Master thesis

Adaptation of algorithms for underwater sonar data processing to GPU-based systems

by

Patricia Sundin

LIU-IDA/LITH-EX-A–13/029–SE

June 2013

Linköpings universitet

Master thesis

Adaptation of algorithms for underwater sonar data processing to GPU-based systems

by

Patricia Sundin

LIU-IDA/LITH-EX-A–13/029–SE

June 2013

Supervisors:	Usman Dastgeer, Alexander Lundh
Examiner:	Christoph Kessler

Abstract

In this master thesis, algorithms for acoustic simulations in underwater environments are ported for GPU processing. The GPU parallel computing platforms used are CUDA, OpenCL and SkePU. The purpose of this master thesis is to adapt and evaluate the ported algorithms' performance on two modern NVIDIA GPUs, Tesla K20 and Quadro K5000.

Several optimizations, described in existing literature for GPU processing (e.g. usage of shared memory, coalesced memory accesses), are implemented and multiple versions of each algorithm are created to study their trade-offs.

Evaluation on two GPUs showed that different versions of the same algorithm have different performance characteristic and execution with the best performing version can give better performance than the original algorithm executing on 8 CPUs. A performance comparison between CUDA, OpenCL and SkePU versions of one algorithm is also made.

Acknowledgments

I would like to thank my examiner Christoph W. Kessler for his support during this project. Many thanks also to my supervisor Usman Dastgeer for all the help and advice, and for putting up with all my email conversations.

I would like to give thanks to my supervisor at Saab, Alexander Lundh, and to Kristian Karlefur for all the help with this project. Thanks also to Per-Ola Svensson at Saab who helped me with the understanding of the original application. I would like to thank Niklas Pettersson and Mikael Niemi for helping me with issues in the the everyday work.

Finally, I would like to thank my family for their support and encouragement throughout the entire project.

Contents

1 Introduction **1**
1.1 Methods and sources 1
1.2 Limitations 2
1.3 Structure 3
1.4 Definitions 4

2 GPU computing **5**
2.1 CUDA 7
2.1.1 CUDA programming model 7
2.1.2 CUDA best practice 9
2.2 Other frameworks 9
2.2.1 OpenCL 9
2.2.2 SkePU 12
2.3 Well-suited problems for GPUs 13
2.3.1 Data parallelism 13
2.3.2 Typical applications 14

3 Performance analysis tools **15**
3.1 NVIDIA Visual Profiler 15
3.2 NVIDIA Nsight 15
3.3 Comparison and Discussion 16

4 Application analysis **17**
4.1 Noisegen() 18
4.2 Edatacalc() 19

5 Algorithm adaptations **22**
5.1 Implementation and adaptations, CUDA 22
5.1.1 Noisegen() 22
5.1.2 Edatacalc() 24
5.2 Implementation and adaptations, OpenCL 24
5.3 Implementation and adaptations, SkePU 25

6 Result and evaluation **28**
6.1 Noisegen() . . . 29
6.1.1 Sample speedup . . . 29
6.1.2 Channel speedup . . . 31
6.1.3 Discussion . . . 33
6.2 Edatacalc() . . . 35
6.2.1 Sample speedup . . . 35
6.2.2 Channel speedup . . . 37
6.2.3 Source speedup . . . 39
6.2.4 Discussion . . . 41
6.3 Discussion: Threads per block . . . 42

7 Related work **44**

8 Discussion and conclusion **46**
8.1 Noisegen . . . 47
8.2 Edatacalc . . . 48

9 Future work **49**

Chapter 1

Introduction

An application which is used to perform calculations for acoustic simulations in underwater environments is provided by Saab Dynamics. The application was originally running on a CPU. The main problem solved by this master thesis is to port a number of selected algorithms within the application from CPU to GPU, and to adapt and evaluate these algorithms in order to achieve better performance. The calculations for acoustic simulations are very performance demanding and this is the reason for wanting to use GPU rather than merely CPU. Also, with the power of a GPU, more advanced and realistic simulations can be performed.

In the application's GUI, the user can load files with sonar and scenario information. Other parameters, such as number of objects, sample frequency, and other, can be set by the user in the GUI. The algorithms that have been chosen for porting (see Chapter 4) are executed when a scenario is simulated in the application.

The given application is a smaller part of a larger application. The given application calculates simulated sonar signals and sends the results to other parts of the parent application which in turn performs beamforming calculations on these simulated signals. The parent application also contains a GUI where the result of the beamforming is shown.

1.1 Methods and sources

A literature study is made in the beginning of the project in order to get a better understanding of which algorithms, within the given application, are suitable to port to GPU and which adaptations can be done in order to achieve good performance. When the algorithms are ported, their performance is tested and adaptations are made to increase performance. The ported algorithms' performance is evaluated on two graphic processors that are based on the NVIDIA Kepler architecture; a NVIDIA Tesla K20 graphic processor and a NVIDIA Quadro K5000 graphic processor.

The primary parallel computing platform that is used when porting the algorithms is CUDA [16]. Besides CUDA, OpenCL [12] is used as a secondary computing platform together with the C++ skeleton programming library SkePU [4]. A comparison is made between the performances of an algorithm ported using CUDA, OpenCL, and SkePU respectively. In addition, a discussion is made about the following:

- Which method is the easiest to use when porting the algorithm?
- Is skeleton programming flexible enough for porting the algorithms?
- Can some symmetries be drawn between each method's ease of programming and its given performance?

When testing the application, files have been used to simulate sonar with a number of signal samples in a number of channels. When studying the performance of the ported algorithms, the following tools have been used:

- NVIDIA's Visual Profiler
- NVIDIA's Nsight for Visual Studio

These tools made it possible to examine each kernel's execution time, time spent copying data to/from GPU memory, and other. More information about the tools used in this project and what they are used for can be found in Chapter 3.

The functionality of the ported algorithms is validated by comparing the original and the ported algorithms' signal data sent to other parts of the larger application. Also the beamforming output in the larger application's GUI were used to detect differences and similarities between the original and the ported algorithms.

There are almost no articles where porting to GPU is treated with the same algorithms as in this master thesis. Therefore, most sources and references used in this master thesis are from articles and books which treat porting of other algorithms with CUDA and OpenCL, optimization of algorithms for GPU processing, and comparisons between CUDA and OpenCL applications. Some sources are from the platform developers homepage, for example NVIDIAs CUDA C Programming Guide [17]. More about related works can be found in Chapter 7.

1.2 Limitations

Only two profiling and performance analysis tools have been used in this master thesis. There are other profiling tools, especially for analyzing OpenCL applications, but those are not used in this project because of lack of time.

Two functions have been chosen to be ported for GPU processing. The first function, *Noisegen*, have been ported with CUDA, OpenCL and SkePU.

The second function, *Edatacalc*, have only been ported with CUDA because of lack of time. The Edatacalc function might also be harder to implement in SkePU than the Noisegen function because it preforms more calculations on more different arrays. It might require a new variant of the ARRAY_FUNC [1] user function generator macro in order to be able to port the function with SkePU. However, because of lack of time, this could not be evaluated and therefore only the Noisegen function is implemented in SkePU.

There might be other functions within the application which also are suited for GPU processing. However, because of lack of time, the application could not be studied in more detail to find these functions. Also, there would not be enough time to port more than two functions.

The two OpenCL versions that are implemented do not allow the programmer to choose the number of threads nor the number of blocks of which the kernel shall be executed with. It is possible in OpenCL to do this, but it is not an available option in these versions because of lack of time.

1.3 Structure

The structure of the report is as follows:

- A general explanation of GPU computing is given in Chapter 2.
- Chapter 3 explains different performance analysis tools that are used, and the benefits and disadvantages of these tools.
- The functionality of original algorithms is described in Chapter 4. Also an explanation of why these algorithms have been chosen for porting is given.
- In Chapter 5, the implementations and adaptations of the ported algorithms with CUDA, OpenCL and SkePU are discussed.
- The performance results of the algorithms are presented and evaluated in Chapter 6.
- Chapter 7 gives some examples on related works to this master thesis.
- A more detailed discussion of the results of the porting is given in Chapter 8. Also a conclusion of which method that is most preferable is presented.
- Some improvements for future work are given in Chapter 9.

[1] `http://www.ida.liu.se/~usmda/skepu/doc/html_v1.0/group__userfunc.html`

1.4 Definitions

CPU	Central Processing Unit.
CUDA	Compute Unified Device Architecture, a parallel computing platform and programming model created by NVIDIA.
GPU	Graphic Processing Unit.
GUI	Graphical User Interface.
IDE	Integrated Development Environment, a software application that usually provides a text editor, compiler, and debugger, all in one in order to facilitate software development.
Kernel	Function that is executed on the GPU.
OpenCL	Open Computing Language, a framework for writing programs that execute across heterogeneous platforms consisting of CPUs, GPUs, DSPs and other processors.
SIMD	Single Instruction Multiple Data, this means that the same instruction is performed on multiple data elements simultaneously.
SIMT	Single Instruction Multiple Thread, this means that a single instruction is executed by multiple threads simultaneously.
SkePU	Skeleton Programming Framework for Multicore CPU and Multi-GPU Systems.
SM	Streaming Multiprocessor.
SP	Streaming Processor.
SSE	Streaming SIMD Extensions.

Table 1.1: Definitions

Chapter 2

GPU computing

General purpose computing on GPUs (GPGPU) implies that a GPU is used together with a CPU to accelerate general-purpose science and engineering applications. With GPU computing, the compute-intensive parts of the application are transferred to the GPU while the rest of the application continues to run on the CPU.

The CPUs design is optimized for sequential code performance. A single thread of execution is allowed to be executed, at instruction level, in parallel or out of the sequential order with help of control logic in the CPU, while maintaining appearance of a sequential execution. Instruction and data access latencies to main memory in the CPU are reduced with large cache memories.

The design of GPUs is optimized for compute-intensive, highly parallel computation and for floating-point calculations. Data-parallel problems that have more computations than memory transfers are particularly profitable to run on GPUs. Compared to CPUs, GPUs have more transistors dedicated to data processing rather than data caching and flow control. The architecture of a CUDA-capable GPU is organized into an array of highly threaded streaming multiprocessors (SMs). Each SM has a number of streaming processors (SP) that shares instruction cache and control logic. Resource allocation, scheduling and thread creation are handled by the SMs in hardware. There is a minimal cost of employing many threads since threads are managed in hardware.

CUDA-capable GPUs have Graphic Double Data Rate (GDDR) DRAM, also called global memory. The GDDR DRAM functions as the frame buffer memory when graphics applications are running. When general-purpose science and engineering applications are running, the GDDR DRAM functions as a very-high-bandwidth, off-chip memory, but with a bit longer latency than a typical system memory. When running massively parallel application, the long latency is not a problem thanks to the high bandwidth.

The Tesla GPU architecture is good for nongraphics applications. It has a more generic parallel programming model with barrier synchronization, hierarchy of parallel threads, and atomic operations to dispatch and manage compute-intensive, highly parallel computations. The Tesla architecture also has L1/L2 cache and shared memory. There is one L1 cache per SM, whose purpose is to improve bandwidth and reduce latency. The purpose of the L2 cache is to act as data unification between the SMs. This master thesis uses a Tesla K20 GPU and a Quadro K5000 GPU, both are based on the Kepler architecture. An overview of the Kepler architecture and the Kepler memory hierarchy is shown in Figure 2.1 and 2.2 respectively.

