two nested loops, is suitable for high-level
parallelizations. In this work we mainly focus on the high-level parallelisation of the function *Compress*.

In addition to the profiling information we show the execution time in Table 6.5. Execution is measured on both Intel Core 2 CPU 6300 and Intel Xeon Processor E5320 using the Linux library function `gettimeofday`.

<table>
<thead>
<tr>
<th>Test case 1</th>
<th>Intel Core 2 CPU 6300</th>
<th>Intel Xeon Processor E5320</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test case 1</td>
<td>61 s</td>
<td>55 s</td>
</tr>
<tr>
<td>Test case 2</td>
<td>1833 s</td>
<td>1557 s</td>
</tr>
</tbody>
</table>

Table 6.5: Execution time of the reference solution.

In the next section we discuss the *Compress* function and its efficient parallel algorithms.

### 6.2.2 Performance Evaluation

In order to discover application bounds, we measure the application requirements in terms of the computational intensity and memory footprint [77].

The following expression gives the general application requirements in terms of the number of required floating-point operations (FLOPs), which determines the computational intensity. Since the number of FLOPs in the preprocessing part is negligible compared to the number of FLOPs in the main processing part, we approximate the overall number of FLOPs by calculating it only for the main processing part. The precise FLOPs calculation is given in Appendix A.

\[
\text{FLOPs} = (\text{Number of times function } contrib \text{ is called from the function } Compress.) \times \\
(\text{Computational intensity of the function } contrib.) \\
= |H| \times |\phi| \times \\
\left( \frac{X_{\text{max}} - X_{\text{min}}}{\Delta X} \right) \times \left( \frac{Y_{\text{max}} - Y_{\text{min}}}{\Delta Y} \right) \times |Z| \times \\
[9 \times F_{\text{add}} + 8 \times F_{\text{sub}} + 13 \times F_{\text{mul}} + 2 \times F_{\text{div}} + \sin (F) + \cos (F) + 2 \times \sqrt{F}] \\
\]

Essentially, \( |H| \times |\phi| \) represents the number of antenna positions, while \( \left( \frac{X_{\text{max}} - X_{\text{min}}}{\Delta X} \right) \times \left( \frac{Y_{\text{max}} - Y_{\text{min}}}{\Delta Y} \right) \times |Z| \) represents the number of output samples.

The meaning of the other parameters is as follows:

- \( F_{\text{add}}, F_{\text{sub}}, F_{\text{mul}}, \) and \( F_{\text{div}} \) - number of operations for single-precision floating point addition, subtraction, multiplication, and division, respectively.

57
• \( \sin(F), \cos(F) \) - number of operations for single-precision floating point sine and cosine functions, respectively. Since these functions are library functions, their computation intensity needs to be measured in a special way, which will be detailed along the profiling information of each solution.

• \( \sqrt{F} \) - number of operations for square-root of a single-precision floating-point number. Square-root is also a library function, like the sine and cosine functions, and we detail its FLOPs count later.

Using the platform specific values for the above parameters and the measured execution time, we can derive the number of floating point operations per second (FLOPS). The main problem in calculating the number of FLOPs is the calculation of the FLOPs for the library functions, \((\sin, \cos \text{ and } \sqrt{\cdot})\). Several methods were used in order to precisely determine the number of required floating-point operations, but none of them gave reliable results. In particular, two Linux profiling tools were used:

• OProfile [4];
• Perfmon2 [20].

Both tools present an interface to x86 performance counters [39], which cannot be directly accessed from the user code due to the protection modes [38]. While we were not able to make the proper installation of perfmon2, the OProfile tool gave unexpected results and we decided to abandon the attempt to precisely measure FLOPS. We used values given in [68, 74]:

• \( F_{add} \) - 1 FLOP;
• \( F_{sub} \) - 1 FLOP;
• \( F_{mul} \) - 1 FLOP;
• \( F_{div} \) - 15 FLOPs;
• \( \sin(F) \) - 20 FLOPs;
• \( \cos(F) \) - 20 FLOPs;
• \( \sqrt{F} \) - 15 FLOPs.

Although [74] does not give the number of FLOPs for sine and cosine, it gives the number of FLOPs for the \( \exp \) function, which can be used as a good estimation for the trigonometric functions \( \sin \) and \( \cos \), due to the Euler’s formula: \( e^{ix} = \cos x + j \times \sin x \). The number of FLOPs for \( F_{div} \) should be less than the number of floating point operation for the \( \sqrt{F} \) and therefore we took the worst-case of 15 FLOPs for the \( F_{div} \) operation. It is important to notice that these values represent the upper (safe) bound on the number of FLOPs.
With the given values, we calculate the number of FLOPs that our application requires: 14.97 GFLOPs for test case 1, and 449.04 GFLOPs for test case 2. Furthermore, using the execution time from Table 6.5, and according to the equation 6.2, we calculate FLOPS. Results are summarized in Table 6.6.

<table>
<thead>
<tr>
<th>Test case</th>
<th>FLOPS</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>245 MFLOPS</td>
<td>1.65%</td>
</tr>
<tr>
<td>2</td>
<td>245 MFLOPS</td>
<td>1.65%</td>
</tr>
</tbody>
</table>

Table 6.6: Floating point operations per second (FLOPS) of the reference solution.

By comparing the peak hardware throughput with the measured throughput, we get that the hardware compute units are heavily under-utilized (< 2%). One of the reasons of such low utilization is that we use only one of the several cores available on the CPU and only one thread of execution (Intel Core 2 has 2 cores and can support up to two threads, while Intel Xeon E5320 has 4 cores and can support up to 8 threads). Nevertheless, even if the utilization would linearly scale-up with the number of cores, it would still be very low.

The application memory requirements are also calculated. The theoretical upper bound on the number of memory reads and writes is approximated by the memory requirements of the main processing part. This is a reasonable approximation, since the memory communication of the preprocessing part is negligible compared to the memory communication of the main processing part. Table 6.7 shows the resulting application memory requirements.

<table>
<thead>
<tr>
<th></th>
<th>floating-point</th>
<th>integer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>read</td>
<td>write</td>
</tr>
<tr>
<td>\textit{Compress}</td>
<td>3 \times</td>
<td></td>
</tr>
<tr>
<td>\textit{contrib}</td>
<td>47</td>
<td>27</td>
</tr>
</tbody>
</table>

Table 6.7: Application memory requirements. Parameter $|\text{obs}|$ represents the number of antenna positions, while parameter $|\text{out}|$ represents the number of output samples.

By knowing the execution time and by using a standard representation for integers (4B) and single precision floating-point numbers (4B) [30] we are able to calculate the application memory requirements in terms of overall required upload and download and the memory bandwidth (number of bytes per second). These are represented in Tables 6.8 and 6.9.

The tables show that besides the under-utilized computational throughput, we even have under-utilized memory throughput. The reason behind this is that we utilize only one core and one thread of execution, and each core utilizes only one
<table>
<thead>
<tr>
<th></th>
<th>Download [GB]</th>
<th>Upload [GB]</th>
<th>Overall [GB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test case 1</td>
<td>25.5911</td>
<td>12.4389</td>
<td>36.03</td>
</tr>
<tr>
<td>Test case 2</td>
<td>707.7318</td>
<td>373.1672</td>
<td>1080.9</td>
</tr>
</tbody>
</table>

Table 6.8: Application memory requirements in terms of overall download and upload.

<table>
<thead>
<tr>
<th></th>
<th>Intel Core 2 CPU 6300 (peak: 2922MB/s)</th>
<th>Intel Xeon Processor E5320 (peak: 4547MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MBw</td>
<td>Utilization</td>
</tr>
<tr>
<td>Test case 1</td>
<td>604MB/s</td>
<td>20.66%</td>
</tr>
<tr>
<td>Test case 2</td>
<td>605MB/s</td>
<td>20.66%</td>
</tr>
</tbody>
</table>

Table 6.9: Application memory bandwidth utilization in MB/s.

part of the memory bandwidth. This means that the memory bandwidth that can be efficiently used is far below the peak hardware memory bandwidth.

In the next section, we see that when all the threads of the execution are used, both computational and memory throughput increase, and ultimately, the system is memory bound.

### 6.3 OpenMP Solutions

The OpenMP API presents an easy way to parallelise sequential code, by using a set of pragmas and APIs. It supports multi-platform shared-memory parallel programming in C/C++ and Fortran on mainstream architectures, including Unix platforms and Windows NT platforms [3].

By using OpenMP pragmas, the reference solution is parallelised by using symmetrical data distribution. More precisely, three parallel solutions have been developed:

1. Output data partitioning (parallelising the main loop);
2. Input data partitioning (parallelising the inner loop);
3. Parallelising the computations inside the `contrib` function.

Among these three solutions, only the first two gave improvements.

The first option parallelises the main loop in the function `Compress`, which effectively means that the output data set is partitioned. Each thread gets a portion of outputs to process. Since each output can be calculated completely independently, there is no need for special synchronization and communication between threads.

The second option parallelises the inner loop, which means that the input data is partitioned. Each thread gets a chunk of observations to iterate through. In this
way, in each iteration, each thread calculates a part of an output sample, requiring synchronization between threads to prevent data races (multiple threads trying to write to the same variable in the same time). In OpenMP, this is implemented by enclosing the critical region with a special `critical` pragma. In addition, another variant of this solution has been implemented, where instead of using the `critical` pragma, the reduce operator on the output is used. However, although this variant of the solution performs significantly better than the first one, the reduce operator gave output with unsatisfactory accuracy and this variant was abandoned.

The third parallel solution tried to run concurrently the two code sections inside the `contrib` function. In particular, since in the function we need to calculate the distance of the observed output sample from both antennas and then to average the result, this parallelisation does the antenna calculations concurrently. However, this led to an increase in the execution time by a factor of 13 compared to the reference solution. We will not consider this parallelisation further.

### 6.3.1 Experiments and Results

Table 6.10 shows timing information of the OpenMP parallel solutions, while Figures 6.3, 6.4, 6.5, and 6.6 graphically present the information from this table.

<table>
<thead>
<tr>
<th>Solution</th>
<th>Test case 1</th>
<th>1</th>
<th>62</th>
<th>2</th>
<th>32</th>
<th>4</th>
<th>55</th>
<th>8</th>
<th>28</th>
<th>1</th>
<th>14</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Test case 2</td>
<td>2</td>
<td>1843</td>
<td>4</td>
<td>956</td>
<td>5</td>
<td>1564</td>
<td>8</td>
<td>23</td>
<td>3</td>
<td>95</td>
<td>206</td>
</tr>
<tr>
<td>Solution 2</td>
<td>Test case 1</td>
<td>2</td>
<td>7</td>
<td>1</td>
<td>49</td>
<td>4</td>
<td>64</td>
<td>8</td>
<td>63</td>
<td>3</td>
<td>111</td>
<td>113</td>
</tr>
<tr>
<td></td>
<td>Test case 2</td>
<td>2</td>
<td>2121</td>
<td>4</td>
<td>1459</td>
<td>4</td>
<td>1828</td>
<td>8</td>
<td>1801</td>
<td>3</td>
<td>3209</td>
<td>3350</td>
</tr>
</tbody>
</table>

Table 6.10: Execution time ($T$ [s]) of the OpenMP solutions.

![Figure 6.3: Execution time of the parallel solution 1 for the test case 1.](image)
Table 6.11 shows the speedup of the OpenMP parallel solutions compared to the reference solution, while Figures 6.7 and 6.8 give graphical representation of the data from this table.

The tables show that the output data partitioning (the first parallel solution) is very scalable and leads to almost perfect speedup. The input data partitioning (the second parallel solution) is not scalable. The reason behind this is that the more threads we have, the more time we lose on synchronization.
Table 6.11: speedup time of the OpenMP solutions.

| Solution 1 | Test case 1 | 0.99 | 1.92 | 1 | 1.99 | 3.93 | 7.39 |
| Solution 1 | Test case 2 | 0.99 | 1.92 | 1 | 1.89 | 3.94 | 7.57 |
| Solution 2 | Test case 1 | 0.86 | 1.25 | 0.86 | 0.88 | 0.49 | 0.49 |
| Solution 2 | Test case 2 | 0.86 | 1.26 | 0.85 | 0.86 | 0.48 | 0.46 |

Figure 6.7: speedup of the parallel solution 1 compared to the reference solution. Figure on the top presents the results for the test case 1, while the figure on the bottom presents the results for the test case 2.

It is interesting to notice in the previous figures, for the second parallel solution, that the execution time and the speedup are almost the same when the program is run on one and two threads, and on four and eight threads. This is due to the fact that the Intel Xeon processor has four cores and is capable of executing eight threads in parallel. Therefore, when we use one or two threads, just one core is utilized, while when we use four or eight threads, all four cores are utilized.

Let us now take a look at the throughput of the solutions. Tables 6.12 and 6.13 summarize the computational throughput and the memory throughput, respectively.

Since the scalable parallel solution 1 is the most interesting in terms of perfor-
Figure 6.8: speedup of the parallel solution 2 compared to the reference solution. Figure on the left presents the results for the test case 1, while the figure on the right presents the results for the test case 2.

Table 6.12: Computational throughput of the solutions.

<table>
<thead>
<tr>
<th></th>
<th>Intel Core 2 CPU 6300 (peak: 14.88GFLOPS)</th>
<th>Intel Xeon Processor E5320 (peak: 29.76GFLOPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Number of threads</td>
<td>Number of threads</td>
</tr>
<tr>
<td><strong>Solution 1</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test case 1</td>
<td>0.242</td>
<td>0.471</td>
</tr>
<tr>
<td>Test case 2</td>
<td>0.243</td>
<td>0.469</td>
</tr>
<tr>
<td><strong>Solution 2</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test case 1</td>
<td>0.211</td>
<td>0.307</td>
</tr>
<tr>
<td>Test case 2</td>
<td>0.212</td>
<td>0.308</td>
</tr>
</tbody>
</table>

From the figures, looking at the case where all the processor threads (and thus all cores) are utilized, we can conclude that the application is memory bound. Moreover, the fact that the memory utilization goes above 100% is expected, since we calculate memory bandwidth without taking into concern the effect of caching and
Table 6.13: Memory throughput of the solutions.

<table>
<thead>
<tr>
<th></th>
<th>Intel Core 2 CPU 6300 (peak: 2922MB/s)</th>
<th>Intel Xeon Processor E5320 (peak: 4547MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Number of threads</td>
<td>Number of threads</td>
</tr>
<tr>
<td>Solution 1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Test case 1</td>
<td>597</td>
<td>1160</td>
</tr>
<tr>
<td>Test case 2</td>
<td>600</td>
<td>1156</td>
</tr>
<tr>
<td>Solution 2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Test case 1</td>
<td>520</td>
<td>757</td>
</tr>
<tr>
<td>Test case 2</td>
<td>522</td>
<td>758</td>
</tr>
</tbody>
</table>

Figure 6.9: The computational throughput utilization of the second parallel solution for test case 2 for Intel Core 2 Duo and Intel Xeon.

Figure 6.10: The memory throughput utilization of the second parallel solution for test case 2 for Intel Core 2 Duo and Intel Xeon.

both processors have two-level cache.

It is also important to notice some details on the test cases used for the measurements. As can be seen, only test cases 1 and 2 were used. Test case 1 is only used to give a rough estimation of the solution bottlenecks. Once identified, these are explored with test case 2. One might also think that although test case 2 corresponds
well to a real case in terms of the ratio between input and output samples, it is still a very small test case, and thus not representative. However, this is not true. The key issue when using the GPP and the main memory is the ratio between the input and the output, since the memory size does not play a critical role\textsuperscript{1}. Therefore, all the results scale linearly with the problem size, since the processing behaviour remains the same.

In the next chapter, we turn to another implementation approach, where the most intensive program parts (functions \textit{Compress} and \textit{contrib}) are offloaded to GPU. Using GPUs means that we have much more processing power at our disposal, but we are also more memory bound; thus, test cases 1 and 2 are not sufficient anymore. There, we will focus on test cases 3 and 4, which represent a problem size that more closely corresponds to a real case and that better exposes the application bottlenecks.

\textsuperscript{1}The real case has input of 6.4GB, which modern computer architectures can easily fit in the main memory.
Chapter 7

OpenCL Solutions

In this chapter we discuss our OpenCL-based solutions for the 3D bodyscan application. Our goals are threefold: (1) to have a portable OpenCL version of the target application, (2) to have a fully optimized OpenCL version for the NVIDIA GPUs, and (3) to investigate the portability vs. performance trade-off that OpenCL brings for platforms like the NVIDIA GPUs.

We use the solutions presented in Chapter 6 as references for introducing a portable, OpenCL solution.

OpenCL is an open industry standard for programming a heterogeneous collection of GPPs, GPUs, and other discrete computing devices, organized as a single platform. A description of the OpenCL is given in Subsection 3.2.1. The standard is very generic and supports many types of platforms.

In our implementations, we use three different platforms (Tables 6.1 and 6.2):

- Intel Xeon Processor E5320 as host processor and an NVIDIA Tesla C1060 [59, 64] GPU;
- Intel Xeon Processor E5320 as host processor and an NVIDIA GeForce GTX 280 [63] GPU;
- Intel Core i7-920 as host processor and an ATI Radeon HD 5870 Graphics.

For all NVIDIA GPUs we use the NVIDIA OpenCL implementation [65], while for the ATI GPU and Intel i7 we use the AMD OpenCL implementation [6].

Figure 7.1 shows the platforms we target in the OpenCL bodyscan application. We start from the NVIDIA GPUs (available at the time we started the work) and we extend the target group to ATI GPUs and even to GPPs.

The structure of an OpenCL solution for a GPU is as follows:

- The host processor does all the preprocessing and plays the role of the GPU manager;
- Functions Compress and contrib are offloaded on the GPU. An OpenCL kernel is created to run the contrib function, while the function Compress does
not have a special implementation, but is implicit when the host processor enqueues the kernels to work-items. In other words, the outer loop of the function \texttt{Compress}, which iterates through the output is effectively implemented in enqueuing the kernels to the GPU, while the inner loop of the function \texttt{Compress}, which iterates through the observations and calls the \texttt{contrib} function is implemented in the OpenCL kernel.

Note that in this chapter we will often use the term work-item. This term is originally introduced in the OpenCL specification \cite{OpenCL} to define the smallest unit of the execution. It is an instance of the kernel that is executed on one or more processing elements as a part of the work-group. Each processing element can execute one work-item at a time. In our explanations, we use the term interchangeably between a software (bodyscan application) and a hardware (specific platforms) meaning. When describing software, ”work-item” is used in its original meaning. When describing hardware, we consider a ”work-item” to be a thread of execution. For GPPs, that would literally mean one thread, whereas each core can usually execute one or two threads, while for the GPUs that would mean one SIMD processor inside the core (a processing element).

Our OpenCL solutions differ in the work distribution (what is each work-item executing) and in the kernel implementation (where the kernel optimizations are applied). By analysing different solutions, we explore the important application parameters that influence performance.

Since the solutions were developed continuously, each new solution was derived based on the experimental results (and their analysis) of the previous implementations. In the following sections, the solutions are described in the order they were developed. Each solution is supported with relevant figures, comparisons, and discussion on the advantages and disadvantages.

First, in Sections 7.1 and 7.2, we developed a set of solutions that focus on portability, and can be easily transferred and executed on different hardware platforms. In Section 7.3, we turn to implementations that focus on performance and which are more platform-dependent. In this way we also discover the trade-offs between portability and performance when using OpenCL (section). Finally, Section 7.4, summarizes all the parallel solutions in terms of various metrics and Section 7.5
qualitatively compares programmability, performance, portability, and productivity of different solutions.

7.1 Reference OpenCL Solution

Our OpenCL implementation, reference OpenCL solution, was implemented by partitioning the output data, such that each output sample is processed individually. It is the solution with the maximal portability and the finest work-item granularity. The kernel of the reference OpenCL solution processes one output sample. In other words, each work-item is responsible for processing one output sample. This OpenCL solution gave remarkable speedup compared to the previous ones. In fact, it was so fast that the test cases used for the previous solutions are too small to precisely measure the execution time. Table 7.1 shows the execution time of test cases 1 and 2 on the NVIDIA GPUs (see table 6.2).

<table>
<thead>
<tr>
<th></th>
<th>GTX 280</th>
<th>Tesla C1060</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test case 1</td>
<td>0.12s</td>
<td>0.12s</td>
</tr>
<tr>
<td>Test case 2</td>
<td>1.73s</td>
<td>1.61s</td>
</tr>
<tr>
<td>Test case 3</td>
<td>165s</td>
<td>153s</td>
</tr>
<tr>
<td>Test case 4</td>
<td>160s</td>
<td>156s</td>
</tr>
</tbody>
</table>

Table 7.1: Execution time of the reference OpenCL solution on the NVIDIA GPUs.

In order to properly estimate the performance of the OpenCL solutions and to compare it with the best OpenMP parallel solution, we run all the solutions on a third test case, test case 3. This test case has considerably more output samples (more than 3000 times versus test case 1 and more than 100 times versus test case 2). Table 7.2 summarizes the execution times of the representative solutions on test case 3. The speedup is given compared to the reference solution.

We can see from the table that the reference OpenCL solution on Tesla is 137 times faster than the best OpenMP solution, running on the 8 threads of the Xeon processor. Also, note that compared to the optimized Cell implementation, the reference OpenCL solution on NVIDIA Tesla is 7 times faster.

In order to be able to cope with large inputs, we developed a solution where the input data (observations) can be partitioned into chunks. It uses almost the same kernel as the reference solution, with the difference that in each iteration over the new input data chunk, the output needs to be accumulated on the previously calculated output. Interestingly, the execution time of this solution is similar to the reference solution even when the input was divided into a larger number of chunks (the execution time variation was ±3% for number of chunks between 2 and 32). It is also interesting to note that when the input was divided into smaller number of chunks.

\(^1\)Note that along this entire chapter, the execution time of the reference sequential solution was not measured, but is estimated according to the previous results and comparisons between OpenMP solutions and the reference solution (see Table 6.11).
Table 7.2: Comparison between the reference solution, OpenMP solution 1, Cell solution, and the reference OpenCL solution. OpenMP solution 1 is run on the Xeon processor, Cell solution is run on the Cell BE processor, while the reference OpenCL solution is run on the NVIDIA GPUs.

<table>
<thead>
<tr>
<th>Solution</th>
<th>Reference</th>
<th>OpenMP</th>
<th>Cell</th>
<th>OpenCL</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Xeon (1 thread)</td>
<td>Xeon (8 threads)</td>
<td>Cell GTX 280</td>
<td>Tesla C1060</td>
<td></td>
</tr>
<tr>
<td>$T$</td>
<td>$167144$ s (2days)</td>
<td>$20893$ s (6h)</td>
<td>$1071$ s (18min)</td>
<td>$165$ s (3min)</td>
<td>$153$ s (3min)</td>
</tr>
<tr>
<td>$S$</td>
<td>1</td>
<td>8</td>
<td>156</td>
<td>1013</td>
<td>1092</td>
</tr>
<tr>
<td>$S_n$</td>
<td>1</td>
<td>1</td>
<td>26</td>
<td>34</td>
<td>36</td>
</tr>
</tbody>
</table>

7.2 Portability

Before going in any deeper analysis of the performance of the reference OpenCL solution on NVIDIA GPUs, we investigate portability of the reference OpenCL solution on a GPP and an ATI GPU. The used GPP is the Intel Core i7-920 and the used ATI GPU is the ATI Radeon HD 5870 Graphics.

We run the reference OpenCL solution on the GPP and compare it with the best OpenMP solution. The results are summarized in Figure 7.2.

Figure 7.2: Comparison between the best OpenMP solution and the reference OpenCL solution on Intel Core i7-920 for test cases 3 and 4.

The speedups of the reference OpenCL solution to the best OpenMP solution are outstanding 2.54 times for test case 3 and 2.16 for test case 4. We explain this large
improvement in the execution time by the fact that the OpenCL solution is implicitly parallel and allows better interleaving of the computation and the memory communication.

We also run the same experiments on an ATI GPU. The comparative results are summarized in Table 7.3.

<table>
<thead>
<tr>
<th>Test case</th>
<th>$T$ [s]</th>
<th>$S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>201s</td>
<td>0.80</td>
</tr>
<tr>
<td>4</td>
<td>216s</td>
<td>0.74</td>
</tr>
</tbody>
</table>

Table 7.3: Execution time of the reference OpenCL solution on the ATI GPU and comparison with the NVIDIA GTX 280.

Finally, in Tables 7.4 and 7.5 we overview the results of the reference OpenCL solution on all hardware platforms and show the speedup compared to the reference sequential solution.

<table>
<thead>
<tr>
<th>Sol.</th>
<th>Reference</th>
<th>OpenMP</th>
<th>Cell</th>
<th>OpenCL</th>
<th>OpenCL</th>
<th>OpenCL</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proc.</td>
<td>Itanium</td>
<td>Itanium</td>
<td>Cell</td>
<td>Itanium</td>
<td>NVIDIA GTX</td>
<td>NVIDIA Tesla</td>
<td>ATI Radeon</td>
</tr>
<tr>
<td>$T$</td>
<td>115112s (1.3days)</td>
<td>14389s (4h)</td>
<td>1071s (18min)</td>
<td>5661s (1.6h)</td>
<td>160s (2.67min)</td>
<td>153s (2.55min)</td>
<td>201s (3.35min)</td>
</tr>
<tr>
<td>$S$</td>
<td>1</td>
<td>8</td>
<td>107</td>
<td>20</td>
<td>719</td>
<td>752</td>
<td>573</td>
</tr>
<tr>
<td>$S_n$</td>
<td>1</td>
<td>1</td>
<td>18</td>
<td>2</td>
<td>24</td>
<td>25</td>
<td>29</td>
</tr>
</tbody>
</table>

Table 7.4: Execution times and speedups of all parallel implementations (OpenMP, Cell, OpenCL) on various platforms, for test case 3. $S$ and $S_n$ are calculated against the reference sequential solutions.