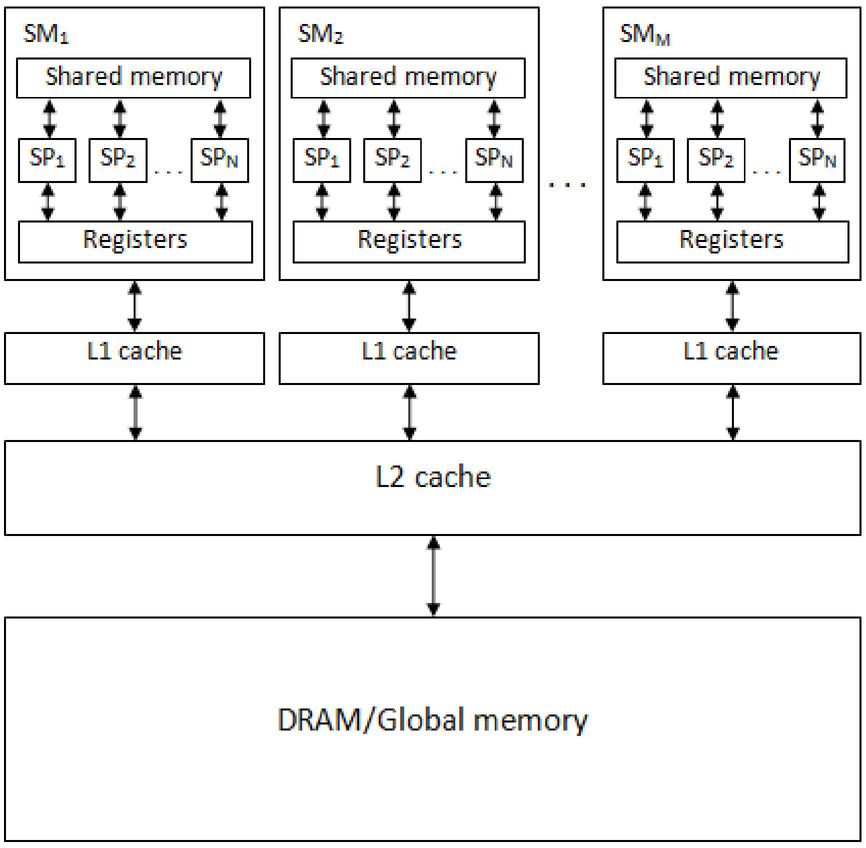

Figure 2.1: Overview of the Kepler architecture.

It is difficult to increase the memory bandwidth in CPUs because the CPUs have to satisfy many different requirements. GPUs, on the other hand, have simpler memory models and fewer design constraints which makes it easier to increase the memory bandwidth. The different design philosophies for CPU respectively GPU are shown in Figure 2.3.

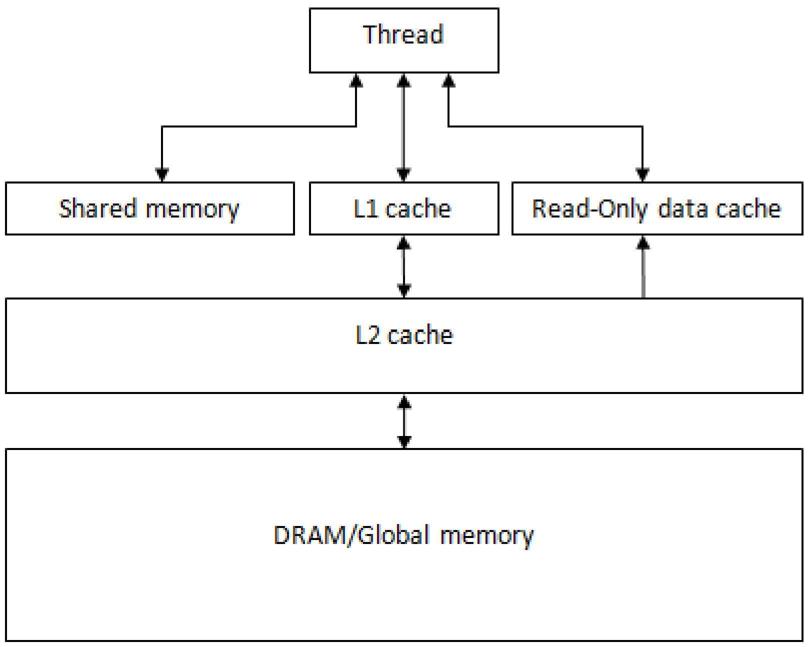

Figure 2.2: Kepler memory hierarchy.

GPGPU can be very powerful when running applications that require both serial and parallel processing, compared to running such applications on CPUs only. The few cores, optimized for serial processing, in CPUs and the thousand smaller cores, designed for parallel performance, in GPUs makes it possible to efficiently process both the serial and the parallel part of an application.

2.1 CUDA

The main parallel computing platform and programming model that has been used in this master thesis is CUDA. This section describes the CUDA programming model and some practices that should be followed when using CUDA.

2.1.1 CUDA programming model

When programming with CUDA, two different platforms are used concurrently. There is a *host* system with one or several CPUs, and there is one or several CUDA-enabled NVIDA GPU *devices*. The main differences between host and device are:

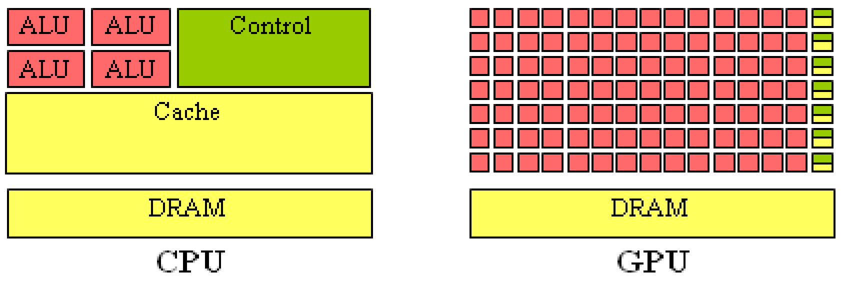

Figure 2.3: Difference in area use between CPUs and GPUs.

- Threading resources: The host system supports much fewer threads to run concurrently compared to the device.
- Threads: CPU threads are heavyweight while GPU threads are lightweight. The design of GPU is optimized for maximizing throughput while the design of CPU cores is optimized for minimizing latency.
- RAM: The host and the device have their own physical memories, separated by a PCI Express bus.

A CUDA program consists of one or more threads on the host processors, and these host threads executes one or more parallel kernels on a GPU device. Each kernel thread executes a scalar, sequential program across a set of parallel *threads*. In the CUDA programming model, the threads are grouped into *blocks*. The threads within a block can communicate and synchronize with each other; they share a software data cache (L1) and a so called *shared memory*, and they are executed on a single multiprocessor. The total size of a block is limited to 512 or 1024 threads depending on the GPU architecture. The GPUs used in this master thesis have a maximum limit of 1024 threads per block. The blocks are grouped into a *grid*, and blocks within a grid are independent from each other. A *warp* consists of 32 threads from a single block. Instructions in the code are executed per warp. If a memory operation is not ready to be issued, the warp will stall. The multiprocessor will then select another ready warp and switch to that one. This will keep the cores busy and productive, as long as there is sufficient parallelism in the code. The context switching between the warps must be fast if it shall be efficient. Since thread blocks always are created by a multiple of 32 threads (a number of warp units), most efficient execution will be given if the chosen size of a thread block also is a multiple of 32 threads.

2.1.2 CUDA best practice

When porting and adapting a application for CUDA, there are some steps that should be followed:

1. Study the application and localize the parts/algorithms that can be executed on a GPU.
2. Parallelize the selected algorithms from step 1 for GPU execution.
3. Optimize the modified algorithms to improve performance. Compare the result to the original application.

If the result from step 3 is not good enough, another iteration of the steps above can be performed. In order to achieve good performance, there are a few things that are worth focusing on. NVIDIA's guide for tuning CUDA applications for Kepler [21] gives the following general recommendations:

- Find ways to parallelize sequential code.
- Minimize data transfers between the host and the device.
- Adjust kernel launch configuration to maximize device utilization.
- Ensure global memory accesses are coalesced. This basically means that consecutive memory addresses in the global memory should be accessed by all of the threads in one half-warp at the same time.
- Minimize redundant accesses to global memory whenever possible.
- Avoid different execution paths within the same warp.

2.2 Other frameworks

The secondary parallel computing platforms that are used in this master thesis are OpenCL and SkePU. This section describes the OpenCL framwork and the SkePU skeleton programming library.

2.2.1 OpenCL

OpenCL, *Open Computing Language*, is an open standard for cross-platform, parallel programming on GPUs and designed by the Khronos group for portable, general purpose computing on GPUs. It offers portability between CPUs and GPUs from different vendors (compared to CUDA which is bound to NVIDIAs GPUs).

Conceptually, CUDA and OpenCL are quite alike, but there are some differences in terminology [26],[6], see Table 2.1, and kernel syntax [10].

CUDA	OpenCL
Thread	Work-item
Block	Work-group
Grid	Index space, NDRange
Host CPU	Host
Streaming multiprocessor (SM)	Compute unit (CU)
Scalar core	Processing element (PE)
Host thread	Host program
Shared memory	Local memory
Constant memory	Space constant memory
Texture memory	Space constant memory

Table 2.1: Differences in terminology between CUDA and OpenCL

The OpenCL architecture is defined in four parts: a *Platform Model*, a *Memory Model*, a *Programming Model*, and an *Execution Model*. The Platform model consists of a host which is connected to one or more OpenCL devices. Each device is composed of one or more compute units, and each compute unit is divided into one or more processing elements. The memory model defines the different types of OpenCL memories, see Table 2.2.

Memory	Description
Global	Accessible by all work-items, visible to all work-groups
Constant	Read-only memory and visible to all work-groups
Local	Shared within a work-group
Private	Private to a work-item
Host memory	The CPUs memory

Table 2.2: OpenCL memory model [23], [26].

There are no guarantees of consistency between the work-groups and the memory management is explicit, in other words, the data must be moved from host, to global, to local, and then back.

There is a data parallel and a task parallel programming model. The data parallel programming model offers one-to-one mapping between work-items and elements in a memory object, and work-groups can be defined implicitly (the programmer only defines the number of work-items and OpenCL handles the work-group creation) or explicitly (as in CUDA). The task parallel programming model defines a model where the kernels are executed independently of the index space. The two programming models also offer synchronization between items in a work-group and between commands in a context command queue.

The execution model can be divided into two parts, host and kernel execution. The OpenCL host program runs on a host and submits work to the devices. The kernels are functions that are executed on one or more devices. A work-item is an instance of a kernel. The work-items and work-groups are defined when the host submits a kernel, as in a CUDA program. A context is a collection of devices and it refers to the environment in which the kernels can be executed. The execution between the host and the kernels is asynchronous. The coordination of kernel executions on the devices is handled by so-called command queues. Commands can be memory and kernel synchronization commands, and they can be executed in-order or out-of-order.

One of the differences between OpenCL and CUDA is the effort of setting up the GPU for kernel execution. A lot more effort has to be made when OpenCL is used for context creations, data copying, kernel mapping, and so on. On the other hand, the effort of writing kernel code is almost the same in OpenCL and CUDA. A typical OpenCL program flow [26] can be as follows:

- Select the desired devices (for example all GPUs)
- Create a context
- Create command queues (per device)
- Compile program
- Create kernels
- Allocate memory on devices
- Transfer data to devices
- Execute kernel
- Transfer results back
- Free memory on devices

In contrast to CUDA, OpenCL compiles its kernels at runtime and requires environmental setup on the CPU before the kernels can be launched at the GPU. This may add to the OpenCL codes execution time. Since OpenCL is a portable language, CUDA-written code often results in better performance than OpenCL-written code. However, the runtime compilation of the kernel in OpenCL can generate code that makes better use of the target GPU, thus resulting in better performance.