<table>
<thead>
<tr>
<th>Sol.</th>
<th>Reference</th>
<th>OpenMP</th>
<th>Cell</th>
<th>OpenCL</th>
<th>OpenCL</th>
<th>OpenCL</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proc.</td>
<td>Itanium</td>
<td>Itanium</td>
<td>Cell</td>
<td>OpenCL</td>
<td>OpenCL</td>
<td>NVIDIA GTX</td>
<td>NVIDIA Tesla</td>
</tr>
<tr>
<td>$T$</td>
<td>115544s (1.3days)</td>
<td>14433s (4h)</td>
<td>/</td>
<td>6675s (1.85h)</td>
<td>160s (2.67min)</td>
<td>156s (2.6min)</td>
<td>216s (3.6min)</td>
</tr>
<tr>
<td>$S$</td>
<td>1</td>
<td>8</td>
<td>/</td>
<td>17</td>
<td>719</td>
<td>741</td>
<td>535</td>
</tr>
<tr>
<td>$S_n$</td>
<td>1</td>
<td>1</td>
<td>/</td>
<td>3</td>
<td>24</td>
<td>25</td>
<td>27</td>
</tr>
</tbody>
</table>

Table 7.5: Execution times and speedups of all parallel implementations (OpenMP, Cell, OpenCL) on various platforms, for test case 4. $S$ and $S_n$ are calculated against the reference sequential solutions.

In summary, OpenCL proves to be able to provide a portable solution that have enviable performance. In the following section, we shall see that a few basic optimizations can lead to an even bigger gain in performance without the loss in portability. We also see that platform-specific optimizations that decrease the portability and aim to further increase performance on a specific platform can be very difficult.
7.3 Optimizations and Tuning

In this section we dive in GPU-specific optimizations and tuning. In particular, our focus is mainly on NVIDIA GPUs, available at the time of the research. First, we look on some basic optimizations, applicable to all GPU-based solutions. After developing the basic OpenCL solution, which is an extension of the reference OpenCL solution, we revisit performance evaluation and conclude that the application is memory-bound on the GPUs. We continue by analysing data layouts and to specific data-dependent optimizations that have a goal to reduce the memory traffic and thus increase the application performance on GPUs.

7.3.1 Basic Optimizations

Let us first take a look at the two simple optimizations, one numerical and one structural: native arithmetic instructions and optimized memory allocation of the observations.

Every GPU has a set of native arithmetic instructions like \( \sin \), \( \cos \), and \( \sqrt{\text{ } } \). The major advantage of using native functions is faster execution. The drawback is the loss of accuracy. After implementing the reference kernel with the native GPU instructions (OpenCL has integrated support for native instructions) we got 10%-30% lower execution time in all cases. The loss in accuracy is acceptable.

We have also used compiler math optimizations, by using the OpenCL compiler options such as -cl-fast-relaxed-math and -cl-mad-enable [49], that allow “relaxed” math and enable multiply-add units, respectively, but neither of two brought any improvement.

Table 7.6 summarizes the execution times of the test cases 3 and 4 for the reference OpenCL solution with native arithmetic instructions on the GPUs.

<table>
<thead>
<tr>
<th></th>
<th>NVIDIA GTX</th>
<th>NVIDIA Tesla</th>
<th>ATI Radeon</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test case 3</td>
<td>82</td>
<td>113</td>
<td>58</td>
</tr>
<tr>
<td>Test case 4</td>
<td>100</td>
<td>159</td>
<td>67</td>
</tr>
</tbody>
</table>

Table 7.6: The impact of native arithmetic instructions on the execution time of test cases 3 and 4 on the GPU platforms. Speedup is measured against the reference OpenCL solution.

Another basic optimization is related to the memory allocation of the observations. Initially, the observations were stored as an array of complex numbers, which are pairs of floating point numbers. However, if we split the observation array into two arrays, one array holding real part of the observations and another holding the imaginary part, we get a performance improvement between 10% and 30% compared to the reference OpenCL solution with native arithmetic instructions. Table 7.7 shows the execution time of this solution (reference OpenCL solution with
native instructions and observation splitting) and compares it with the reference OpenCL solutions with native arithmetic instructions.

<table>
<thead>
<tr>
<th></th>
<th>NVIDIA GTX</th>
<th>NVIDIA Tesla</th>
<th>ATI Radeon</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$T$ [s]</td>
<td>$S$</td>
<td>$T$ [s]</td>
</tr>
<tr>
<td>Test case 3</td>
<td>75</td>
<td>1.07</td>
<td>103</td>
</tr>
<tr>
<td>Test case 4</td>
<td>84</td>
<td>1.19</td>
<td>125</td>
</tr>
</tbody>
</table>

Table 7.7: The impact of data structure splitting on the execution time of test cases 3 and 4 on the GPU platforms. Speedup is measured against the reference OpenCL solution with native arithmetic instructions.

We also experimented with splitting the arrays that hold antenna positions into separate arrays, meaning that from one array holding structures of three values (x, y, and z) we made three arrays. However, that did not lead to improvement.

The reason behind the gain when the observations are split must lies in the way the data is arranged in the global GPU memory - memory coalescing and alignment [62] are essential for the performance of the GPU memory system.

Since using native instructions and splitting the observation array do not reduce portability (when there is no support for native instructions, "normal" functions are called), we extend the reference OpenCL solution with the optimizations and call the new solution basic OpenCL solution. Tables 7.8 and 7.9 overview the results of the basic OpenCL solution on all hardware platforms and show the speedup compared to the reference sequential solution.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Itanium</td>
<td>115112s</td>
<td>14389s</td>
<td>1071s</td>
<td>5661s</td>
<td>75s</td>
<td>103s</td>
<td>57s</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>(1.3days)</td>
<td>(4h)</td>
<td>(18min)</td>
<td>(1.6h)</td>
<td>(1.25min)</td>
<td>(1.72min)</td>
<td>(1min)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>8</td>
<td>107</td>
<td>20</td>
<td>1535</td>
<td>1118</td>
<td>2019</td>
<td>101</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>18</td>
<td>2</td>
<td>51</td>
<td>37</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.8: Execution times and speedups of the basic OpenCL solution and other parallel implementations (OpenMP, Cell) on various platforms, for test case 3. $S$ and $S_n$ are calculated against the reference sequential solutions.

We can see from all these results that although the reference OpenCL solution is slower on the ATI GPU compared to the NVIDIA GPU, when the optimizations are included, the ATI GPU performs better than the NVIDIA GPU. In particular, this major improvement is due to the native arithmetic instructions (99%), whereas observation splitting brought almost no improvement.

In summary, the development of our first OpenCL solution was fairly simple and the basic optimizations led to a very good improvement in the execution time. However, we will see in the following that diving into deeper, platform-specific optimizations can be very difficult. This supports the claim that OpenCL was built
Table 7.9: Execution times and speedups of the basic OpenCL solution and other parallel implementations (OpenMP, Cell) on various platforms, for test case 4. $S$ and $S_n$ are calculated against the reference sequential solutions.

7.3.2 Performance Evaluation Revisited

Before we dive into more structural optimizations, we re-analyse some of the characteristics of the application. First, we recompute the computational intensity of the application. The overall computation intensity equals to the number of kernels times the number of FLOPs per kernel. The number of kernels, on the other hand, is equal to the number of output samples. We use the number of FLOPs according to the GPU FLOP values given in [74]:

- $F_{add}$ - 1 FLOP;
- $F_{sub}$ - 1 FLOP;
- $F_{mul}$ - 1 FLOP;
- $F_{div}$ - 1 FLOP;
- $\sin(F)$ - 1 FLOP;
- $\cos(F)$ - 1 FLOP;
- $\sqrt{F}$ - 1 FLOP.

Thus, we get that each kernel utilizes 36 FLOPs per antenna position iteration.

Table 7.10 shows the number of GFLOPs and the number of GFLOPS for test cases 3 and 4, for the NVIDIA GPUs, together with the measure of utilization.

The global memory requirements of the reference OpenCL solution, in terms of the memory communication, are expressed with the following formulas:

- Global memory read and write per kernel:

$$
reads = 12B + |a| \times 32B \\
\approx |a| \times 32B \\
writes = 8B
$$
Table 7.10: Number of GFLOPs for test case 3 and test case 4 and the GFLOPS and utilization for the NVIDIA GPUs.

<table>
<thead>
<tr>
<th></th>
<th>GTX 280</th>
<th>Tesla C1060</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GFLOPs</td>
<td>GFLOPS Utilization</td>
</tr>
<tr>
<td>Test case 3</td>
<td>12762</td>
<td>113</td>
</tr>
<tr>
<td></td>
<td>156</td>
<td>13%</td>
</tr>
<tr>
<td>Test case 4</td>
<td>12079</td>
<td>76</td>
</tr>
</tbody>
</table>

Table 7.11 summarizes the memory utilization of the NVIDIA GPUs based on the memory requirements and the execution times given in Table 7.6.

<table>
<thead>
<tr>
<th></th>
<th>mem. requirements</th>
<th>bandwidth</th>
<th>utilization</th>
<th>bandwidth</th>
<th>utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test case 3</td>
<td>10.21TB</td>
<td>127GB/s</td>
<td>88%</td>
<td>90GB/s</td>
<td>89%</td>
</tr>
<tr>
<td>Test case 4</td>
<td>9.76TB</td>
<td>102GB/s</td>
<td>72%</td>
<td>64GB/s</td>
<td>63%</td>
</tr>
</tbody>
</table>

Table 7.11: Estimation of the global GPU memory bandwidth and utilization for the basic OpenCL solution on the NVIDIA GPUs.

We calculate the application operational intensity as the number of arithmetic operations per byte of memory communication. In the case of GPUs, when we say memory, we refer to the global GPU memory. Moreover, in the case of the GPUs, for our application, the operational intensity is equal to the kernel operational intensity, which is exactly what we are interested in. The operational intensity equals:

\[ \frac{|a| \times 36 \text{FLOPs}}{|a| \times 32B} = \frac{9 \text{ FLOPs}}{8B} = 1.125 \frac{\text{FLOPs}}{B} \]

Using the application operational intensity, we draw the roofline model [77] for our application on NVIDIA GPUs. Figure 7.3 presents this model.

The roofline model presents the platform bounds. Placing the application operational intensity on it gives an indication on how the application behaves on the given platform. The top bound is the maximal computational throughput of the...
platform (the horizontal line), while the steep line on the left represents the memory bound. Depending on where the application reaches the "roof", we say that the application is computation or memory bound on the platform. In our case, it is clear that the application is memory bound.

After making the previous static analysis of the application, let us also look at the results obtained from the OpenCL profiler (summarized in Table 7.12) and compare them with the results of the static analysis.

In general, we can observe that computational and memory utilization are close to what is statically calculated (the difference is less than 10%). Moreover, from the occupancy numbers, we can see that the implicit distribution model gives good results, mapping the work-items almost optimally. Table 7.13 summarizes the computational and memory intensity obtained from the static analysis and from the OpenCL profiler, for the NVIDIA GPUs and test case 4.

To conclude, the bodyscan application is memory-bound on the given NVIDIA GPUs. In general, application-wise, there are two ways to fight this bound. We can either use a platform with higher memory bandwidth, or we can try to optimize the application such that we increase its operational intensity. In turn, increasing application operational intensity can be done in two ways: we can either increase computation or reduce memory traffic. Typically, an increase in computation comes only with (significant) algorithmic changes, while memory traffic decrease is obtained by increasing data re-use. We try to address both these techniques for our bodyscan application.
Table 7.12: Profiling information of the basic OpenCL solution with the native instructions for NVIDIA GPUs and test cases 3 and 4.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Tesla</th>
<th>Tesla</th>
<th>GeForce</th>
<th>GeForce</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test case</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>global mem read [GB/s]</td>
<td>44.3893</td>
<td>45.2828</td>
<td>56.7382</td>
<td>65.7497</td>
</tr>
<tr>
<td>global mem write [GB/s]</td>
<td>24.2494</td>
<td>17.4326</td>
<td>8.95724</td>
<td>11.0487</td>
</tr>
<tr>
<td>total mem throughput</td>
<td>68.6387</td>
<td>62.7153</td>
<td>65.6954</td>
<td>76.7984</td>
</tr>
<tr>
<td>instruction throughput</td>
<td>0.116937</td>
<td>0.0840648</td>
<td>0.128976</td>
<td>0.0739341</td>
</tr>
<tr>
<td>nd Range X</td>
<td>40401</td>
<td>9050</td>
<td>40401</td>
<td>9050</td>
</tr>
<tr>
<td>work group sizeX</td>
<td>500</td>
<td>402</td>
<td>500</td>
<td>402</td>
</tr>
<tr>
<td>static private mem per work group</td>
<td>52B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>registers per work-item</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>occupancy</td>
<td>1</td>
<td>0.812</td>
<td>1</td>
<td>0.812</td>
</tr>
<tr>
<td>work-groups per compute unit</td>
<td>4041</td>
<td>905</td>
<td>4041</td>
<td>905</td>
</tr>
</tbody>
</table>

Table 7.13: Comparison between the static analysis results and the OpenCL profiler results for the test case 4 on the NVIDIA GPUs. Difference is calculated as Static - Profiler.