2.2.2 SkePU

SkePU is a skeleton programming framework for multicore CPU and multi-GPU systems. It is a C++ template library designed to provide the application programmer with a higher abstraction level with the use of high-order functions, so-called *skeletons*, when specifying data- and task-parallel computations. SkePU supports execution on multi-GPU systems both with CUDA and OpenCL, and it also contains support for a sequential CPU and a parallel OpenMP backend.

Skeletons are predefined generic components which are derived from higher-order functions. Skeletons provide abstraction with a generic sequential high-level interface and this may give structure to parallel applications. The details involved in parallel computation structure are concealed from the user when a suited set of skeletons exists for the computations. The term *skeleton programming* refers to the approach where applications are written with the help of skeletons.

The advantages with skeleton programming are that parallelism and synchronization are given with very little effort. Skeletons are also beneficial when it comes to programmability, portability and performance:

- **Programmability:** Skeletons are simplifying the construction of applications by raising the level of abstraction. With a sequential interface to the outside world, applications can be written almost the same way as structured sequential applications are constructed. The low-level concurrency issues such as communication, synchronization, and the load-balancing are hidden by the skeleton.
- **Portability:** Portability are enhanced since the description of the algorithmic structure are modeled in a platform-independent manner by a skeleton interface. This makes the implementation portable across different platforms. Also the programmer is released from the responsibility of detailed realization of the underlying patterns.
- **Performance:** By exploiting knowledge about parallelism, synchronization and communication, optimization can be done for a skeleton implementation despite the skeleton's generic interface, and thereby performance can be improved.

Another advantage is that the programmer does not need to write CUDA, OpenCL, or OpenMP code by hand when using SkePU. The choice of backend is made by turning on/off some flags. The disadvantage with skeletons is that computations that do not fit the predefined skeletons have to be written manually.

Several preprocessor macros have been implemented in SkePU in order to easily be able to define functions that can be used with the skeletons. These macros expand to the right kind of structure that constitutes the function and they can be written and used with the skeletons regardless of the target architecture. The skeleton functions in SkePU are represented by objects. SkePU also includes implementations for Vector and Matrix containers to support skeleton operations.

SkePU uses so-called *lazy memory copying* to avoid unnecessary memory transfer operations between main memory and device memory. The SkePU vector and matrix containers keep track on which parts of it are currently allocated and uploaded to the GPU. Elements in a container that have been modified by computations are not immediately transferred back to the host memory. The container does not copy back the elements until the host wants to access an element. Lazy memory copying is preferable when several skeletons operating on the same data are called one after the other and when no modifications of the container data have to be performed in between by the host.

2.3 Well-suited problems for GPUs

GPUs are best suited for problems and applications that include data parallelism, see Section 2.3.1. If an application includes data parallelism, it is easier to achieve a speedup. Applications that involve large problem sizes and more complex models are suited for parallel computing. The reason for this is because these applications typically process large amounts of data and/or do a lot of iterations on the data, which is what GPUs are optimized for. It is however important that these applications are formulated in the right way in order to achieve good performance on the GPU. The problem must be decomposed into subproblems that can be solved at the same time without conflicts.

Most parts of an application must usually be executed sequentially. These parts should not be executed on a GPU; a CPU can execute them much better. A list of some typical applications that are suited to be executed on GPUs can be found in Section 2.3.2.

2.3.1 Data parallelism

Data parallelism is a form of parallelization where the same computation is executed on each data elements of an large array in parallel. In order for this to work, all element computations must be independent of each other. A simple example of data parallelism is to increment values of all elements in an array by some constant.

2.3.2 Typical applications

GPUs are for the most part used for image and media processing applications, but they can also be used for accelerating science and engineering applications. The following is a list of some typical applications that are suitable to be executed on GPUs:

- Video encoding/decoding
- Stereo vision
- Image scaling
- Pattern recognition
- Signal processing
- Computational finance, biology and chemistry
- Physics simulations

Chapter 3

Performance analysis tools

Two performance analysis tools have been used when analyzing the ported algorithms' performance. These tools are described in this chapter along with the advantages and disadvantages of each tool and how each tool is used in this master thesis. The two tools complement each other; where one tool performs unsatisfactory, the other tool can be used instead.

3.1 NVIDIA Visual Profiler

NVIDIA Visual Profiler is a stand-alone, cross-platform performance profiling tool. It can analyze and provide feedback on CUDA C/C++ applications. With the *timeline* option, the user can view CUDA activities such as memory transfers, kernel launches, and other API functions, that occur on both the CPU and GPU in one timeline. The programmer can also choose specific parts of the code where the profiler should collect information by adding the commands *cudaProfilerStart()* and *cudaProfilerStop()*. This results in no unnecessary collection of data. Visual Profiler is able to collect metrics[1] such as DRAM read/write throughput, branch efficiency, and cache hit rate. Moreover, it is able to collect instruction/memory/cache event data such as warp/thread launches, active cycles/warps, and others (event data can be different on different GPU's).

3.2 NVIDIA Nsight

NVIDIA Nsight is a tool for debugging and analyzing CPU and GPU code. The tool can be integrated with the IDEs *Visual Studio* [15] and *Eclipse* [7]. With the *timeline* option, the user can view activities such as memory transfers, kernel launches, and other API functions, that occur on both the CPU

[1]Metrics references: `http://docs.nvidia.com/cuda/profiler-users-guide/index.html#metrics-reference`

and GPU. The Nsight tool can also collect performance information about instruction statistics, memory statistics, branch statistics, instruction efficiency, achieved occupancy, and more [2].

The Nsight tool can analyze different types of applications, among others CPU, CUDA, and OpenCL applications. It is also possible to choose which kernel shall be analyzed without adding extra lines of code into the application code.

3.3 Comparison and Discussion

In this project, the Visual Profiler has mostly been used for quickly generating timelines. From this timeline, kernel execution time and memory copy execution time could easily be observed. Nsight has mostly been used to study the memory usage, CPU threads, memory transfers, kernel launches, and other API functions. Moreover, it has been used to study detailed performance information about the application.

A problem with the Visual Profiler tool is lack of proper documentation. In that aspect, Nsight is preferable because it gives a better description of what the collection data actually collects and it presents the collection data better than the Visual Profiler tool.

Another problem with Visual Profiler is that, in order to collect data, the application needs to be executed several times in a row. This is a big problem since the tester may need to set preferences, press run, and then close the program repeatedly, which is time consuming. This is not a problem when using the NVIDIA Nsight tool which only has to execute the application once to collect data.

The Nsight tool does not support the Tesla K20 card. Because of this, the Visual Profiler tool is used when analyzing the algorithms on the Tesla K20 card. The Visual Profiler tool can only profile CUDA applications, and because of this, the OpenCL applications can only be analyzed with the Nsight tool.

[2]Nsight Visual Studio User Guide: `http://http.developer.nvidia.com/NsightVisualStudio/2.2/Documentation/UserGuide/HTML/Nsight_Visual_Studio_Edition_User_Guide.htm`

Chapter 4

Application analysis

This chapter discusses the selected algorithms original design and rationale for selection of these algorithms.

After studying the application with the knowledge gained from the literature study, two functions were found suitable for GPU processing. Those function were *Noisegen()*, see Section 4.1 and *Edatacalc()*, see Section 4.2. Together these functions create signals with noise which is used as input signals for beamforming processing. The functions are supposed to simulate signal samples that have been generated from a number of sources and then received in a number of channels.

There are almost no other functions in the application that are suited for GPU processing. One reason for this is because the other functions does not perform enough loop iterations. Therefore, no gain is given to port them to the GPU. Also, because of the limitations explained in Section 1.2, there were only enough time to implement two functions.

The amount of work spent in these two functions in the CPU thread from which they are triggered from, depends on the problem size and the number of CPU threads the programmer choose to execute the functions with. If there is a small problem size and the two functions are executed with eight threads each, little time will be spent executing the function and more time will be spent waiting for other functions to be executed. The trigger CPU thread's amount of work spent executing the functions can in that case be less than 5 % of the total amount of work for that CPU thread. If there is a large problem size and the two functions are executed with one thread each, then more time will be spent executing the functions and less time will be spent waiting for other functions to be executed (since these functions can be executed in other CPU threads during the executions of the Noisegen and Edatacalc functions). The trigger CPU thread's amount of work spent executing the functions can in that case be about than 99 % of the total amount of work for that CPU thread. No matter how fast the execution of the two functions are, the total amount of time for the trigger

CPU thread will not be lower than a certain value. This lowest value are determined by the other functions that are executed in other CPU threads.

4.1 Noisegen()

The purpose of the Noisegen algorithm is to calculate colored noise for signal samples. The noise shall be added to each of the calculated samples and channel in the function *Edatacalc()*. In the original code, the workload of the algorithm is divided between a number of CPU threads (the maximum is eight threads). Each thread receives a number of channels on which it is supposed to execute the algorithm in parallel to the other threads.

The Noisegen function consists of two nested for-loops. The computation that generates the noise is placed in the inner for-loop which loops through each channel's samples. The outer for-loop loops through the number of channels which the thread has been assigned. Other samples' noise values are not included in the calculation of the noise value for a sample, and this makes the two loops suitable to be parallelized on a GPU. Algorithm 1 shows the Noisegen function's pseudocode.

Algorithm 1: Noisegen in pseudocode

```
for each channel do
  for each sample do
    {Calculate noise for signal samples}
  end for
end for
```

The size of the vector with the noise calculation data depends on the number of threads. Each noise calculation data is of a `noisestate struct` type which contains 26 values of the type `double`. The size of the result vector with the calculated noise is the number of channels multiplied with the number of samples. Each value in the result vector is of the type `float`. Since the `noisestate struct` contains values of the type `double`, double precision must be used in the ported versions of the function in order to achieve the same accuracy in the result. Unfortunately, the use of double precision will increase the execution time of the function.

The access pattern of the noise calculation data is that each thread reads each variable's value, preforms some calculations, and then assigns the same variable with the new value. It also happens that it assigns one variables value to another variable. These assigns occur in each loop iteration.

The access pattern of the result data depends on which iteration the thread is at. The result of one iteration is written to the result array in the inner most loop, and the indexing of the result array will be as follows: $currentSample * numberOfChannels + currentChannel$. This way of indexing the array results in that the writes to the result array will not be to indices in subsequent order.

Although a large amount of data is being calculated and processed in the Noisegen function, it is still suitable to run on a GPU because of the large amount of loop iterations performed in each instance of the function. Since the calculated noise values are independent of other noise values, the calculations can be performed in parallel rather than in a loop. It is important though that each noise is calculated with fairly unique noise calculation data in order to attain the random noise values which the Edatacalc function requires.

A downside with the Noisegen function is that it cannot be executed on the GPU while the CPU continues to work since the CPU is dependent of the result from the function directly after the function has been executed. Noisegen can also not be executed in advance because the CPU does not know that Noisegen shall be executed until just before the execution.

4.2 Edatacalc()

The purpose of the Edatacalc algorithm is to create signal samples for a number of channels. The generated noise from the function *Noisegen* will be added to the calculated signal samples in the end of this function. The workload of the algorithm is, as for the Noisegen function, divided between a number of CPU threads (the maximum is eight threads). Each thread is assigned a number of samples on which it is supposed to execute the algorithm in parallel with the other threads.

The Edatacalc function consists of three nested for-loops with two extra for-loops within the innermost for-loop, see Algorithm 2. The first outer for-loop loops through the number of samples which the thread has been assigned. The second outer for-loop loops through all sources in the environment and the third for-loop loops through all channels. The third loop calculates the signal samples for four channels at a time. To do this, two for-loops following each other are executed within the third loop. The first inner for-loop assigns values to a number of arrays with four elements each. When the assignments are done, these arrays are used to calculate signals for four channels at a time with help of a function *Spline4* that uses SSE Intrinstics functions [9] instead of ordinary addition, subtraction and multiplication operators. The second inner for-loop loops four times in order to assign the result values from the Spline4 function to the correct index in the result array, one iteration for each channel. Algorithm 2 shows the Edatacalc function in pseudocode.

The Edatacalc function is, just like the Noisegen function, a good candidate for porting to the GPU because of the large amount of loop iterations performed in each instance of the function. Also, the calculated signal values are not dependent on other signal values and this makes it possible to perform the calculations in parallel. A drawback is that the original function uses values of the type `double`, this means that double precision must be

Algorithm 2: Edatacalc in pseudocode

```
for each sample do
  {Update platform};
  for each source do
    {Update source};
    for 4 channels at a time do
      for $i = [0..3]$ do
        {Fill the arrays which have four elements each};
      end for
      {Calculate signal samples, four channels at a time}
      Spline4(..);
      for $i = [0..3]$ do
        {Add results from Spline4 to result array};
      end for
    end for
  end for
end for
```

used in the ported versions of the function which results in higher execution times, see Chapter 5.

Another disadvantage with the Edatacalc function is that a large amount of data is used to calculate the signal samples. It is especially the source type data that contains very large arrays. It is unfortunately not possible to minimize the amount of source type data that has to be sent to the GPU in the ported version of the function, which will increase the function's execution time.

The access pattern of the data used for calculating the signal sample values is a bit arbitrary. The reads and write can be to the same or different variables, and the instructions performing the reads/write does not do it with variables that are in subsequent order to each other in the structs. Also, there are a lot of reads and writes to many different variables in each for-loop.

The access pattern of the result data is similar to the Noisegen function and depends on which iteration the thread is at. The result of one iteration is added to the result array in the inner most loop, and the indexing of the result array will be as follows: $currentSample * numberOfChannels + currentChannel$. This means that the result array is first read to get the old value, this value is then added to the result value of the current iteration's calculations, and this sum is then written to the same index that were read. This way of indexing the array results in that the reads and writes to the result array in one iteration will be to the same index, but the indices will not be accessed in subsequent order from iteration to iteration.

There are already optimizations in the Edatacalc function to make it execute faster on the CPU. The original code calculates the signal values for four channels at a time. This is done with the use of SSE Intrinsics functions. SSE is an SIMD instruction set extension to the x86 architecture designed by *Intel*. The programmer can use C++ function calls and variables with assembly-coded functions called *intrinsics* in place of assembly instructions. SSE instructions can be used directly from C++ code when SSE Intrinsics are supported. This eliminates the need of writing assembly instructions. By using SSE Intrinsics, function call overhead from the C++ code is eliminated. It provides the same benefits as using inline assembly; however, it is easier to write readable code. The use of SSE Intrinsics is a good optimization when running the application on CPU. The equivalent to SSE Intrinsics for NVIDIA GPUs is warp SIMT execution, where each individual thread in a warp executes the same sequence of instructions.

Chapter 5

Algorithm adaptations

This chapter describes how the chosen algorithms from Chapter 4 have been implemented in CUDA, OpenCL and SkePU.

Since the original versions of the Noisegen and the Edatacalc functions uses variables of the type double to calculate the results, double precision had to be used in order to achieve equivalent results. The disadvantage with double precision compared to single precision is that the functions executes slower by a factor of 2 on a NVIDIA Quadro GPU and by a factor of 3 on a NVIDIA Tesla K20 GPU.

5.1 Implementation and adaptations, CUDA

This section describes how the two algorithms Noisegen and Edatacalc, described in Chapter 4, are implemented in CUDA. The result vector is placed in the global memory in each version of the function.

5.1.1 Noisegen()

Four different versions of the Noisegen function have been implemented in order to be able to compare which kind of implementation gives the best performance. These four versions of the function are:

- CUDA_NV1: One GPU thread per channel (calculates noise for all samples inside a channel). The noise calculation data is placed in the global memory during entire kernel execution.
- CUDA_NV2: One GPU thread per channel (calculates noise for all samples inside a channel). The noise calculation data is placed in the shared memory.
- CUDA_NV3: One GPU thread calculates a number of samples inside a channel. The noise calculation data is placed in the shared memory.

- CUDA_NV4: One GPU thread calculates a number of samples inside a channel. The thread's part of the noise calculation data is copied to the thread's local registers.

The first version that was implemented, CUDA_NV1, assigned GPU threads to calculate noise for all samples for one channel each. This version is similar to the original code, the inner loop is the same and the outer loop is replaced by one GPU thread per outer loop-iteration, where each GPU thread is working in parallel. The noise calculation data is placed in the global memory, resulting in each thread reading and writing to global memory several times in each sample iteration. This may increase the execution time of the function since communication with the GPU's global memory is more time consuming compared to communication with shared memory or local registers.

The second version, CUDA_NV2, is implemented in the same way as CUDA_NV1 with the exception of the use of shared memory. Before calculating the noise for each sample, the noise calculation data for each thread is copied from global memory to shared memory. Because of the fact that the noise calculation data is changed in each iteration of calculation noise, it is less time consuming to save the changed data to shared memory than to global memory.

In both CUDA_NV1 and CUDA_NV2 each thread has to perform as many iterations as there are samples per channel. If for example the data set contains 2205 samples per channel, then each thread must perform 2205 iterations. To perform that many iterations is very time consuming and may result in poor execution times.

In the third version of the function, CUDA_NV3, each GPU thread calculated an arbitrary number of samples and shared memory is used for storing the noise calculation data. This version provides greater flexibility because the programmer can choose how many samples each thread will compute noise values for. The programmer can also choose how many threads that shall be executed per block, and how many blocks there shall be in each grid. These choices may affect the execution time of the function. The first and second version does not have this flexibility, the number of channels decides the number of threads. A problem with using shared memory is that if the number of threads per block is too large, the noise calculation data will not fit in the shared memory, which forces the programmer to use a smaller number of threads and this may increase the execution time for the function.

The fourth and final version of the Noisegen function, CUDA_NV4, is similar to the third version, but the noise calculation data is copied from global memory to registers and local memory for each thread instead of copying it to shared memory. The new noise calculation data is copied back to global memory when the thread has finished calculating the noise values for its share of the samples.

5.1.2 Edatacalc()

Three different versions of the Edatacalc function have been implemented in order to be able to compare which kind of implementation that gives the best performance. These three versions of the function are:

- EV1: One GPU thread per sample (calculates the signal for that sample over all channels and sources).
- EV2: One GPU thread:
  - EV2a: per sample and source;
  - EV2b: per sample, source and channel.

The first version, EV1, is almost identical to the original function except that the most outer for-loop had been replaced with one GPU thread per iteration. The variables that are used within the calculations are either copied to local variables for each thread or read directly from the global memory.

The second version, EV2, can be executed in two different modes. For the first mode, EV2a, both the first and second outer for-loop has been replaced with one thread per iteration. For the second mode, EV2b, all for-loops has been replaced with one thread per iteration. For EV2a and EV2b, there are a lot of redundant calculations which may increase the execution times. The third outer for-loop uses calculations from just before the third outer for-loop. If there is one thread per iteration and no for-loops (EV2b) then all the threads will perform one iteration each of the third for-loop calculate the same values before entering the third for-loop code. The same argument goes for the second outer for-loop, if there is one thread per iteration and there are no for-loops (EV2a) or only the third inner for-loop (EV2b).

It is possible to launch fewer threads than the total amount of iterations, this will result in that some threads do all the calculations one more time with another thread id. If there are much fewer threads than the total amount of iterations, a lot of redundant calculations will be performed, causing higher execution time. If the programmer wants to use few threads it is not recommended to use version EV2a or EV2b but rather the first version (depending on the amount of threads the programmer is willing to launch).

5.2 Implementation and adaptations, OpenCL

This section describes how the algorithm Noisegen, which is described in Chapter 4.1, is implemented using OpenCL.

Two different versions of the Noisegen function have been implemented in order to be able to compare which kind of implementation gives the best performance. These two versions are almost the same as the CUDA versions CUDA_NV1 and CUDA_NV4 of the Noisegen function, see Section 5.1.1. The noise calculation data and the result vector are placed in the global memory in each version of the function.

The first version that was implemented with OpenCL, OPENCL_NV1, is equivalent to CUDA_NV1, their kernel functions are exactly the same. The outer for-loop from the original function has been replaced with one thread per iteration, and the inner for-loop is unchanged. This means that each thread has to perform as many for-loop iterations as there are samples per channel.

The kernel code of the second version that was implemented, OPENCL_NV2, is almost the same as the kernel code of CUDA_NV4. The outer for-loop from the original function has been removed and replaced with a number of threads per iteration. The inner for-loop has been changed to only iterate through a chosen number of samples per thread. The difference between them is that CUDA_NV4 places the noise calculation data in registers and in local memory while OPENCL_NV2 places the data in global memory. However, their results may still almost be the same. The reason for this is because the registers will most likely be overflowed with data in CUDA_NV4 and the data may therefore be placed in the local memory, which is located in the global memory.

OPENCL_NV2 provides greater flexibility than OPENCL_NV1 because the programmer can choose how many samples each thread shall compute noise values for and thereby can a larger number of computations be parallelized.

The kernel codes for these two OpenCL versions are almost the same as CUDA_NV1 and CUDA_NV4. The kernel setup is on the other hand much different. More work has to be put down before a OpenCL kernel can be launched, as described in Section 2.2.1, and this may increase the total execution time of the functions.

The total number of threads is calculated in the code by the number of channels and the number of samples per channel defined in the GUI. The programmer are in these versions not allowed to choose the number of threads per block which the kernel shall be executed with, see Section 1.2. However, the programmer can choose the number of samples per thread in the second version, and thereby indirectly affect the total number of threads.

5.3 Implementation and adaptations, SkePU

This section describes how the algorithm Noisegen, which is described in Chapter 4.1, is implemented in SkePU.

Two different versions of the Noisegen function have been implemented with two backends each in order to be able to compare which kind of implementation gives the best performance. The versions are implemented with a CUDA and a CPU backend. These two versions are almost the same as CUDA_NV1 and CUDA_NV4, see Section 5.1.1.

A few modifications had to be done in the SkePU code in order to be able to port the Noisegen function with SkePU. Another user function generator macro had to be created. A new variant of the ARRAY_FUNC [1] user function generator macro was written by Usman Dastgeer (one of the developers of SkePU). The two first operands in the original ARRAY_FUNC are input operands (read) and the third is an output operand (write). In the modified ARRAY_FUNC, also the content of the first input operand can be modified (readwrite) while the other two are the same (second operand is read and third operand is write). A small modification was also made to make it possible for the programmer to choose which GPU to execute the function on.

The kernel functions of the two versions are both defined with the new variant of the ARRAY_FUNC user function generator macro and called with the *MapArray* skeleton. All calculations within the two for-loops in the original function are placed in the kernel function for each version. The few number of calculations which are performed before the two loops in the original function, are calculated outside the kernel function and then passed to the kernel each time it is invoked. The initiation of the noise calculation data were defined with the GENERATE_FUNC user function generator macro and called with the *Generate* skeleton.

The first version that was implemented with SkePU, SKEPU_CPU_NV1 and SKEPU_CUDA_NV1, is equivalent to CUDA_NV1. The outer for-loop from the original function has been replaced with a MapArray skeleton with one vector element per iteration, and the inner for-loop is unchanged.

The second version that was implemented, SKEPU_CPU_NV2 and SKEPU_CUDA_NV2, is equal to CUDA_NV4. The outer for-loop from the original function has been removed and replaced with a MapArray skeleton with a number of vector elements per iteration. The inner for-loop has been changed to only iterate through a chosen number of samples per vector element. All this is placed in the kernel function. This version provides greater flexibility than the first version because the programmer can choose how many samples each vector element shall compute noise values for and thereby can a larger number of computations be parallelized.