<table>
<thead>
<tr>
<th></th>
<th>Tesla</th>
<th>GeForce</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computational utilization</td>
<td>8%</td>
<td>13%</td>
</tr>
<tr>
<td>Memory utilization</td>
<td>63%</td>
<td>72%</td>
</tr>
</tbody>
</table>

7.3.3 Re-computing Antenna Positions

Since the application is memory bound it is worthwhile to look at the algorithm to see if there is something that could be re-computed instead of loaded from the global memory. That is exactly the case with the antenna positions. In the body scan processing we know in advance the positions of both the antennas and the object under the observation. Therefore, instead of pre-calculating the positions according to geometries in the pre-processing part and then load them from the memory in the main processing part, we could simply re-compute them inside the kernel.

Note that the antenna and the object positions are often not known in advance for other SAR-like applications. For instance, when SAR is used in air-surveillance, we usually do not know the exact path of the aircraft and the positions are usually calculated by the GPS and stored in a special file. In that case we cannot re-compute the positions.

Since in the reference OpenCL solution, the output positions are loaded only once from the global memory, while the antenna positions are loaded for every iteration of each output, we re-compute only the antenna positions inside the kernel.

We run both the reference and the basic OpenCL solution with the new kernel using test cases 3 and 4 on NVIDIA Tesla. The results are summarized in Table 77.
7.14.

<table>
<thead>
<tr>
<th></th>
<th>Test case 3</th>
<th>Test case 4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Recompute Reference OpenCL</strong></td>
<td>76</td>
<td>1.48</td>
</tr>
<tr>
<td></td>
<td>94.83</td>
<td>1.66</td>
</tr>
<tr>
<td><strong>Recompute Basic OpenCL</strong></td>
<td>77</td>
<td>1.47</td>
</tr>
<tr>
<td></td>
<td>75</td>
<td>2.09</td>
</tr>
</tbody>
</table>

Table 7.14: The execution time and the speedup of the new kernel run with test cases 3 and 4 on NVIDIA Tesla. Speedup is given according to the reference OpenCL solution.

The more realistic speedup is the one calculated for the test case 4, since it has considerably more antenna positions than the test case 3, but still less than a real case; and the more antenna positions there are, the higher the speedup we expect when they are re-computed. The results of the test case 4 show, indeed, that with the basic optimizations and the new kernel we achieved 2 times speedup compared to the reference OpenCL solution.

Note that the antenna positions are calculated using native arithmetic instructions, which results in lower accuracy. When the non-native arithmetic instructions are used, the speedup is lower than 1 (i.e. the performance is worst, as the repeated recalculations becomes expensive).

When the antenna positions are calculated inside the kernel, the kernel needs to be aware of the path that antenna is traversing. When the large input needs to be divided, we face additional implementation difficulties (thus, we leave that implementation for future work). Consequently, we cannot run the test cases on NVIDIA GeForce and ATI Radeon GPU due to memory limitations. Nevertheless, we believe that the speedup cannot go above the ones given in Table 7.14. These GPUs have higher memory bandwidth than NVIDIA Tesla, meaning that reducing memory traffic has a lower impact on the overall performance.

Let us look at the roofline model for this solution. According to the values given in subsection 7.3.2, each kernel now utilises 57 FLOPs per antenna position iteration and has a memory requirement of 8\(B\) in total. Therefore, the new operational intensity equals:

\[
\frac{|a| \times 57\text{FLOPs}}{|a| \times 8B} = \frac{57}{8} \frac{\text{FLOPs}}{B} = 7.125 \frac{\text{FLOPs}}{B}
\]

This operational intensity is approximately 6 times larger than the one of the reference solution. It is interesting to look at the roofline model presented in Figure 7.4. We see from the Figure that this operational intensity hits the computation-bound on NVIDIA GeForce, while on NVIDIA Tesla it is still memory-bound, although it is much closer to the computation-bound compared to the initial operational intensity.

We also estimated GFLOPS and memory bandwidth utilization. Theoretical results are obtained from code analysis and FLOPS values given in subsection...
Figure 7.4: Roofline model of the NVIDIA Tesla C1060 and NVIDIA GeForce GTX 280. Leftmost vertical line presents application operational intensity of the reference OpenCL solution, while the rightmost vertical line presents operational intensity of the basic OpenCL solution where antenna positions are re-computed.

7.3.2, while empirical results are obtained from the OpenCL profiler. The results are summarized in Table 7.15.

<table>
<thead>
<tr>
<th>Test case 3</th>
<th>Theoretical</th>
<th>Empirical</th>
</tr>
</thead>
<tbody>
<tr>
<td>GFLOPS</td>
<td>263</td>
<td>28%</td>
</tr>
<tr>
<td>MBw [GB/s]</td>
<td>34</td>
<td>34%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test case 4</th>
<th>Theoretical</th>
<th>Empirical</th>
</tr>
</thead>
<tbody>
<tr>
<td>GFLOPS</td>
<td>254</td>
<td>27%</td>
</tr>
<tr>
<td>MBw [GB/s]</td>
<td>33</td>
<td>33%</td>
</tr>
</tbody>
</table>

Table 7.15: The theoretical end empirical results for GFLOPS and MBw, for test cases 3 and 4, running on NVIDIA Tesla. Abbreviations are for the absolute value (Abs.), respective utilization (Util.) and the difference compared to the basic OpenCL solution.

From the table we see several things. On the one hand, the theoretical results show that although the computation utilization is increased, the memory traffic utilization is decreased, and thus we have more balanced ratio between the computation and memory traffic. However, when we compare the safe bounds theoretically calculated with the profiler results, we see that they are obviously too strict and can be relaxed. On the other hand, the empirical results show that the applica-
tion has almost reached the platform memory bandwidth limitation, while there is still a lot more of space for computations, thus meaning that the application is still strictly memory-bound. The roofline model shows that the computation bound is almost reached, which is invalidated by the empirical results. Therefore, in this place, we reached the crossroad where we need to estimate bottlenecks according to three fairly different results: the theoretical estimation, the roofline model, and the profiler results. The profiler results should be taken as a reference and that the application is still memory-bound (although there are a lot of assumptions there that can lead to significant accuracy issues). As this analysis is somewhat beyond the purpose of this thesis, we leave the question "which model is correct" open and its answer as future work.

Finally, in Table 7.16 we make a comparison between different solutions and platforms, using the basic OpenCL solution extended with the new kernel and the test case 4.

<table>
<thead>
<tr>
<th>Sol.</th>
<th>Reference</th>
<th>Proc.</th>
<th>OpenMP</th>
<th>Cell</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Itanium</td>
<td>Itanium</td>
<td>Cell</td>
<td>NVIDIA Tesla</td>
<td></td>
</tr>
<tr>
<td>$T$</td>
<td>115544s</td>
<td>1443s</td>
<td>/</td>
<td>75s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(1.3days)</td>
<td>(4h)</td>
<td>/</td>
<td>(1.25min)</td>
<td></td>
</tr>
<tr>
<td>$S$</td>
<td>1</td>
<td>8</td>
<td>/</td>
<td>1541</td>
<td></td>
</tr>
<tr>
<td>$S_n$</td>
<td>1</td>
<td>1</td>
<td>/</td>
<td>51</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.16: Execution time and speedup of the OpenCL solution where the antenna positions are re-computed during the processing on a GPP (Intel Core i7-920), Cell (IBM Cell), and NVIDIA Tesla C1060 GPU, for test case 4.

As mentioned, the presented optimization is very bodyscan-specific and most often we do not know the exact positions of the antennas. Therefore, we leave out this optimization in the following sections, and look at other possibilities for memory traffic optimizations, which are mainly related to the observations.

### 7.3.4 Optimization Strategies

In general, there are three key issues when looking at the application optimizations for GPUs [62]:

- Maximizing parallelism (maximizing the number of work-items);
- Maximizing memory bandwidth;
- Maximizing instruction throughput.

**Maximizing parallelism.** Since our reference OpenCL solution uses several millions of work-items we argue that it will utilize the maximum degree of parallelism
on modern hardware platforms and hardware platforms of the near future. However, the work-items of the reference OpenCL solution have the smallest granularity, meaning that each work-item (kernel) does the least possible work. The output has millions of samples and since processing each sample is given to one work item, there are millions of work items (in the real case we have 16 000 000 output samples). General NVIDIA guideline for the number of work-items and work-groups is that in order to have a good scalability and to give to platforms enough work-groups per one processing element, there should be more than 100 work-groups per one core. Moreover, the number of work-items per work-group should be larger than 256. Thus, to have optimal utilization of hardware resources on most platforms, we should have at least 25600 work-items per one processing element².

Let us now consider the real case where we have 16000000 work-items. On the one hand, NVIDIA GPUs used for the experiments have 240 processing elements divided over 30 cores. Thus, we have \( \frac{16000000}{30} = 533333 \) work-items per core \((533333 \approx 21)\). On the other hand, the latest NVIDIA Tesla GPU (subsection 3.1.2), has 512 processing elements divided over 32 cores. This means that we have \( \frac{16000000}{32} = 500000 \) work-items per core \((500000 \approx 20)\).

In order to explore how the number of work-items influences execution, we have implemented a second version of the computation kernel, where more outputs are processed per kernel, meaning that we increased work-item granularity. With this increase in the granularity (each work-item processes more outputs at once), the execution time increases. Figure 7.5 shows the results. This demonstrates the recommendation that it is indeed better to have more simpler kernels than less, more complicated kernels. The reason is in the fact that the application is memory bound. When we increase granularity, there are less work-items, and the platform thread management is no longer able to efficiently hide the memory latencies with computation.

**Maximizing memory bandwidth.** First, we distinguish two types of memory traffic: (1) between the host and the global GPU memory, and (2) between the on-chip memories and the processing units.

For our application, the host-to-device memory pathway is not causing a performance bottleneck, as data needs to be copied only once in the global memory, and needs to be written back also only once. Therefore, to identify potential sources of increased data-reuse, we need to analyse in more detail the GPU internal memory traffic.

GPUs have three types of on-chip memory: global memory (at system level), shared memory (at core level), and private memory (at processing element level). The internal memory traffic is essentially generated by the data going back and

²We do not consider the grouping of the work-items into the work-groups since we leave that job to the compiler (implicit work-item distribution model [49]) and we assume the best possible distribution.
forth in between the processing units and these memories. Therefore, there are
two immediate optimization opportunities here: (1) decrease the traffic itself, by
enforcing some data-reuse and removing unnecessary loads/stores, or (2) force the
traffic go faster - i.e., use the right memory for the data sizes and re-use patterns
you choose.

We will revisit both these options shortly.

Maximizing instruction throughput. Instruction throughput is the number of
instructions per clock cycle and maximizing throughput means using smaller num-
ber of instructions to get the job done. We are not interested in these optimizations
for two reasons. First, they require going in more depth with the hardware archi-
tecture, and that is not the aim of this work. Second, our application is memory
bound, which effectively limits the instruction throughput, so we must first focus
on that obstacle.

From the previous subsection we can point out that the main performance bott-
tleneck is the global GPU memory. From this subsection we conclude that the only
way to tackle this problem is to use the GPU memory hierarchy, since the other two
methods, maximizing parallelism and maximizing instruction throughput, cannot
bring major improvement. Therefore, we focus on finding ways to use the local
GPU memory more efficiently. In the next subsection, we analyse data layouts
to identify potential data locality to be exploited to make use of the local GPU
memory.

Figure 7.5: Varying the number of outputs processed per kernel on test case 4.
The platform used is NVIDIA Tesla.
7.3.5 Data Analysis

In the following, we give several figures showing data dependencies. We use test cases 3 and 4 and run them on NVIDIA Tesla.

Each output iterates through all the antenna positions and calculates the contribution of the antenna position observation to the overall radar reflection. Therefore, from each antenna position observation, each output takes only one observation for processing and calculates partial radar reflection. Figure 7.6 shows the partial radar reflections of one output sample. Antenna position indexes, shown on the $x$-axis are calculated by "concatenting" the $H$ and $\phi$ coordinates. If $i$ and $j$ are iterators through the $H$ and $\phi$ coordinates, respectively, then an index of an antenna position observation is determined as $\text{antenna\_position\_index} = i \times |H| + j$, whereas each coordinate is calculated from 0 to the number of output samples used in that dimension.

Figure 7.6: Partial radar reflections for one output sample of the cylinder test case (on the left) and the Eva test case (on the right).

Figure 7.7 shows the radar reflections for a set of outputs. Output samples are represented on the $x$-axis. Each output sample index is calculated by "concatenting" the $x$, $y$, and $z$ coordinates. If $i$, $j$, and $k$ are iterators through the $x$, $y$, and $z$ coordinates, respectively, then an index of the output sample is calculated as $\text{output\_sample\_index} = i \times |Y| \times |Z| + j \times |Z| + k$, whereas each coordinate is calculated from 0 to the number of output samples used in that dimension.

From Figure 7.6, where we look at the smallest granularity of one output sample, we can conclude that the data is dense, meaning that most partial radar reflections are greater than zero. In our case, all partial radar reflections are greater than zero. Therefore, using special fast algorithms that use data sparsity is not an option. In turn, Figure 7.7 gives an overview on the typical values for the radar reflections for all the outputs of the used test cases. Since no outputs that have radar reflection equal to zero, we argue that all the data is indeed dense.