In contrast to CUDA, SkePU with CUDA backend does not allow the programmer to choose whether shared or global memory shall be used. That is on the other hand not a necessity in this case since the performance results of the CUDA versions shows that shared memory only results in a little bit lower execution time and that it limits the number of threads per block.

An advantage with implementing the Noisegen function with SkePU is

[1] `http://www.ida.liu.se/~usmda/skepu/doc/html_v1.0/group__userfunc.html`

that, when CUDA backend is used, the programmer does not need to hard code a specific number of threads per block or a specific number of blocks, SkePU chooses the numbers that it finds most suitable. This is good when the problem size changes at runtime and when perhaps a larger or smaller number of blocks gives better performance. The programmer can, if desired, choose the maximum number of threads per block that shall be used, SkePU then might set a lower number of threads per block if that gives better performance.

It is necessary that the programmer is able to define a maximum limit on the number of threads per block since SkePU sets the GPUs maximum number of threads per block as default and the best performance may not be given by using all thread in a block.

Chapter 6

Result and evaluation

This chapter lists the speedup results that were given when testing and evaluating the ported algorithms in Chapter 5. The CUDA and OpenCL applications (and the SkePU versions with CUDA backend) have been evaluated on two different GPUs during the tests: a NVIDIA Quadro K5000 [19] and a NVIDIA Tesla K20 [20]. There are two figures for each speedup, one with Quadro K5000 and one with Tesla K20. The speedup of the CPU versions are the same in both figures. The speedup is calculated as the following: Speedup = $\mathrm{T}_{\text{Original function(1 thread)}}$ / $\mathrm{T}_{\text{Ported function}}$.

During the tests, the following parameters have been varied:

- Number of samples
- Number of channels
- Number of sources (only for the Edatacalc tests)
- Number of threads per block and number of blocks (only for CUDA versions)
- Number of samples per thread (only for the third and fourth CUDA versions, the second OpenCL version, and the second SkePU version)

The function's execution time has been measured with the C++ function *clock()* from the time the functions have been invoked until the functions return. The average execution time has then been given by taking the average of ten execution times.

The performance analysis tools have measured time in the C kernel code; all the CUDA allocation function, CUDA copy functions, the kernel invoke, and CUDA free functions. These measured execution times are therefore smaller than when measuring with the clock function. The execution time of the kernel and the time spent in copying to/from GPU memory before and after the kernel launch has been measured with the Visual Profiler tool, which gives about the same result as the Nsight tool.

6.1 Noisegen()

This section present the Noisegen functions speedups compared to the original Noisegen function executed with one thread. More information about the Noisegen function can be found in Chapter 5. The OpenCL versions were only tested on the Quadro K5000 card because problems occured when using OpenCL with the Tesla K20 card [20].

6.1.1 Sample speedup

The sample speedups for CUDA_NV2 and CUDA_NV3 are not displayed in the plots because their speedups are almost the same as the speedup for CUDA_NV1 and CUDA_NV4 respectively.

The best sample speedup is given by OPENCL_NV2 on Quadro K5000 and CUDA_NV3 on Tesla K20. CUDA_NV4 also gives the second best speedup on Quadro K5000.

The original Noisegen function executed with eight threads, Org_8t, gives good speedup on Quadro K5000, but if the number of samples would increase, then would CUDA_NV4, OPENCL_NV2, and SKEPU_CUDA_NV2 most likely perform much better than the original function.

Figure 6.1 and Figure 6.2 show the sample execution time speedup on Quadro K5000 and on Tesla K20 respectively for each Noisegen version compared to the original function executed with 1 thread, Org_1t.

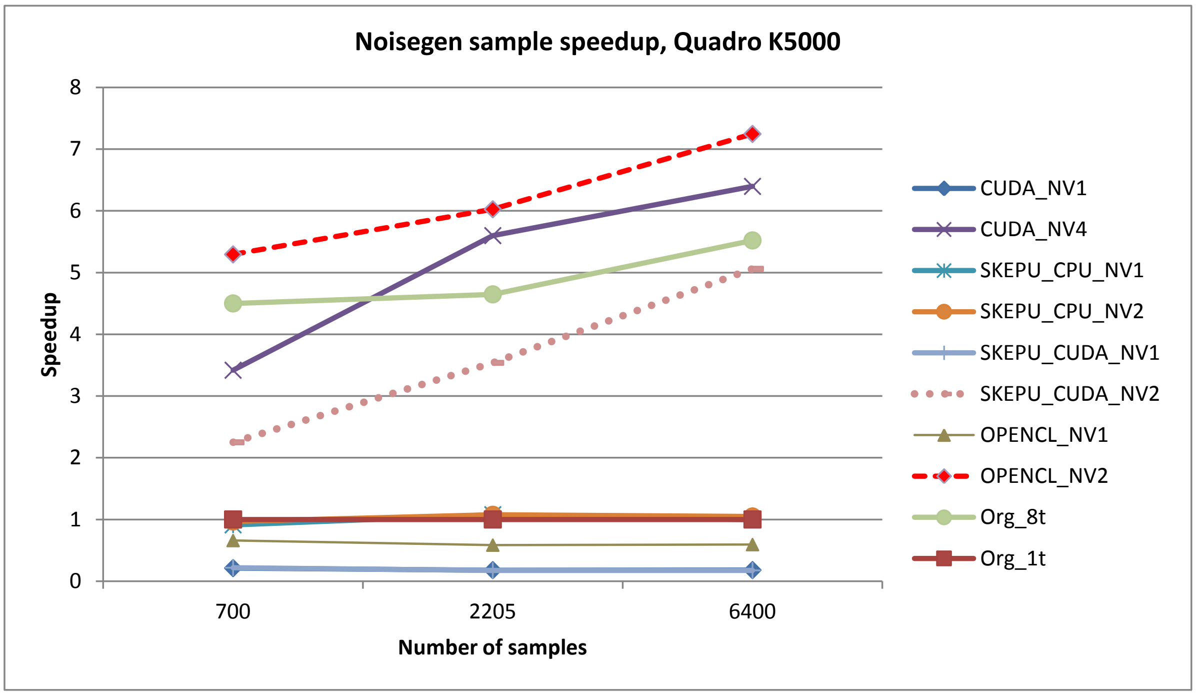

Figure 6.1: Sample execution time speedup on Quadro K5000 for each Noisegen version compared to the original function executed with 1 thread, Org_1t.

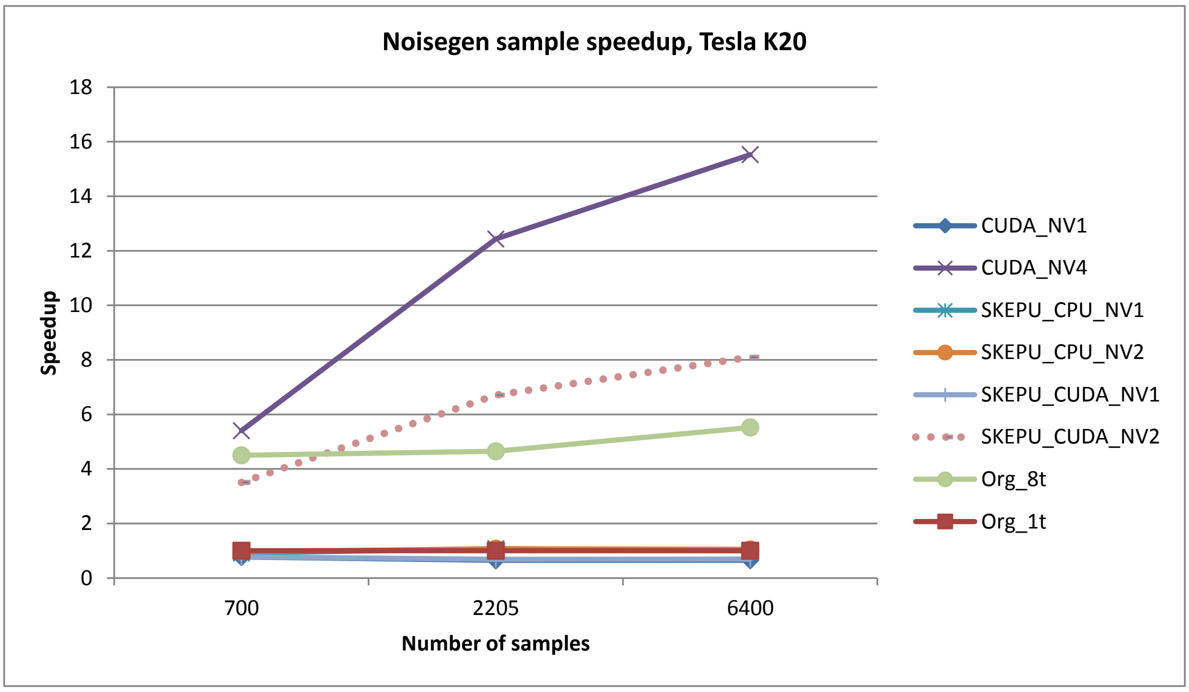

Figure 6.2: Sample execution time speedup on Tesla K20 for each Noisegen version compared to the original function executed with 1 thread, Org_1t.

6.1.2 Channel speedup

The best channel speedup is given by OPENCL_NV2 on Quadro K5000 and by CUDA_NV4 on Tesla K20. CUDA_NV4 version also gives the second best channel speedup on Quadro K5000.

The original Noisegen function executed with eight threads, Org_8t gives good speedup on Quadro K5000, but if the number of channels would increase, then CUDA_NV4, OPENCL_NV2, and SKEPU_CUDA_NV2 would most likely perform much better than the original function.

Figure 6.3 and Figure 6.4 show the channel execution time speedup on Quadro K5000 and on Tesla K20 respectively for each Noisegen version compared to the original function executed with 1 thread, Org_1t.

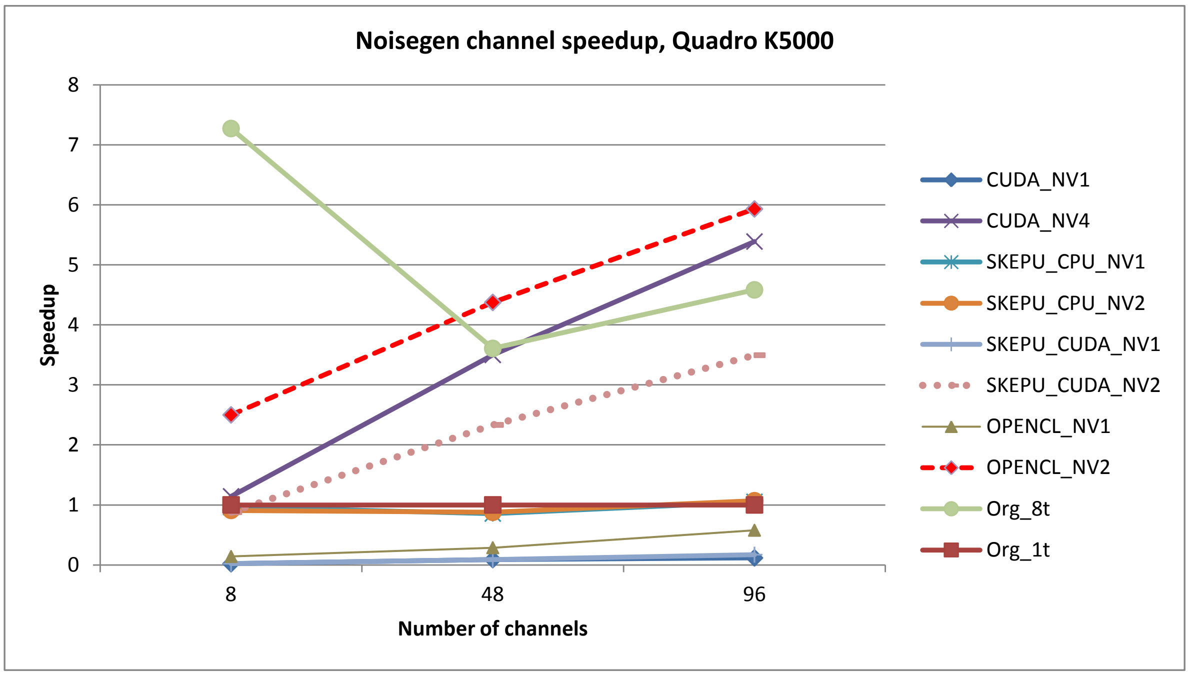

Figure 6.3: Channel execution time speedup on Quadro K5000 for each Noisegen version compared to the original function executed with 1 thread, Org_1t.