Next, we aim to find the correlation between the observations and the outputs, with the goal to determine some data clustering that can be used to increase locality, and consequently reduce memory traffic. For instance, consider the real test case.
Figure 7.7: Radar reflections for a set of outputs of the cylinder test case (on the left) and the Eva test case (on the right). For the cylinder test case, we use the following: along $x$ and $y$ axis we take 40 samples, and along $z$-axis we take 10 samples, while for the Eva test case we use: along $x$ axis we take 50 samples, along $y$ axis we take 20 samples, and along $z$-axis we take 10 samples.

Each antenna position observation holds 300 observations, whereas we have 16 million of outputs. That means that 300 observations are used by 16 million of outputs in each antenna position. In the ideal case, that would mean that $\frac{16000000}{300} \approx 53000$ outputs use the same observation per every antenna position observation. If we could find a pattern describing the correlation between the outputs and the observations in different antenna positions, we could efficiently implement data re-use. This presents a very tempting optimization, if indeed the application would have a simple correlation between the output samples and the observations.

Note that, in general, since we know the exact geometry of both the output and the antenna positions, we could store antenna position indexes that each output sample accesses. However, that would require too much memory. If we consider the real case, besides the memory for the observations and the output, we would also need: $|\text{out}| \times |a| \times \text{sizeof(integer)} \approx 168$TB of memory. Storing this amount of data would need to be done on slow memories, since this amount of fast memory will not be available in the near future. Therefore, it is still faster to recompute when needed. Let us go back to the analysis of the correlation between the observations and the outputs.

First, we observe how an output iterates through the antenna position observations. From every antenna position observation, each output uses only one observation, whose index is calculated from the distance between the antenna and the observed output sample. Figure 7.8 shows how the antenna position observations are accessed.

From Figure 7.8 we can see the regularity of the access, which is determined by the moving trajectory of the antenna. The horizontal lines present antenna moves in the azimuth direction, while the vertical moves present antenna moves along vertical direction (for each step along the vertical direction, antenna makes a circle in the azimuth direction).
One interesting optimization rises from this observation. We could calculate the distance between the antenna and an output sample only in the first iteration through the antenna position observations. Then, in the following iterations through the antenna position observations we could use a pattern that corresponds to the antenna moves to access the observation. Since available test cases are still not a precise match of a real case and since the resolutions of the output and the input are not set accurately, we leave this for the future work.

Second, we analyse how all the outputs access one antenna position observations. Figures 7.9 and 7.10 show this.

Although each output iterates in a very predictable way, from Figures 7.9 and 7.10 we can see that it is hard to establish a correlation between all the outputs and one antenna position observation due to the fairly irregular access patterns. This is most visible on the lower left graph in the figures 7.9 and 7.10, and the reason behind this is the geometry. On one hand, the output samples coordinates are "still" and observed in the 3D Cartesian coordinate system. On the other hand, antenna moves in the polar coordinate system. In each antenna position observation, the observation that is accessed by all the outputs is determined by the distance from the antenna. In particular, all the outputs that have the same distance to antenna
Figure 7.9: Antenna position indexes accessed in one antenna position observation by all the outputs in the cylinder test case. The figures on the top show 1000000 outputs, while the figures on the bottom show 10000 outputs. The figures on the left show un(sorted) data, while the figures on the right show sorted data.

will access the same observation. Therefore, to make a good use of data locality, we could sort the outputs such that we cluster together the outputs that access the same observation. That would significantly reduce memory traffic since we would need to load only one observation once per output set and not per each output. In the reference OpenCL solution each output loads its observation from the global memory, thus leading that 16 million of outputs load one observation. On the contrary, in the case where we have output sets sorted according to the observation they access, we need to load only 300 observations from the global memory. Take into account that that needs to be done per each antenna position observation.

To see if such an optimization is worth the trouble, we measure how much faster is the application if the observation loading is excluded, and we got a speedup of 2.6 for test case 4 on NVIDIA Tesla.

However, the problem is that in each antenna position, a different set of outputs access the same observation. Therefore, it is of no benefit to sort the outputs unless we do it in each antenna position iteration, which would add a lot of overhead. Even if such an algorithm might lead to performance gain, that is hardly implementable with the OpenCL model. First of all, in each iteration we would need to sort outputs. That means that the output sets that access the same observation should be grouped together in work-items, so that an observation can be stored in the
local/private memory and easily reused by all the outputs in a set. Unfortunately, we cannot assign different output sets to work-items in each iteration on the GPU. This means that we would need to make a new enqueue each time the outputs are sorted. Thus, we argue that sorting outputs per antenna position observation cannot lead to performance improvement due to the additional overhead of sorting itself, increased communication between the GPP and the GPU (in each antenna position, new enqueue is needed), and the additional computation needed to properly store the results of sorted outputs.

Finally, to conclude, although the application has a lot of data reuse, that cannot be efficiently exploited, due to the complex correlation between the observations and the outputs.

### 7.3.6 Data-dependent Optimizations

Let us take a look at the variables stored in the global memory that are needed by every kernel. Every kernel uses three data structures from the global memory: observations, antenna positions, and the output, whereas in each output sample we distinguish output position and radar reflection. In our reference implementation, each kernel reads the following from the global memory: output position, antenna

---

Figure 7.10: Antenna position indexes accessed in one antenna position observation by all the outputs in the Eva test case. The figures on the top show 1000000 outputs, while the figures on the bottom show 10000 outputs. The figures on the left show un-sorted data, while the figures on the right show sorted data.
positions, and antenna position observations. After doing the calculations, the kernel writes the radar reflection back to the global memory.

From the perspective of the global GPU memory we can consider the following most important algorithmic operations:

• Iterating through the outputs - loop that iterates through the output samples;
• Iterating through the antenna positions - loop that iterates through the antenna position observations;
• Read antenna position - read the antenna position;
• Read output position - read the position of the output sample;
• Read observation - read the specific observation from the antenna position observations. An observation is read according to the antenna position and the output position.
• Read output reflection - this step is only needed when we are accumulating output reflections in a particular way. This will be clear in the following solutions;
• Write output reflection - write the result of the calculations.

We describe each following solution using these operations. The top level iteration in the algorithmic description shows how the work-items are distributed (the forall loop), and it is not a part of the kernel implementation itself. Moreover, for each read/write operation we write a prefix indicating the source and the destination memory. We use three letters: g, l, and p, indicating global, local, and private memory, respectively. Let us start with the reference OpenCL solution:

```opencl
forall outputs (i = 1..N)
g.p.read position(output[i]);
for antenna_positions (j = 1..M)
g.p.read position(antenna_position[j]);
g.p.read observation(antenna_position[j], output[i]);
compute partial output;
p.g.write reflection(output[i]);
```

In the reference OpenCL solution, we are focused on the output samples. On the one hand, the output positions are brought from the global memory only once and the final radar reflection is written to the global memory only once, as well. On the other hand, the antenna positions and observations are brought from the global memory in each iteration. Therefore, we look for other solutions that focus on the other two data structures: observations and antenna positions. We analyse the impact of loading observations and antenna positions in the local/private memory and compare the effects with the reference OpenCL solution.

Note that the private memory is very small and only very small chunks of data can be stored in it. Local memory is considerably larger than the private memory,
but it is shared among several processing units, thus meaning that the local memory is accessible to one processing element, equals: 
\[
\text{size of local memory per core} = \frac{\text{number of processing elements per core}}{\text{number of processing elements per core}}
\]
Moreover, we derive formulas that can be used to determine which is the best solution for a particular hardware platform, in terms of memory communication between the global memory and the processing elements. In the calculations of the memory bandwidth, we calculate only global memory communication and neglect the cost of communication between the processing element and the local/private memory. This is reasonable assumption since in modern GPUs the global memory communication is around 100 times slower than the local memory communication.

We make three attempts to use data locality better. First attempt reorders loops, and although it is impractical and even theoretically has worse performance than the reference solution, we present it for the sake of completeness. Second attempt considers the problem from the hardware point of view. However, we realize that the portability of the OpenCL is paid in the loss of the ability to reach platform resources in a manual way. Finally, the third attempt summarizes the previous solutions and proposes a solution where each output loads a chunk of observations that is likely to be re-used.

**Attempt No. 1.** By iterating through the antenna position observations instead of through the outputs we could re-use observations, or antenna positions, or even both by using local/private memory. This is the counter example to the reference OpenCL solution. It interchanges the loops and the emphasis is on the antenna positions instead on the outputs. The simplest example would be to store antenna positions in the private memory.

```c
for all antenna_positions (i = 1..N)
g.p_read position(antenna_position[j]);
for outputs (j = 1..M)
g.p_read position(output[j]);
g.p_read reflection(output[j]);
g.p_read observation(antenna_position[i], output[j]);
compute partial output;
p.g.write reflection(output[j]);
```

In this case we would have:

- **Global memory read and write per kernel:**
  
  \[
  \begin{align*}
  \text{reads} & = 24B + |out| \times 28B \\
  & \approx |out| \times 28B \\
  \text{writes} & = |out| \times 8B
  \end{align*}
  \]

- **Total memory requirements:**
  
  \[
  \begin{align*}
  \text{total} & = |a| \times (\text{reads} + \text{writes}) \\
  & \approx |a| \times |out| \times 36B
  \end{align*}
  \]

89
We can see that compared to the reference OpenCL solution, this solution has \( \frac{36}{32} = 1.125 \) higher memory requirements. Furthermore, it also uses less kernels with higher workload compared to the reference solution. For example, in the real case, we would have 2880000 work-items. According to the guidelines given in the previous subsections, which state that we should have 25600 work items per core, that means that with the new NVIDIA Tesla C2070, we would have \( \frac{2880000}{32} = 90000 < 25600 \) work-items per processing element \( (\frac{90000}{25600} \approx 3.5) \). Finally and most importantly, in this case we need to have synchronization, because every work-item operates on the same set of outputs, which definitively makes this solution unattractive. In order to avoid synchronization, each kernel (work-item) needs to operate on an exclusive set of output samples.

**Attempt No. 2.** Another solution is to store antenna position observations in the local memory, with the idea that the observations are re-used. As we have seen in the previous subsection, the observations are heavily re-used, but from the data layouts we also concluded that it is difficult to make a correlation between the outputs and the observation. Therefore, one solution can be to load whole antenna position observations into the local memory, since then we are certain that the outputs will have access to the appropriate observation. This further means that we need to have the same antenna position observations in all the local memories in order to be certain that outputs allocated for processing on a specific core will have the observations available in the local memory. However, this is not possible, since we do not have the control on the content of the local memory. Before explaining this in more details, let us look at some memory considerations.

Since the whole array of antenna position observations has several gigabytes, we can bring only a very small portion of it to the local memory of a GPU, which has size in order of kilobytes (on both NVIDIA GPUs, local memory has only 16KB). The smallest possible chunk of data that can be brought to the local memory without taking into account the observed output area is one antenna position observation. In a real case, each antenna position observation is 2.5KB large. For test case 3 and 4, antenna position observations are 9.5KB and 25KB, respectively. This means that we must limit ourselves to the test case 3. To deal with test case 4, we need at least 25KB of local memory.

From a software side, a local memory is shared by the work-group. From the hardware side, a local memory is shared by all the processing elements on one core. When looking at software, at OpenCL, we only have the control over the work-items and the work-groups. We can group work-items into work-groups and allocate work-items/work-groups in up to three dimensions. In that sense, we could implement such that each work-group loads an antenna position observation into the local memory, since local memories are used by work-groups. Now, the key point lies in the fact that we do not know how are the work-groups going to be mapped on the hardware. We know that work-groups are allocated per core and cannot migrate on a different core during the execution, but we do not know how
many work-groups are going to be allocated per core, even if we have the number of work-groups equal to the number of cores. If more work-groups are allocated on one core, then, at least in our test cases, we do not have enough of local memory for two work-groups to operate simultaneously, since we can have only one antenna position observation per local memory and having even two work-groups per core

Therefore, with OpenCL we do not have direct access to the hardware resource. This is reasonable, since OpenCL is made for portable solutions and it releafs the programmer of thinking about the hardware.

**Attempt No. 3.** Before introducing our last solution, let us briefly summarize the conclusions brought from the previous attempts, which will naturally lead to the proposed solution.

- The work-distribution should be done according to the outputs, such that each work-item operated exclusively on an output or a set of outputs (Attempt No. 1);
- In OpenCL we cannot make implementations based on the assumptions about the hardware memory utilizations (Attempt No. 2.).