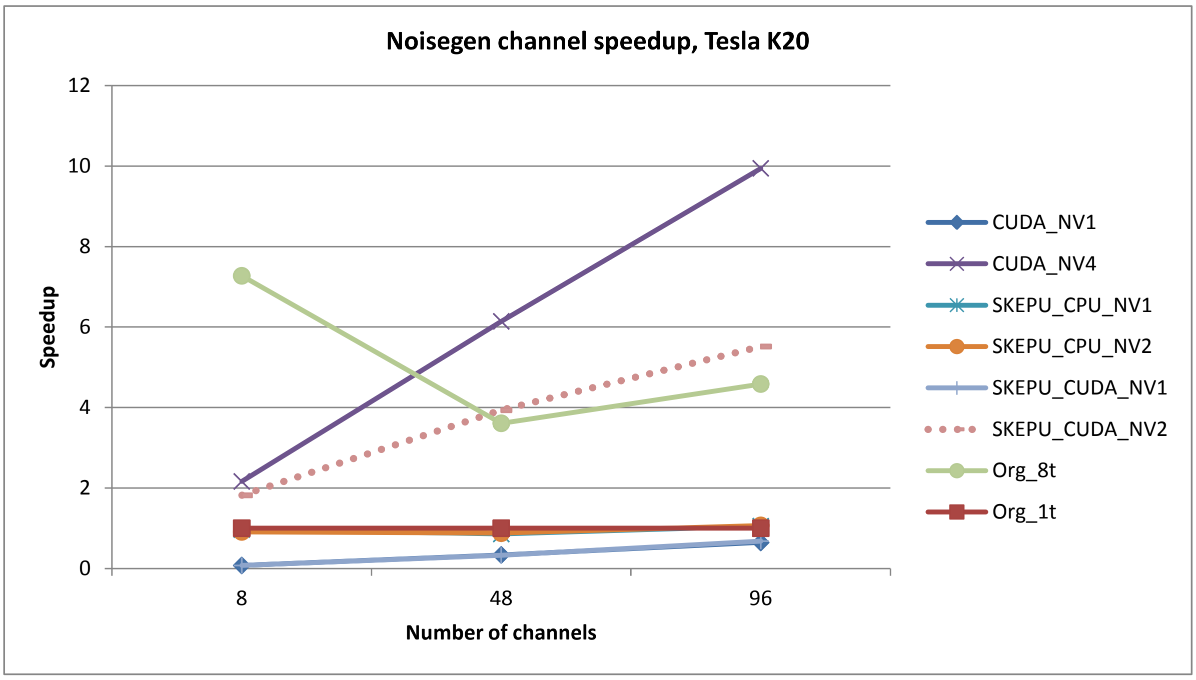

Figure 6.4: Channel execution time speedup on Tesla K20 for each Noisegen version compared to the original function executed with 1 thread, Org_1t.

6.1.3 Discussion

The execution times for CUDA_NV1, OPENCL_NV1, and SKEPU_CUDA_NV1 are all similar. The reason for this is because the kernel algorithms are the same. The speedups are therefore almost the same for all three versions.

The reason of why the execution times for SKEPU_CPU_NV1 is almost the same as the execution times of the original function when one thread is used is because they do the same amount of work. SKEPU_CPU_NV1 also runs on the CPU with one thread and is only a bit slower than the original function because SkePU has to do some extra work before the algorithm can be executed. SKEPU_CPU_NV1 also has higher speedup than CUDA_NV1, OPENCL_NV1, and SKEPU_CUDA_NV1 because the cost of running those versions on the GPU is higher than what it costs to do the same work on the CPU. Because of their kernel implementations they cannot yield enough parallelism for them to be profitable.

For CUDA_NV3 and CUDA_NV4, OPENCL_NV2, and SKEPU_CUDA_NV2, the time spent in copying data to and from the GPU memory before and after the kernel launch depends on the number of samples per thread. If there are few samples per thread, then more data has to be copied to the

GPU memory, and this takes more time. If there are a large number of samples per thread, then each thread will have to iterate a lot which increases the total execution time but it decreases the time spent in memory copy. These versions can parallelize the noise calculations for the samples in each channel, in other words, they can have one thread calculating the noise for a subset of the samples in one channel and thereby splitting up the inner of the two for-loops from the original version. This result in lower execution time and a better sample speedup compared to the other versions.

If CUDA_NV4 is being executed with the same parameters as CUDA_NV1 and CUDA_NV2, that is one thread for each channel, then it will run slower than those versions. The reason for this is the extra calculations that have to be performed in CUDA_NV4 to ensure that all variations of the chosen parameters give a correct result. CUDA_NV3 gives faster execution time when the same number of threads per block is used in CUDA_NV3 and CUDA_NV4, but CUDA_NV4 is able to give a lower execution time since it can use more threads per block.

CUDA_NV4, OPENCL_NV2, and SKEPU_CUDA_NV2 have almost the same execution times because their kernel implementations are almost the same. However, SKEPU_CUDA_NV2 have a bit lower execution time because of implementation differences in SkePU with CUDA backend and CUDA; the OpenCL versions have a bit higher execution times.

CUDA implementations generally result in better performance than OpenCL implementations. There are some cases though where the result is the opposite. The compilers used for translating the CUDA and OpenCL kernels into intermediate PTX assembly code, produce different results, even though the kernel codes are similar. Therefore there may be cases where the OpenCL compiler produces better PTX assembly code than the CUDA compiler. This is probably the case for CUDA and OpenCL versions in this master thesis, and the reason why OpenCL results in better speedup. The same outcome where OpenCL results in better performance, also occurred in two algorithms which were studied in a master thesis by Sanden [25].

As the graphs show (see Figure 6.1 - 6.4), the versions running on the GPU do not give that much speedup compared to the original version executed with eight threads. The reason for this is because the kernel threads communicate a lot with the global memory. The noise calculation data, used to perform the calculations, are in almost all GPU versions placed in the global memory. The threads read from and write to this noise calculation data several times in each kernel invocation, and this results in a lot of global memory traffic. Since communication with the global memory is slow, the speedup cannot get much better. CUDA_NV3 is an exception, where the noise calculation data is placed in the shared memory with the intention to decrease the communication with global memory. The problem with this implementation was that it limits the number of threads per block that could be used. If too many threads per block are used, the shared memory would overflow with data. The usage of shared memory limited the number of threads/block so much that the occupancy per SM deteriorated (available parallelism inside a SM decreased) and this resulted in an increase of the execution times. By putting the noise calculation data in the global memory, the maximum number of threads per block could be used if needed and lower execution times could be achieved than when shared memory is used.

The fact that the speedup is limited by the communication with the global memory is proven by analyzing the memory communication. There are a lot of load/stores transactions from/to the global memory and the kernel execution times decreases if the number of transactions decrease. A low number of transactions to global memory is necessary for good speedup because else the caches swap data in/out more often and this will cause the cache hit rate to decrease. It is also bad to have a lot of store transactions since stores are not cached in L2, and this decreases the L2 cache hit rate.

It is possible to increase the cache hit rates by, among other things, configuring the size of the L1 cache and the shared memory. By increasing the L1 cache size and decreasing the shared memory size with the CUDA function *cudaFuncSetCacheConfig*, the L1 hit rate more than doubled. However, the total execution time did not change particularly much, and the kernel execution only decreased with 2,3%. By adding dummy shared memory and having a low number of threads per block, the L1 hit rate could increase to 83,9% and the L2 hit rate to 46,9%. However, the execution time became five times higher. This because the occupancy decreased from 36,8% to 6,15% when the number of threads decreased, and this also decreased the speedup.

There is a trade-off, having more threads per block will cause a lot of cache misses and more global memory traffic. On the other hand, having low number of threads per block also give bad performance considering that not enough work will be done to hide global memory access latency, the occupancy will decrease.

6.2 Edatacalc()

This section present the CUDA Edatacalc versions' speedup compared to the original Edatacalc function executed with one thread, Org_1t. More information about how the CUDA versions are designed can be found in Chapter 5.

6.2.1 Sample speedup

The best sample speedup is given by the original Edatacalc function executed with 8 threads, Org_8t.

All CUDA versions parallelize the signal calculations for the samples in each channel, and therefore will the version with lowest global memory latency have the best sample speedup. This results in that EV1 has the second best sample speedup.

If the number of samples is increased to more than 6400 samples, then will EV1 most likely give better speedup than the original function executed with 8 threads. Also EV2a will most likely give better speedup if the number of samples is increased.

Figure 6.5 and Figure 6.6 shows the sample execution time speedup on Quadro K5000 and Tesla K20 respectively for each Edatacalc version compared to the original function executed with 1 thread, Org_1t.

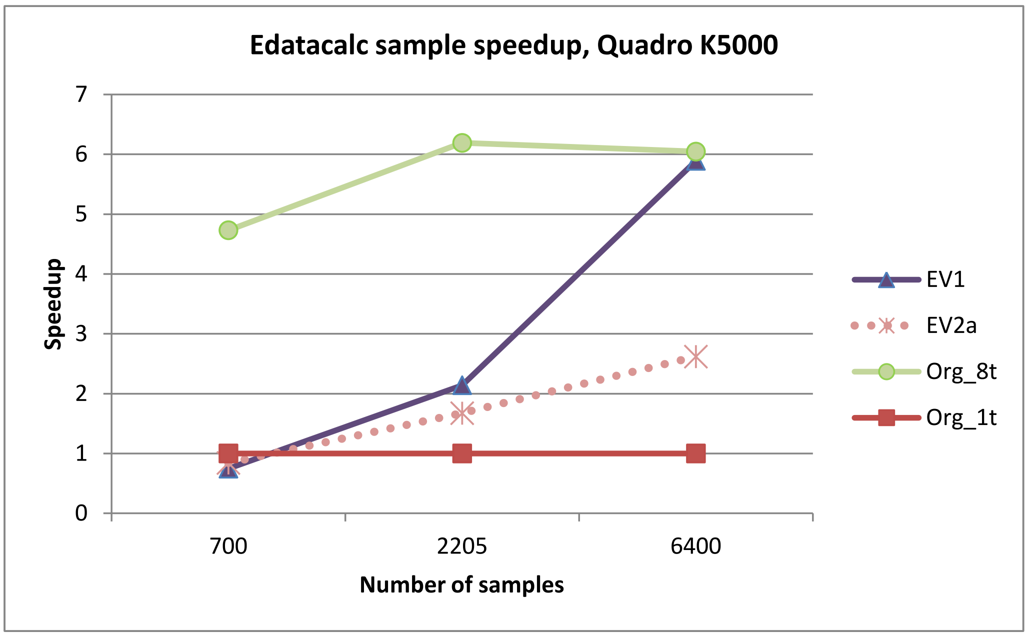

Figure 6.5: Sample execution time speedup on Quadro K5000 for each Edatacalc version compared to the original function executed with 1 thread, Org_1t.

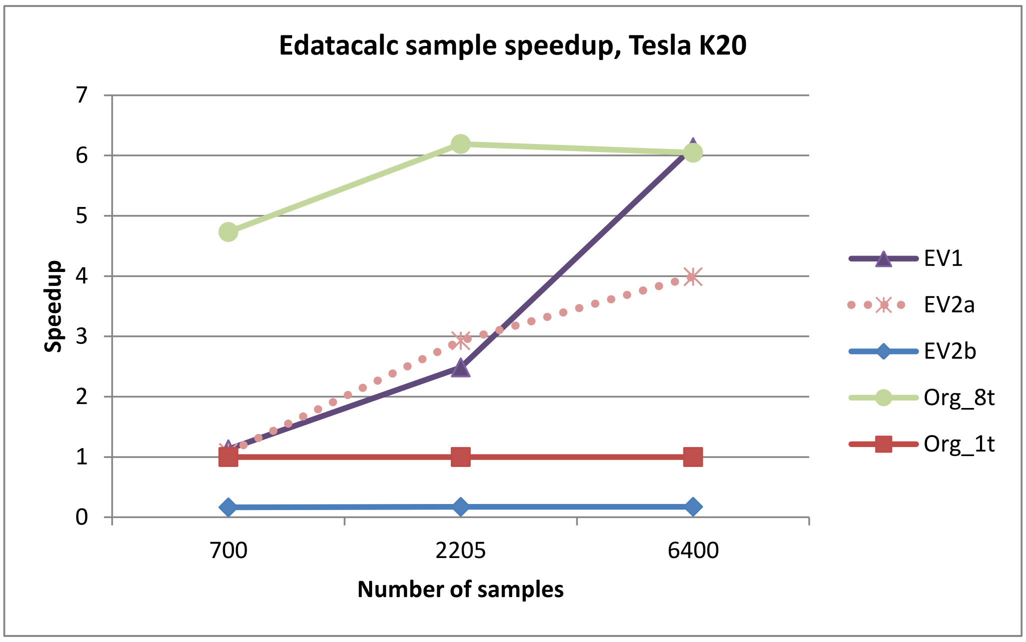

Figure 6.6: Sample execution time speedup on Tesla K20 for each Edatacalc version compared to the original function executed with 1 thread, Org_1t.

6.2.2 Channel speedup

The best channel speedup is given by the original Edatacalc function executed with 8 threads, Org_8t.

Even though EV2b parallelizes the inner most for-loop that loops through the channels, it gives a worse speedup than the other versions. The reason for this is because the more channels that are used, the more duplicated data between the threads, and this results in more data in local memory and thereby more global memory communication. So even though more is parallelized, this cannot hide the global memory latency (which increases with more channels being used).

EV1 and EV2a all have each thread looping through all channels, and therefore will the version with lowest global memory latency have the best channel speed-down. This results in that EV1 has the second best channel speedup.

If the number of channels is increased to more than 96 channels, then will the EV1 most likely give better speedup than the original function executed with 8 threads. Also EV2a will most likely give better speedup if the number of channels is increased.

Figure 6.7 and Figure 6.8 shows the channel execution time speedup on Quadro K5000 and on Tesla K20 respectively for each Edatacalc version compared to the original function executed with 1 thread, Org_1t.

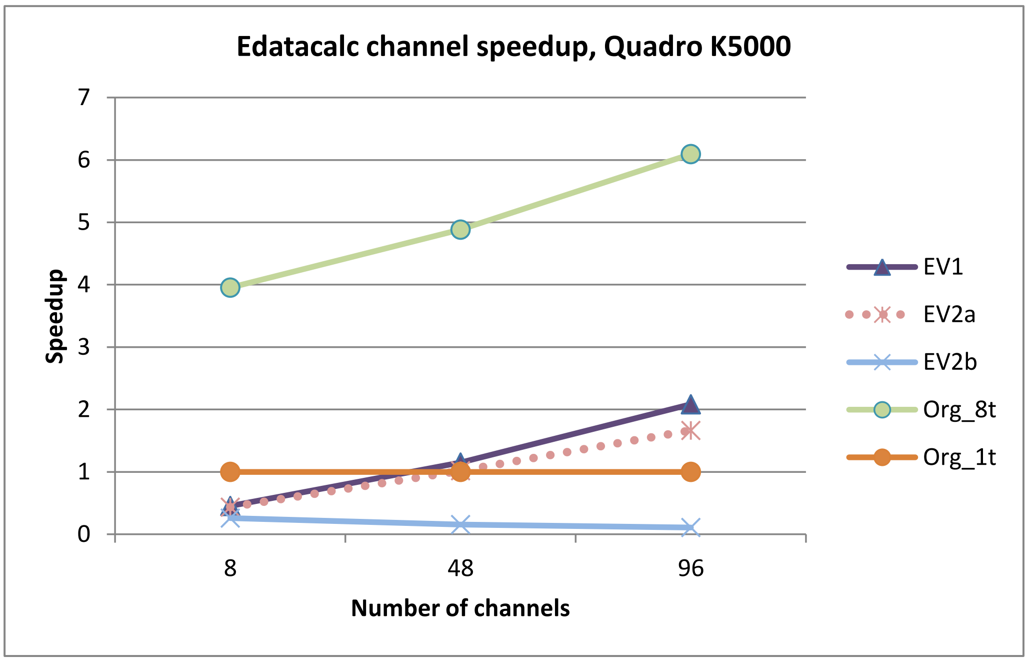

Figure 6.7: Channel execution time speedup on Quadro K5000 for each Edatacalc version compared to the original function executed with 1 thread, Org_1t.

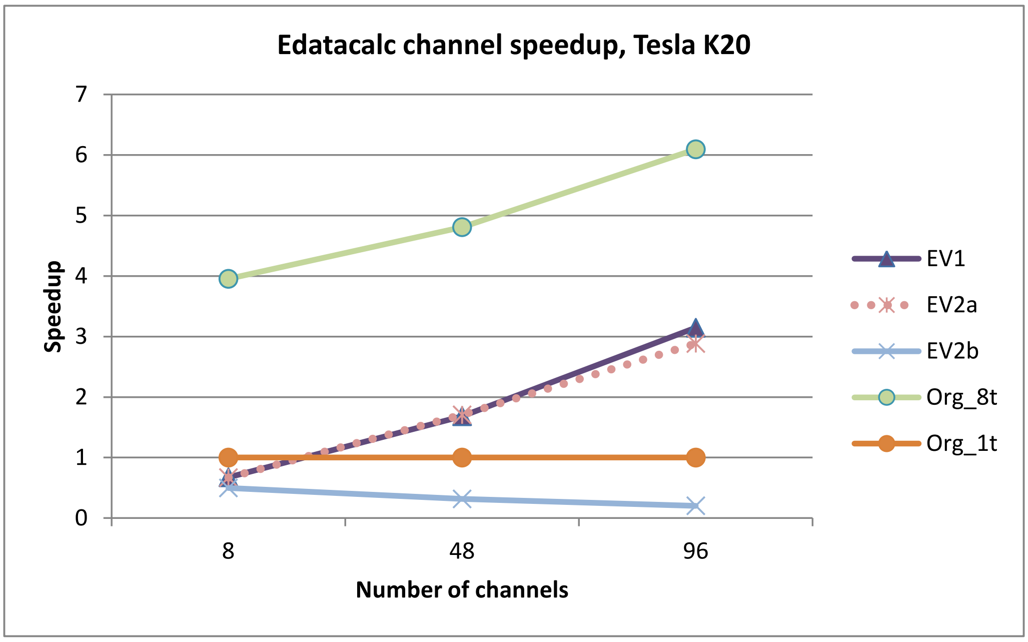

Figure 6.8: Channel execution time speedup on Tesla K20 for each Edatacalc version compared to the original function executed with 1 thread, Org_1t.

6.2.3 Source speedup

The best source speedup, when 10 or more sources are being used, is EV2a.

Since EV2a can have one thread per source for-loop iteration, and thus parallelize that for-loop, it gives better source speedup than the EV1 and the original version of the Edatacalc function executed with 8 threads, Org_8t. This is particularly clear when the maximum number of sources is 100 sources or more.

EV2b, which also can have one thread per source for-loop iteration, could not be tested with a maximum of 100 sources because the global memory cannot handle the amount of data that a source maximum at 100 sources would result in. The speedup should in theory though be almost the same as for EV2a because both of them can have one thread per source, but this is not the case since EV2b communicates to much with the global memory. EV2b speedup will decrease as the number of sources increases.

Figure 6.9 and Figure 6.10 shows the source execution time speedup on Quadro K5000 and on Tesla K20 respectively for each Edatacalc version compared to the original function executed with 1 thread, Org_1t.

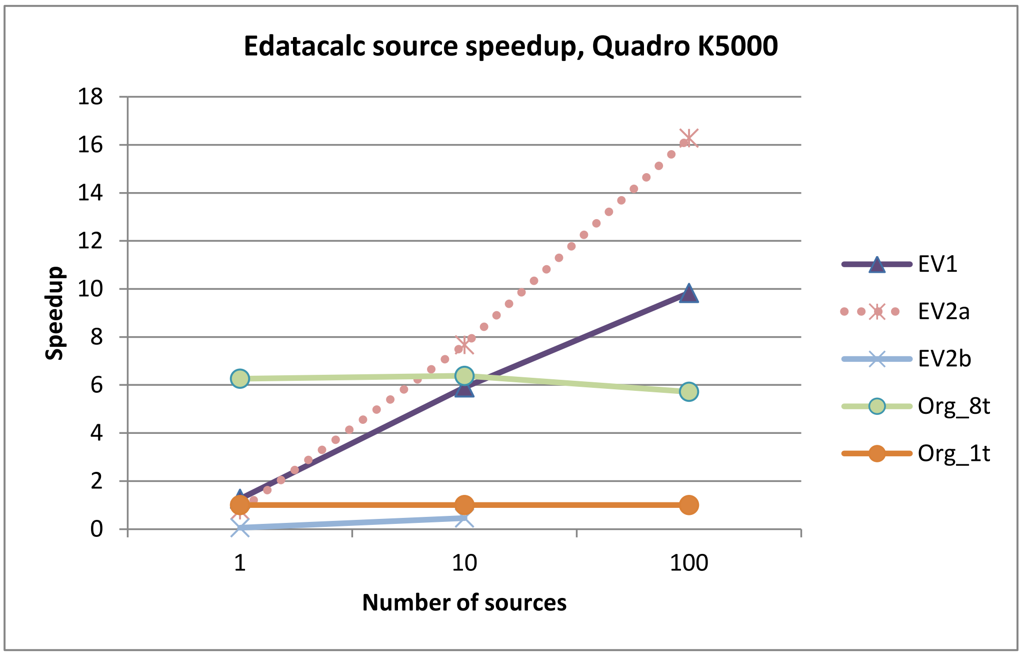

Figure 6.9: Source execution time speedup on Quadro K5000 for each Edatacalc version compared to the original function executed with 1 thread, Org_1t.