Therefore, the only way to use the local memory is to extend reference (or basic) OpenCL solution such that each output locally stores a chunk of observations where it will most likely hit. Although this solution has even higher memory traffic, we expect that it has shorter execution time since the data is brought to the local memory into bigger chunks. Here we developed several solutions. In the basic solution each output sample (work-item) iterates through the antenna positions in steps. In each step, the distance is calculated and the appropriate antenna position observation array index. According to that index we load a chunk of data that includes observations that are likely to be hit in the antenna positions of the step size and we iterate through them. The algorithm of the solutions is presented in the following.

```c
for all outputs (i = 1..N)
    g_p_read_position(output[i]);
for antenna_positions/step (j = 1..M)
    g_p_read_position(antenna_position[j]);
    g_l_read_observation_chunk(antenna_position[j], output[i]);
    for step (k = 1..P)
        compute partial output;
    p_g_write_reflection(output[i]);
```

This solution gave a 1.4 speedup compared to the reference OpenCL solution. However, since we iterate through the observations brought in the local memory without taking into account the geometry, the results are not correct. Moreover, we made an attempt to implement a correct solution by loading observations from the global memory if the observation is not present in the local memory, but we did not gain any performance improvement in that case and the results were still not correct. Since we cannot easily look at the cause of the incorrect results due
to a very low observability of the solutions running on the GPU, we leave further analysis of these solutions for future work.

7.4 Summary of the Parallel Solutions

In this section we summarize different parallel solutions. First we compare different OpenCL solutions, which are central to the thesis, and then we compare all the parallel solutions to the reference solution. Parallel solutions used in the comparisons are: basic OpenCL solution, best OpenMP solution, and Cell solution.

Tables 7.17 and 7.18 compare all the OpenCL solutions, for different platforms and test case 3 and 4, respectively. The reference solution is executed on Intel Itanium. The three OpenCL solutions used in comparisons are: the reference OpenCL solution (REF), the basic OpenCL solutions (BASIC) and the basic OpenCL solution where antenna positions are re-calculated (ANT). The hardware platforms in the table are NVIDIA Tesla C1060, NVIDIA GeForce GTX 280, ATI Radeon 5870, and Intel Itanium i7-920. Beside the standard metric abbreviations, we also use the following: $S_{\text{REF}}$ - speedup comparison with the reference solution; $S_{\text{CL}}$ - speedup comparison with the reference OpenCL solutions. Moreover, for the computation and the memory throughput (GFLOPS and MBw), both the absolute value and the utilization are presented (utilization is presented in the parentheses).

<table>
<thead>
<tr>
<th>plat.</th>
<th>sol.</th>
<th>$T$[s]</th>
<th>$S_{\text{REF}}$</th>
<th>$S_{\text{CL}}$</th>
<th>GFLOPS</th>
<th>MBw[GB/s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tesla</td>
<td>REF</td>
<td>153</td>
<td>752.38</td>
<td>1</td>
<td>/</td>
<td>67 (65%)</td>
</tr>
<tr>
<td></td>
<td>BASIC</td>
<td>103</td>
<td>1117.61</td>
<td>1.49</td>
<td>83 (9%)</td>
<td>67 (65%)</td>
</tr>
<tr>
<td></td>
<td>ANT</td>
<td>77</td>
<td>1494.99</td>
<td>2.09</td>
<td>263 (28%)</td>
<td>37 (36%)</td>
</tr>
<tr>
<td></td>
<td>REF</td>
<td>160</td>
<td>719.46</td>
<td>1</td>
<td>/</td>
<td>64 (45%)</td>
</tr>
<tr>
<td></td>
<td>BASIC</td>
<td>75</td>
<td>1534.86</td>
<td>2.13</td>
<td>80 (9%)</td>
<td>64 (45%)</td>
</tr>
<tr>
<td></td>
<td>ANT</td>
<td>/</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GeForce</td>
<td>REF</td>
<td>201</td>
<td>572.71</td>
<td>1</td>
<td>/</td>
<td>51 (33%)</td>
</tr>
<tr>
<td></td>
<td>BASIC</td>
<td>57</td>
<td>2019.55</td>
<td>3.52</td>
<td>64 (12%)</td>
<td>51 (33%)</td>
</tr>
<tr>
<td></td>
<td>ANT</td>
<td>/</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Radeon</td>
<td>REF</td>
<td>/</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BASIC</td>
<td>5661</td>
<td>20.33</td>
<td></td>
<td>2.25 (5%)</td>
<td>1.8 (14%)</td>
</tr>
<tr>
<td></td>
<td>ANT</td>
<td>/</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7.17: Comparison of different OpenCL solutions on various platforms for the test case 3.

Tables 7.19 and 7.20 compare the selected parallel solutions to the reference solution, for test case 3 and 4, respectively. The parallel solutions used in the comparison are: best OpenMP solution (OpenMP) and basic OpenCL solution (OpenCL). The hardware platforms in the table are Intel Itanium i7-920, NVIDIA Tesla C1060, and ATI Radeon 5870. The speedup ($S$) is given according to the
Table 7.18: Comparison of different OpenCL solutions on various platforms for the test case 4.

<table>
<thead>
<tr>
<th>plat.</th>
<th>sol.</th>
<th>$T$ [s]</th>
<th>$S_{\text{REF}}$</th>
<th>$S_{\text{CL}}$</th>
<th>GFLOPS</th>
<th>MBw [GB/s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tesla</td>
<td>REF</td>
<td>156</td>
<td>740.67</td>
<td>1</td>
<td>/</td>
<td>62 (61%)</td>
</tr>
<tr>
<td></td>
<td>BASIC</td>
<td>125</td>
<td>924.36</td>
<td>1.25</td>
<td>77 (8%)</td>
<td>62 (61%)</td>
</tr>
<tr>
<td></td>
<td>ANT</td>
<td>77</td>
<td>1500.58</td>
<td>2.09</td>
<td>263 (28%)</td>
<td>37 (36%)</td>
</tr>
<tr>
<td>GeForce</td>
<td>REF</td>
<td>160</td>
<td>722.15</td>
<td>1</td>
<td>/</td>
<td>61 (43%)</td>
</tr>
<tr>
<td></td>
<td>BASIC</td>
<td>84</td>
<td>1375</td>
<td>1.90</td>
<td>76 (8%)</td>
<td>61 (43%)</td>
</tr>
<tr>
<td></td>
<td>ANT</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>Radeon</td>
<td>REF</td>
<td>216</td>
<td>534.93</td>
<td>1</td>
<td>/</td>
<td>45 (29%)</td>
</tr>
<tr>
<td></td>
<td>BASIC</td>
<td>65</td>
<td>1777.61</td>
<td>3.35</td>
<td>56 (36%)</td>
<td>45 (29%)</td>
</tr>
<tr>
<td></td>
<td>ANT</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>Itanium</td>
<td>REF</td>
<td>6675</td>
<td>17.31</td>
<td>/</td>
<td>1.91 (4%)</td>
<td>1.5 (12%)</td>
</tr>
<tr>
<td></td>
<td>BASIC</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
</tbody>
</table>

Table 7.19: Comparison of different parallel solutions for the test case 3.

Table 7.20: Comparison of different parallel solutions for the test case 4.

7.5 A Qualitative Comparison

To summarize different solutions we will compare them between each other against four quality metrics: programmability, performance, portability, and productivity.

Programmability indicates how easy it is to express the algorithm in terms of a specific language. It includes objective estimates as code size and the available
support of language features, as well as some more subjective estimates, such as the ease of writing correct programs, ease of language learning, and code maintainability. Since all of the solutions are written using the C (with extension), we will consider the ease of language learning as the effort needed by the C programmer to learn it.

*Performance* is an objective metric, estimated according to the solution execution time, and its scalability.

*Portability* shows how efficiently can a solution be ported over different platforms, while being able to maximally utilize hardware resources.

*Productivity* is a subjective measure, that summarizes the other three qualities and it expresses how easy it is, in terms of time, to develop a solution that is portable and efficient.

Let us now compare the previously described solutions.

The reference solution (section 6.2) presents the simplest C sequential implementation of the bodyscan SAR processing algorithm. It has the highest programmability: it took us the shortest time to develop the solution and it has the smallest code size. Of course, this is paid in the lowest performance: the longest execution time and no scalability. Essentially, there is no portability since the solution always uses only one thread of the execution.

The best OpenMP solution gave considerable performance gain with minimal additional programming efforts. With only two additional lines of code we obtained almost perfect scalability and therefore an impressive reduction in execution time. However, when we needed to introduce synchronization between threads in the OpenMP solution 2, programming efforts were increased. Therefore, although the first parallel solution gave almost perfect speedup, due to an ideal parallelization case where no synchronization is needed, when the synchronization was introduced in the OpenMP solution 2, the scalability had a decreasing trend as the number of threads was increased. This means that in addition to the fact that portability is constrained to the general-purpose processors, the scalability is in general also limited, depending on the synchronization.

Finally, the OpenCL solutions gave outstanding performance, the highest portability, and the lowest programmability. Execution time is hundreds of times lower than both the execution time of the reference solution and the OpenMP solutions, while portability is limited to the devices that support OpenCL. However, since OpenCL is made to be generic and implementable by almost any platform, the OpenCL solution can be ported to various hardware platforms. Programmability is the lowest, since besides writing kernels for which we use the basic C, we need to use a large API to appropriately initialize the OpenCL platform and devices, which includes extra programming effort. Moreover, the maintainability of the solution is low, since it is very hard to test and debug the solutions.

If we look at the productivity as the amount of time needed for developing a portable and efficient solution, then the OpenCL is the absolute winner, since the higher programming effort of a few additional weeks heavily pays off in terms of performance and portability. A brief summary of the solution comparison against
the four metrics is summarized in the table 7.21.

<table>
<thead>
<tr>
<th></th>
<th>Programmability</th>
<th>Performance</th>
<th>Portability</th>
<th>Productivity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference</td>
<td>+</td>
<td></td>
<td>+/-</td>
<td></td>
</tr>
<tr>
<td>OpenMP</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>OpenCL</td>
<td></td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

Table 7.21: The reference solution has a high programmability but unacceptably low performance and portability, and therefore also productivity. The best OpenMP solution, which is considered here, gave a great performance gain compared to the reference solution with negligible additional effort concerning programmability. Portability is good, but limited to general-purpose processors. Due to still very low performance, the productivity is marked as low. Finally, the OpenCL solution have outstanding performance and portability, with considerable increase in productivity, which leads to very high productivity.
Chapter 8

A Strategy for OpenCL Application Design and Implementation

In this chapter we propose an empirical, step-by-step strategy for effective implementation of parallel applications in the OpenCL language/model. We discuss each step in detail, and show how it applies to our case-study application. We strongly believe that following such a systematic strategy for implementing an application in OpenCL is highly beneficial for most applications, resulting in increasing both the portability and the performance.

8.1 Strategy

In this section we propose the strategy.

Setting the Baselines

Step 1: Application Specification

First, the application has to be properly specified: its functionality, input and output data sets, and performance requirements have to be analysed, well understood, and clearly specified. Note that there is no need yet to dive into possibly available source code: we consider the functionality, the data characteristics, and the performance bounds of an application available in the ”problem domain”, i.e., they are requirements from the application side, not artificial constraints from the implementation or hardware side.

Functionality analysis investigates what the application is doing and how. Consequently, the functionality specification should reflect the application structure and computation requirements. Further, the first functional and data dependencies are already spotted here, hinting towards potential parallelization options.
**Data requirements analysis** focuses on finding out what are the main data structures, types, and sizes for both the input and the output data, what is the distribution of the data (is it sparse or dense, one- or multi-dimensional), and what are the requirements for temporary data (in terms of memory-footprint). Note that we are not proposing a detailed analysis to give perfectly accurate values for these parameters, but rather for coarse estimates and ratios which will give insight into memory fitting, allocation, and re-use potential. Thus, *data requirements specification* includes a list of the most important data structures in the application, together with information on their type, the number of dimensions and size(s), and any available distribution information.

**Performance requirements** are usually provided by the application nature or by the user as bounds on the execution time or on platform utilization. For instance, it is reasonable that scanning a human body at the airport checkpoint and processing the scanned data should be no longer than a few seconds. We believe that performance requirements specification for a new application is essential in setting a pragmatic goal for the application and providing a clear stop condition in the iterative process of code implementation, tuning, and optimization.

**Step 2: Design or Choose the Algorithms**

Once the application specification is ready, there are several potential algorithms to solve the problem. In most cases, these algorithms are defined in the problem field (like it is the case of the body-scan application), but they might as well be designed by the programmers themselves. Ideally, in this step, we select multiple algorithms, more promising in terms of meeting our performance goals, and work with them as alternatives. However, in practice, due to multiple development time constraints, we just choose one algorithm which we will try to tune until it fits in the set performance numbers.

**Step 3: Develop the Sequential Reference Solution**

For each algorithm that seems promising, one reference solution without any optimizations should be made available. This reference solution has to be implemented as simply as possible, in order to properly reflect the essence of the algorithm. Note that, for the sake of time, one can potentially skip this step if there is already a known parallel solution for the problem at hand. In many cases, though, parallel solutions inherited from larger parallel systems do not work "out-of-the-box", so having a simple, quick-and-dirty sequential reference implementation helps.

**Step 4: Design the Parallel Solutions**

In most cases, there are several parallel versions to be designed for any reference solution. Although some parallelisations might not lead to improvements at the first glance, they could be effective on some hardware platforms, or for special
data layouts. Thus, we prefer to expand the collection of possible parallelizations in all possible directions, filtering out the worse cases in a later stage. Note that, at this point, a "parallelization"/"parallel solution" is in fact a parallelized version of the reference algorithm/implementation, and it should be as platform-agnostic as possible.

The extended set of parallel solutions is further filtered to only a subset of "high-potential" solutions, which are ranked in terms of potential performance and portability. The most promising ones (usually, not more than 2 or 3) are chosen for implementation.

Step 5: Implement and Benchmark the First Parallel Solutions

The first parallel solutions are implemented using a generic hardware platform (typically, a simple multi-core GPP) that provides flexibility, observability, and enough tools for efficient development and analysis.

After implementing the selected solutions on a generic platform, they have to be benchmarked to determine the computation intensity and the memory intensity of the application. In this way we get to know what are the major application bottlenecks.

The typical metrics that should be used for analysis and comparisons are throughput (GFLOPS), memory traffic (GB/s), and speedup compared to the reference solutions. Other useful insights are also given by utilization, where we compare the achieved application GFLOPS and GB/s on a specific platform to the platform peak performance. By checking the utilization numbers, we conclude whether the application is computation-bound or memory-bound.

The Portable OpenCL

Step 6: Design, Implement, and Analyse a Portable Solution in OpenCL

In this step, we design the first OpenCL solution. The emphasis in this step is on portability, since a target platform has not been chosen yet. Our experience on developing portable OpenCL solution suggests the following:

- Rather use a lot of simpler work-items than only few complex work-items;
- Keep kernels as simple as possible, since the observability is usually low;
- Use very simple data structures;

Therefore, from the best performing parallel applications implemented in Step 5, we select the one with the smallest task granularity, and proceed to implement it. Typically, each OpenCL work-item will correspond to the smallest independent work unit that has been found in the original parallel implementation.

The first performance, scalability, and portability studies can already be performed here. In most cases, the first utilization numbers to be seen are below 10% for computation and bandwidth utilization.
Step 7: General Optimizations

Before going into platform-specific optimizations, two types of basic, generic optimizations have to be considered:

- **OpenCL optimizations:** if the loss of accuracy is acceptable, it is advisable to use the native arithmetic instructions and OpenCL compiler flags for relaxed math;

- **Data-dependent optimizations:** inspecting the data layouts can lead to ways of storing the data in a convenient fashion for easy access and/or improved locality. For finding access patterns, we propose a simple trace analysis, where the application is executed and all its data accesses are recorded and plotted. If input data is accessed in any regular manner, this can and should be used to increase data re-use.

When this optimizations are added, we have a portable OpenCL solution with a first set of basic optimizations available.

Step 8: Performance Check-point

This step is dedicated to performance analysis and estimates, investigating how far off the current implementation is from the required performance. In most cases, a performance boost is still needed, and it can only be gained by applying lower-level optimizations, which are platform-specific. This is the point where the OpenCL trade-off between performance and portability becomes visible.

We estimate the performance of a new implementation by benchmarking and/or profiling. The metrics to be used are: GFLOPS, memory traffic [GB/s], and speedup compared to the reference solution. Note that benchmarking has to be done with representative data, i.e., data that matches the properties identified in the application specification phase.

Further, at this point, we also investigate the main performance bottlenecks, i.e., the compute units or the memory. One way to systematically investigate this is building the roofline model, an easy to derive model that gives a good, graphical overview on the application bounds on a specific platform. Thus, by building a roofline model for each platform that the application might target, we determine on which side of the performance spectrum we are: compute-bound or memory-bound. Note that we indeed need one roofline model per platform, as the same application can be compute-bound on one platform, and memory-bound on another. We also use the roofline approach because, once we built a model, it can be easily updated when new optimizations are added on the platform.

If the current solution satisfies the performance requirements on any of the potential hardware platforms, the development can and (in most cases) should stop. However, it is rarely the case, and going one step further to look into hardware-specific optimizations is usually mandatory. Once on this path, however, the porta-
bility of the OpenCL implementation diminishes significantly - not in terms of code/language, but in terms of parallelization and structure.

**Iterative Optimizations and Tuning**

**Step 9: Platform-specific Optimizations**

This step should be iteratively performed while the following conditions are satisfied: there are still optimizations available, the time budget allows optimizations, and the performance goals are not yet reached. If any of these conditions is no longer satisfied during the iterations, the process should jump to the next step. Note that in this step, we already consider one single platform as the target of our application.

To be sure we apply the various optimizations in the most effective way, we first check if the application is compute-bound or memory-bound, using the roofline model already built.

- **Compute-bound applications** are typically optimized by reducing the number of operations. In most cases, this means re-using some results instead of re-calculating, or reordering instructions in a more compiler-friendly way. Specifically, application-wise modifications include pre-calculating and/or re-using results by storing and loading them from memory, which is in fact a way to trade calculations for the memory traffic. Alternatively, most platforms offer different types of low-level optimizations that speedup the available operations by fusing them (e.g. multiply-add) and/or group them with SIMD-like techniques.

- **Memory-bound applications** can be optimized by reducing or speeding-up the memory traffic. Application-wise, we can trade memory communications for computation, if there are computation parts that are pre-calculated: instead of loading data from the memory, one can simply re-calculate it. Platform-wise, memory-optimizations are very diverse. For example, GPUs have a complex memory hierarchy. The lower the memory is in the hierarchy, the faster and the smaller it is. Using this hierarchy properly may require quite some trickery, but it can reduce the memory access times by up to two orders of magnitude. Alternatively, one can try to increase data locality by properly reordering the data. Finally, there are platform-dependent optimisation, like data coalescing and the use of very specific cached memories, usually available in GPUs.

Note that in each step, one single optimization is applied, followed by an update of the roofline model and the performance check. As soon as any of the loop stop conditions is met, the iterations stop. If the performance requirements have been met, the algorithm stops and we have one implementation up and running. If the iterations stopped due to either development time expiring or optimizations complete, we jump to the final step.
Roll-back

Step 10: Feedback Loops

In the case the required performance is not met in the previous steps, the process can be restarted from various checkpoints along the design path. Therefore, we consider the following options:

- Replace platform: one can also investigate another target platform (see the platform-specific optimizations step), having the advantage that there is no need to start the process all over again.

- Change algorithm: one can get back at the pool of potential parallel or even sequential algorithms to choose another path.

- Relax performance requirements: it might be the case that the current performance requirements are too strict. Relaxing these requirements is equivalent with a "best-effort" approach in the previous step, where the expiring of the development time and/or the optimization completion can deliver the best possible performance for the application running on each potential target; the best performance of all platforms can be taken, if needed.

- Constrain data: if the performance penalties are due to relaxed assumptions on the data set (i.e., data is considered random and dense, although it might not be) or due to out-lier data cases, a more constraint data set might allow better data-dependent optimizations, which in turn might increase the overall application performance.

8.2 Summary

We note here three non-trivial differences between designing and implementing an application using OpenCL (with portability in mind) versus an implementation using a platform-specific environment:

1. The OpenCL parallelization should be as generic as possible, preferably at the lower end of the granularity spectrum. In other words, the work-items assigned by the data distribution should be as homogeneous and as simple as possible. This will obviously favour many-core architectures like GPUs, but also any aggregation for the benefit of more complex cores is easier than further decomposition.

2. Application-specific optimizations - i.e., results re-calculation, data-dependent optimization, reordering data structures, etc. - are applied first. Portability is still preserved.

3. Platform-specific optimizations are applied last, when tying the implementation on a specific platform. Often, this step sacrifices portability.
8.3 Bodyscan Case-study

In this section we show how our step-by-step strategy is applied to the bodyscan application case-study.

Setting the Baselines

Step 1: Application Specification

Functionality analysis This analysis was already performed in the previous work and the basic algorithm was provided.

Data requirements analysis In this analysis, which is mainly done in Chapter 5, the main application parameters were identified and the data requirements were estimated. Furthermore, a representative set of test cases was selected, with both smaller and larger data sets.

Performance requirements The idea is to have real-time processing. Therefore, the processing time should last in order of seconds. However, in this stage of the project the main emphasis was not on performance, but on portability, since the specific hardware platform has not yet been chosen.

Step 2: Design or Choose the Algorithms

The 3D bodyscan SAR processing algorithm comes from the problem domain and was settled in the previous phases of the project.

Step 3: Develop the Sequential Reference Solution

The sequential solution follows directly from the algorithm and problem description, and was already implemented. However, the solution went through several optimization steps, and by reverse engineering we removed all the optimizations and produced the sequential reference solution. We benchmarked the reference solution on a single-core processor to get a rough estimation on its performance and bottlenecks.

Step 4: Design the Parallel Solutions

Based on the application analysis and the reference solution, we extracted three possible parallelizations of the algorithm. We have eliminated one due to high synchronization overhead, and we kept two more: one based on the input data distribution, and the other based on the output data distribution.

Step 5: Implement and Benchmark the First Parallel Solutions

In order to select the representative parallel solution, we selected a GPP for performing our first benchmarks. In addition, we selected OpenMP, as a simple, pragma-based language, for implementing the solutions. After somewhat extensive
benchmarking, we discarded the input-partitioning solution due to lower concurrency, and higher synchronization overhead. We selected output-partitioning as the more promising approach. In particular, the measured FLOPS, memory traffic, and speedup compared to the reference solution (the key metrics for evaluating both solutions) also showed that the output data-partitioning leads to the best performance and scalability. Finally, we determined that the application is memory-bound on the GPP.

The Portable OpenCL

Step 6: Design, Implement, and Analyse a Portable Solution in OpenCL

We developed a first OpenCL solution - the solution with the smallest work-item granularity. In addition, we selected a set of different hardware platforms to estimate the portability, including GPPs and GPUs. The solution gave considerable improvement on all target platforms compared to both the reference solution and the test solutions we developed for step 5. We also performed a series of analysis in order to set the base performance of the portable solution. In particular, we did a theoretical analysis, empirical analysis, and we derived a roofline model. In general, in this step, a detailed, time-consuming performance analysis is not necessary; a simple indication of the application performance will suffice.

Step 7: General Optimizations

Here, we did two optimizations: we used the native arithmetic instructions (less accurate, but faster) and we simplified some of the data structures (to provide better alignment). Neither of the two optimizations influences portability, and they lead to a significant improvement on the GPU platforms used for testing (more than 60%).

Step 8: Performance Check-point

We repeated the analysis done in the step 6, and did theoretical analysis, empirical analysis, and we drew the roofline model for the hardware platforms of interest. Again, the results pointed to the memory traffic as the bottleneck. As we are working on an "best effort" performance model, we continued exploring additional optimizations. Therefore, we had to specify our target platform. We settled for two NVIDIA GPUs: NVIDIA Tesla C1060 and NVIDIA GeForce GTX 280.

Iterative Optimizations and Tuning

Step 9: Platform-specific Optimizations

From the previous analysis, we concluded that the application is memory-bound. Therefore, we focused on reducing memory traffic. Application-wise, we replaced
a part of the algorithm with re-computations instead of the memory communication. Next, we explored data-dependencies in order to efficiently use GPU memory hierarchy. All these optimizations increased the speedup between 40% and 90%.

Roll-back

Step 10: Feedback Loops

This step is provided in our strategy as the back-up plan. In our case, as the performance requirements were not strict, we did not need to pass through this step. Nevertheless, based on the achieved application performance, we believe that the current implementations running in parallel on multiple interconnected GPUs, can achieve real-time execution.

We believe that the proposed strategy is not trivial to apply on a new application, as seen in the bodyscan case-study. We acknowledge that some of its steps might be seen as redundant. For example, implementing a sequential solution for an application that has to be parallelized. One of our future research directions is to study ways to replace these "test" implementations with simpler prototypes or even high-level models. However, we strongly believe that such a systematic approach towards implementing parallel applications using OpenCL will only simplify the design tasks, allowing programmers to focus more effectively on the various types of optimizations, their applicability, and their ordering. Finally, despite seeming like a more time consuming approach than a simple hands-on approach, our strategy allows elegant roll back solutions, that allow quick recovery from failed design/implementation paths.
Chapter 9
Conclusions and Future Work

Multicore systems have become an indispensable part of our everyday life. All processor families, ranging from GPPs, over GPUs, to DSPs, are becoming multicores, with a few to many cores. From the analysis of the current state of the art developments in both multicore hardware and software, we observe that the number of cores is increasing and that multicore architectures are becoming more heterogeneous. For instance, GPPs are becoming equipped with more special-purpose hardware, while GPUs add general-purpose computing capabilities. Software needs to bridge the increasing gap between the various multicore hardware platforms on one side, and the demand for application development, on the other side. Terms like GPGPU and many-core computing have started as HPC synonyms, but they have now become ubiquitous. It is, for sure, the best time for parallel computing to raise up to the expectations and provide more scalable and more portable solutions for effective multicore parallel application implementation.

In this thesis, we have investigated the potential of multicore platforms to boost the performance of a data-intensive application, namely a bodyscan application based on 3D close-range SAR processing. To do so, we have first analysed several hardware platforms, which are likely to be found in the near future hardware trends, and have the theoretical potential to increase the application performance by orders of magnitude when compared with the sequential version. We have also analysed the software tools that accompany these platforms, aiming to ease the programmers’ task of implementing new applications on multicore hardware. Both analyses have revealed a large variety of platforms and software tools, which in turn reflected on an even larger number of potential combinations. As the decision on the target hardware platform is still open, we have opted to implement a portable solution, based on OpenCL, and investigate its performance on multiple multicores.

We started with a thorough application analysis. We revealed a high degree of potential parallelism (a lot of fine-grain concurrency), somewhat obstructed by a lot of apparent random/irregular accesses in very large data structures, a typical example of a data-intensive application. To study the application behaviour with-
out any platform-specific or language-specific overhead, we devised a sequential,
reference solution using C, and a set of generic parallel solutions using OpenMP
on a generic GPP.

From the set of the generic parallel solutions, we picked and analysed the most
promising one, we ported it to OpenCL, and tested successfully on various hard-
ware platforms. Our first goal was to achieve portability, after which we focused
on performance. Thus, we performed a set of basic, hardware-independent opti-
mizations which included algorithmic changes, native arithmetic instructions, and
data-dependent optimizations. Portability was preserved and performance did im-
prove. This OpenCL solution already gave an outstanding speedup compared to
the previous solutions implemented on GPP and Cell.

Nevertheless, in order to explore the trade-offs between portability and p erfor-
mance, we went one step further. We settled for the NVIDIA GPUs and studied in
detail their hardware architecture. In particular, we looked at the core and memory
organization, on the hardware side, and data layouts, on the application side. We
made several attempts to use the provided hardware-specific resources and data lo-
calities. Some of the attempts showed that the portable nature of the OpenCL does
not give a complete control over the application, while others have failed due to
application-specific issues (large data sets reordering, reshuffling, and/or replicat-
ing).

Overall, we have succeeded to implement 3 different OpenCL variants for the
given application, with speedups (versus the reference implementation) from 752
for the basic parallel version, to 924 for the optimized one, and to 1541 for the
version with increased arithmetic intensity (due to an algorithmic optimization).
For the portable solution, the maximum speedup obtained on the NVIDIA GPUs
(namely, NVIDIA Tesla C1060) was 1541, while the GPP (Intel Itanium i7-920)and
the ATi GPU (ATi Radeon HD 5870) achieved speedups of 20 and 2019, respec-
tively. Furthermore, for the NVIDIA GPUs, we have investigated several scenarios
for using advanced memory alignment and coalescing, which lead to a maximum
speedup of 2100, when assuming perfect coalescing. We estimate that the use of
shared memory, another optimization in progress, will lead to an extra performance
boost of 40%-70%. However, these last solutions could not be fully validated on
the real test cases, because the complex reshuffling method for proper aligning of
the input data set is not yet finalized.

Finally, we summarized our experiences into a set of guidelines for develop-
ing applications using OpenCL. In particular, we sketched a generic strategy for
efficient design and implementation of both portable and performant OpenCL so-
lutions.

Our main conclusions are twofold. First, we conclude that OpenCL is a promis-
ing standard (and language) for enforcing the implementation of portable multi-
core applications. The system is fairly robust, and it allows the programmer to
address both general portability and hardware-specific performance. Second, we
conclude that the bodyscan application is a good fit for running on multicore platforms, due to its inherent parallelism. Furthermore, the results we have obtained recommend the GPUs as the target platform for such an application.

9.1 Research Questions Revisited

In this section we revisit the research questions listed in the introduction and show how our work, described in this thesis, answered them.

1. What are the current multicore systems trends, in both hardware and software?
   From the overview of the current multicore developments, we conclude about several important trends. First, most of the modern processor technologies are based on multicores, including GPPs, GPUs, and DSPs. The number of cores is increasing while the clock frequency is stagnating. Second, the hardware platforms are becoming more heterogeneous: GPPs are adding more special-purpose components, while GPUs are moving towards general-purpose programming. Third, in order to bridge the gap between various hardware platforms and the large collection of different applications to be ported, an increasingly important issues in modern multicore software is portability.

2. What is the structure of the application? How does the main algorithm look like? What is its sequential (baseline) performance?
   The application is inherently parallel with large data sets and apparent irregular accesses—clear obstacles for efficient parallel solutions. The reference solution is implemented in C and run on a single-core GPP. The results showed very poor performance and utilization. Further, the analysis pointed to the memory traffic as a bottleneck—expected for such a data-intensive application.

3. How can the algorithm be parallelized on a homogeneous, shared-memory multicore? What is the most suitable parallel algorithm for this problem, in terms of performance?
   We developed several parallel solutions in OpenMP and tested them on a multicore GPP. Analysis and benchmarking pointed to a solution based on output-data partitioning as the most promising one. Again, although we achieved a linear speedup compared to the reference solution, the application was memory-bound.

4. How can the parallel algorithm be transformed to suit graphics-processing units (GPUs)? Can the solution be portable, both in terms of design and implementation?
   We selected GPU as a suitable platform for the high level concurrency that
the application inherently has. Furthermore, since portability is the first concern, we chose OpenCL, the novel software approach that targets different target platforms including multicore GPPs, GPUs, and others.

5. Is OpenCL truly portable, both in terms of code and performance?
Several of our OpenCL solutions truly showed portability: although programmed and tested on NVIDIA GPUs, they were able to run almost "out-of-the-box" (less than 1% modifications) on ATI GPU and GPPs. Further, the obtained speedup on all these platforms was very good, which proves that OpenCL provides a good trade-off between portability and performance.

6. Are there any generic optimizations? What is the impact of platform-specific optimizations on the application performance when running on a particular family of processors? What is the impact of data-dependent optimizations on the performance of the OpenCL solution?
After achieving portability, we shifted our focus to performance. We looked at generic optimizations (trying not to influence portability), hardware-dependent optimizations, and data-dependent optimizations. The tested generic optimizations brought considerable speedup. The hardware-dependent and data-dependent optimizations were much more difficult to implement (partially due to the application, partially due to the OpenCL nature), but in the end we have seen some promising results.

7. Can we derive a strategy for developing OpenCL solutions? How shall we approach portability and performance in OpenCL?
We derived a strategy, by showing a step-by-step algorithm that describes how to develop a parallel OpenCL solution of any application. First, the top level application is analysed and the reference and generic parallel solutions are developed. Then, we focus on portability, by developing a generic OpenCL solution. Finally, we select a specific platform and dive into platform-specific optimizations, thus trading portability for additional performance.

9.2 Future Work

We broadly define two potential direction for future work: generic research and application-specific/technical research.

Generic research. We developed a strategy for a systematic development of OpenCL solutions. In our immediate future work, we would like to validate and refine this strategy with additional case-studies.

Next, we would like to examine closely the performance measuring tools for multicores - both the theoretical and the empirical ones. For instance, in our analysis, we used three methods. First, the classical approach, where the statically calculated computation intensity and memory traffic is compared to the platform
characteristics. Second, we used profiler tools provided by the vendors. Finally, we built and used Roofline models. Still, the classical and the Roofline models are not very accurate to build and apply, and still very difficult to validate; profilers are more accurate, but, besides being platform specific, they are also taking a lot of shortcuts to present average numbers for various hardware parameters. Ideally, we would want to replace all these coarse estimates with a model-based approach, which would combine a machine model with the OpenCL implementation model, being able to predict the performance to be achieved. All in one, we would like to explore performance modelling for OpenCL applications.

Finally, on longer term, we would like to devise a modelling technique that allows applications to be specified and modelled such that the reference sequential and parallel solutions no longer need to be fully implemented in C-like or OpenMP-like languages just for analysis purposes. Instead, we envision this analysis done at the design level, using the model, expecting that such an approach will simplify the application development process.

**Application-specific/Technical Research.** This research path mainly focuses on adding various types of optimizations to the current framework. First of all, our main focus are NVIDIA GPUs. Their memory hierarchy has a lot of opportunities for optimizations and, if properly used can bring outstanding application speedup. We only dealt with this memory hierarchy to a certain extent, because the application data layouts made our job particularly difficult. In our future work, we would like to analyse these issues in more detail. In particular, we would take a closer look to coalescing, conflicting, and texture memories.

Second, the ATI GPU gave even better performance compared to the NVIDIA GPUs. In our future work, we would like to examine the ATI architectures more closely and to explore the hardware-specific optimizations applicable for those platforms.

Third, we showed that OpenCL on GPP performs 2 times faster than OpenMP on the same GPP, with the same application parallelisation. On GPPs, we could look into more details on caches and SSE instructions.

Finally, we could go away from the hardware and focus on the application itself. In particular we would like to focus on data-dependent optimization. If the data is sorted in a proper manner, the algorithm can be significantly simplified and both the computation and the memory traffic considerably reduced.

To conclude, we analysed many important aspects of OpenCL and GPGPU programming. We analysed, designed, implemented and tested various parallel solutions for our bodyscan case study. We derived a strategy for systematic implementation of portable OpenCL solutions. However, since both OpenCL and GPGPU programming present a multitude of novel techniques in dealing with multicores systems, there is still a wide horizon of possibilities to be explored.
Bibliography

Appendix A

Application Computational Intensity

The following expression gives precise application requirements in terms of the number of required floating-point operations (FLOPs), which determines the application computational intensity. It is derived for the reference solution, but it can easily be adjusted for any other. Although this expression is somewhat cumbersome, it is statically calculated and determines the theoretical upper bound of FLOPs.

\[
(FLOPs \text{ overall}) = (FLOPs \text{ preprocessing part}) + (FLOPs \text{ main processing part})
\]

\[
FLOP = FLOP_{pre} + FLOP_{main}
\]

\[
FLOP_{pre} = (\text{Preprocessing phase - function that reads antenna positions.}) +
(\text{Preprocessing phase - function that initializes the output.})
\]

\[
= |H| \times |\phi| \times
\]

\[
[5 \times Fadd + 8 \times Fmul + 4 \times Fdiv + 2 \times \cos(F) + 2 \times \sin(F)] +
sizeH \times Fadd +
\frac{X_{max} - X_{min}}{\Delta X} \times Fadd +
\left(\frac{X_{max} - X_{min}}{\Delta X}\right) + 1 \times \frac{Y_{max} - Y_{min}}{\Delta Y} \times Fadd +
\left(\frac{X_{max} - X_{min}}{\Delta X}\right) + 1 \times \left(\frac{Y_{max} - Y_{min}}{\Delta Y}\right) + 1 \times
\]

\[
(|Z| - 1) \times Fadd
\]
\[ FLOP_{\text{main}} = (\text{Number of times function contrib is called from the function Compress.}) \times (\text{Computational intensity of the function contrib.}) \]

\[ = |H| \times |\phi| \times \left( \frac{X_{\text{max}} - X_{\text{min}}}{\Delta X} \right) \times \left( \frac{Y_{\text{max}} - Y_{\text{min}}}{\Delta Y} \right) \times |Z| \times [9 \times F_{\text{add}} + 8 \times F_{\text{sub}} + 13 \times F_{\text{mul}} + 2 \times F_{\text{div}} + \sin (F) + \cos (F) + 2 \times \sqrt{F}] \]

This equation can be easily approximated by looking only at the main processing part: \( FLOP = FLOP_{\text{main}} \), since \( FLOP_{\text{main}} \) is significantly greater than \( FLOP_{\text{pre}} \) when the number of output samples is significantly greater than 1 and when the number of output samples is greater than the number of antenna positions, which is in practice always the case. The reason behind this is that the number of FLOPs in the preprocessing part is proportional to the sum of the number of antenna positions and the number of output samples, while the number of FLOPs in the main processing part is proportional to the multiple of the number of antenna positions and the number of output samples.
Appendix B

NVIDIA OpenCL Bugs

Due to the immature NVIDIA OpenCL implementation, throughout the development, several bugs have been discovered. Most of them were resolved as the new hardware and the software were introduced. Here we give a list of the most severe bugs that we discovered:

- In the NVIDIA SDKv2, there is a severe bug that limits the number of work-items. Although, it should be possible to have millions of work-items per one dimension of the index space, the maximal number of work-items allowed by the SDK is 65000. The work-around this bug is to increase the work-item granularity, which can have a big impact on performance. This is resolved in the NVIDIA SDKv3.

- Sometimes the NVIDIA implementation of the OpenCL function `clFinish` returns a status that is not defined in the OpenCL specification: `CL_OUT_OF_RESOURCES`. The exact cause of this status, which aborts the execution is not known. It seems that this is a hardware-dependent bug, since for the same test case parameters and the same platform memory parameters, on the newer hardware we did not experience such bug. Moreover, it turned out that when we divided the large input set into smaller chunks of data, even the older hardware was able to execute the test case, even if all the parameters of the test case are corresponding well to the platform parameters.

- In our experiments two platforms with almost the same hardware parameters were used: NVIDIA Tesla C1060 and NVIDIA GTX 280. The only difference between these two platforms is in the size of the global memory: the Tesla has 4GB of global memory, while the GTX has 1GB of the global memory, whereas GTX has higher memory bandwidth. Although the only visible hardware difference is in the memory, some test cases with the memory requirements less than 1GB were able to execute on the Tesla and not on the GTX. Finally, it is interesting to notice that the difference in price between these two platforms is fairly big (the Tesla C1060 costs around 700€, while the GTX 280 costs around 100€).