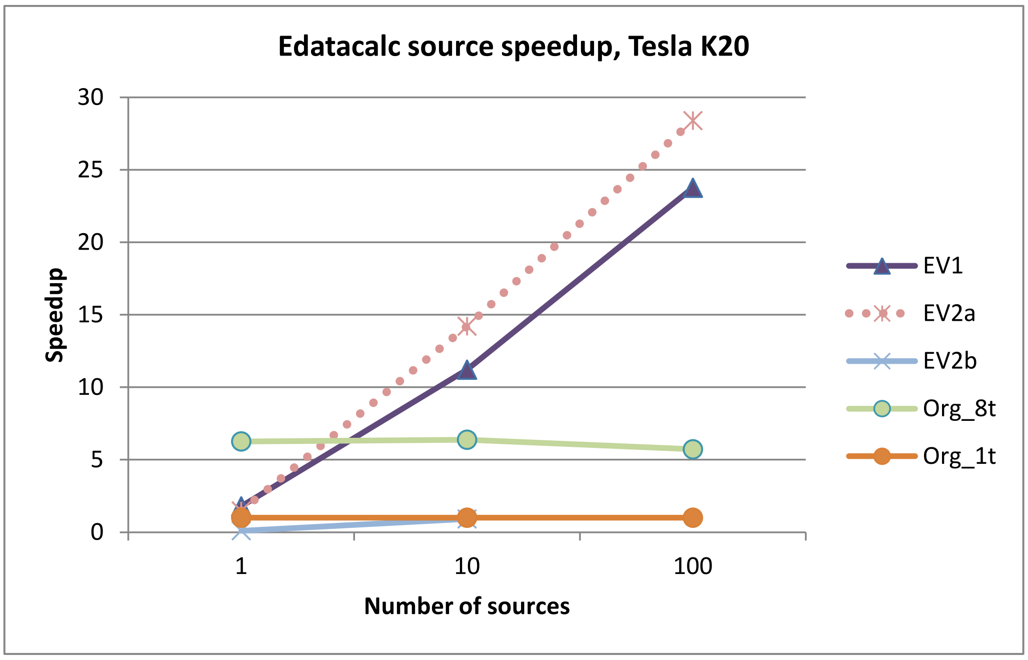

Figure 6.10: Source execution time speedup on Tesla K20 for each Edatacalc version compared to the original function executed with 1 thread, Org_1t.

6.2.4 Discussion

When collecting data for the sample and channel speedups for each version, only two sources were used when executing the functions. Since a lot of data has to be transferred to the GPU each invocation of the kernel functions, the problem size does not become large enough to hide the time for copying. This is why the sample and channel speedup for the CUDA versions are not that good and why the original function executed with 8 threads are better in those tests. If the number of sources increases, the speedup for the CUDA versions increases (see Section 6.2.3).

The difference in the speedup at 8 to 96 channels is almost the same for the original function executed with 8 threads, Org_8t, EV1, and version EV2a. However, the difference in sample speedup from 700 to 6400 samples is much higher for EV1 compared to the original function. Also, the difference in source speedup from 1 to 10 sources is much higher for EV2a compared to Org_8t. Since it is more likely to increase the number of samples and the number of sources for more advanced simulations, the CUDA versions will most likely give better performance than the original function if the problem size increases.

6.3 Discussion: Threads per block

In most cases, higher occupancy is achieved with a large number of threads per block. High occupancy have a positive effect on the execution time because that means that when some threads stall for data access from the cache/global memory, other threads can continue to work. In other words, the SM is active most of the time because the threads do not have to wait for other threads before they can get data from the cache/global memory. To achieve good occupancy, the number of threads should be a multiple of 32, which is the warp size. If it is not a multiple, the warps do not become fully utilized, the occupancy decreases and the execution time increases. For CUDA_NV3, CUDA_NV4, and SKEPU_CUDA_NV2, also the number of samples per thread (number of iterations per thread) affects the occupancy.

Unfortunately the shared memory limits the number of threads per block in CUDA_NV3. Only a small number of threads per block can be used and therefore it is not possible to achieve high occupancy per SM. It is because of this reason CUDA_NV4 can achieve lower execution times than the third version. If a large number of threads per block is used in CUDA_NV3, then there will also be more conflicts toward the shared memory. If there are a lot of conflicts, threads have to wait to get data from the shared memory which decreases the number of request per second to the shared memory, and this result in increasing execution time.

The L1 cache hit rate is higher when more than 15 and less than 1024 threads per block are used. High hit rate indicates that the data exists in the cache most of the time when the thread needs it (there is no need to get data from the global memory, which is a slow process compared to getting data from the L1 cache). The L1 cache (which is closest to the SM) can be overflowed with data when a large number of threads per block is used and each thread has a lot of data to process. With a lot of threads per block, data will be swapped in and out from the L1 cache more often than if there are few threads per block. This will result in lower L1 cache hit rates and more transactions per requests to the global memory, and since getting data from global memory is time consuming, this increases the functions execution time. The execution time does not have to increase that much when a lot of threads per block are used. If the L2 hit rate is quite high there will be less time spent communicating with the global memory. If both the L1 and L2 cache hit rate is low, the execution time could increase much more. Also, if shared memory is used, there is less need to get data from the global memory and this decreases the number of request per second to the L1 cache and it increases the L1 cache hit rate compared to when shared memory is not used.

A low execution time depends on both the number of samples per thread and the number of threads per block in CUDA_NV3 and CUDA_NV4. It has to be enough threads per block to occupy the SMs, each thread has to perform a balanced amount of work in such a way that they do not have to wait for each other to acquire computational units to perform the calculations, and there shall be few swaps to/from the L1 cache. A large number of threads per block is preferred to achieve high occupancy per SM. However, this only applies if the number of samples per thread are low enough to prevent L1 cache misses, achieve a high number of requests per second to the L1 cache, and low enough to achieve high occupancy per SM.

If there are too many threads per block and too many samples per thread then the *data request stall reason* will increase. An ineligible warp will increase the number of data request stalls if a request cannot currently be made as the required resources needed to perform the calculation are not available, or are fully utilized, or too many operations of that type are already outstanding. These problems can be avoided by either decreasing the number of threads per block or the number of samples per thread.

A small number of threads per block and a small number of samples per thread will decrease the occupancy per SM and thereby increase the execution time. There is a limit on how many samples per thread gives good performance. A high number of samples per thread may decrease the data request stall reasons, but the total parallelism will decrease since fewer threads have to perform more work and this increases the functions execution time.

The fastest execution time in EV2b of the Edatacalc function is much slower than the fastest execution in the other versions of the Edatacalc function. This is because a lot of data will be duplicated between the threads in this version (since the for-loops are gone) and this fills up the registers. With a lot of threads per block, the registers in a SM will not be enough to hold all data and therefore some data will be put in the local memory (which is placed in the global memory). Since a lot of calculation data is placed in the local memory, the L1 and L2 caches will overflow more easily than in the other versions of the function. Therefore more communication with the global memory will occur. This increases the execution time. The execution time will increase even more when the number of threads per block is increased because then more of the calculation data for each thread will be placed in the local memory. For 64 threads per block and more, the occupancy will be high, but this will not help to lower the execution time because of the large amount of communication with the global memory.

Chapter 7

Related work

A paper by Vinod et al. [27] describes nearly the same noise generation algorithms as those that are being used by the functions in this master thesis. They do not port their algorithms for GPU processing, but they discuss topologies for parallel implementations. They suggest, among others, one design where one processor should be allocated for each channel. This is what the first versions of the CUDA, OpenCL and SkePU implementations do in this master thesis.

Not many other articles or papers have been written about the problem handled by this master thesis. There are however more articles and papers about algorithm optimizations and performance evaluations on GPUs using CUDA.

A paper by Ryoo et al. [24] discusses different optimizations principles and how to evaluate an application's performance on a multithreaded GPU using CUDA. Their way of studying an application, optimizing it, and evaluating its performance has served as guidelines in this master thesis. They also discuss the challenge of balancing the threads' resources and the number of simultaneously active threads. This is an issue which also we had to deal with.

Another paper that studies performance of general-purpose applications on GPUs using CUDA is [1], by Che et al. They explore how the performance can be improved by porting a number of applications for the GPU, and they also describes some coding idioms which can improve the performance of their applications. Their study shows that the use of shared memory increases the performance since the communication with global memory decreases, especially when threads need to share data amongst each other. However, they also mention the limitations of the shared memory and that a large amount of threads per block may fill up the shared memory. In their paper as well as in this master thesis, a lower number of threads per block must occasionally be used to avoid filling up the shared memory. They also conclude that it is not always the maximum number of threads per block which results in the best performance.

Porting of complex scientific applications written in FORTRAN to CUDA have been done by Delgado et al. [5]. Their methodology of porting and performance improvement is almost the same as in this master thesis, even though their original application is written in FORTRAN and the original application in this master thesis is written in C++. Their ported application spent more time transferring data between the host and the device memories than the kernel took to be executed. This is a problem that also occurred with the porting of the Edatacalc function with CUDA, more than half of the total execution time was spent copying data to the device.

CUDA was the primary parallel computing platform used when porting the algorithms, and OpenCL was the secondary. It is therefore interesting to compare the performance of these two. A paper by Dua et al. [6] evaluates OpenCL as a programming tool for developing performance-portable applications for GPGPU. The algorithms used in that project focuses on dense matrix routines for numerical linear algebra, which are quite different from the original algorithms in this master thesis. Their conclusion is that OpenCL is a good choice for porting their algorithm for GPU processing, even though CUDA gives slighly better performance. This master thesis also concludes that OpenCL is a good choice, and it sometimes even gives better performance than CUDA.

Another paper in which CUDA and OpenCL are being compared is [10] by Karimi et al. They show that converting a CUDA kernel to an OpenCL kernel involves minimal modification but it requires more modifications to set up the OpenCL kernel. Also in their work, CUDA results in slightly better performance than OpenCL.

Chapter 8

Discussion and conclusion

The Tesla K20 card is better at double precision calculations than the Quadro K5000 card, and this is the reason for why the execution time of the functions are faster on the Tesla K20 card. It also has a larger global memory bandwidth and larger L2 cache. The OpenCL versions resulted in better performance on the Quadro K5000 card than the CUDA versions. This might have had something to do with the card itself, and it is possible that the OpenCL versions would perform better on the Tesla K20 card than the CUDA versions. Unfortunately we faced problems when using OpenCL with the Tesla K20 card and we could therefore not compare OpenCL and CUDA on that GPU.

To write the CUDA code was more simple than to write the code of the other frameworks. Not many lines of code were needed for setting up the GPU for kernel execution. It took however more work to decide the number of threads per block and the number of blocks, especially when shared memory is used.

OpenCL required a lot more code to set up the GPU for kernel execution than both CUDA and SkePU. On the other hand, there is no need to set number of threads per block or to wonder if the number of threads per block together with the number of blocks would yield enough threads (which is a problem when setting up the GPU for CUDA).

Not much SkePU code was needed for setting up the GPU for kernel execution, a little bit more than for CUDA and a little bit less than for OpenCL. However, since SkePU does not have that many user function generators a new macro had to be written in order to be able to port the Noisegen function.

One typical issue that is dealt with when optimizing an algorithm is to ensure that global memory accesses are coalesced. Coalesced memory accesses cannot be achieved in neither the Noisegen nor the Edatacalc function when writing to the result arrays. Each thread writes to non-consecutive indexes, and also for one thread in two consecutive for-loop iterations the writes are to non-consecutive indexes. The access pattern of the result arrays cannot be changed without changing the entire algorithm.

8.1 Noisegen

For all problem sizes larger than 700 samples and 48 channels, the author would recommend using the second OpenCL Noisegen version (OPENCL_NV2) or the fourth CUDA version (CUDA_NV4) depending on which GPU the application shall use. OPENCL_NV2 provides the best sample and channel speedup on the Quadro K5000 card compared to the original Noisegen function executed with one thread. Since it is an OpenCL implementation, it also provides platform portability. On other NVIDIA cards however, the compilation of the OpenCL version might give less optimal PTX assembly code than the CUDA versions. CUDA_NV4 is also a strong candidate since its performance is only slightly lower than the OpenCL version on the Quadro K5000. On Tesla K20 it even gives better sample speedup than when the second OpenCL version is executed on Quadro K5000.

Another reason for choosing OPENCL_NV2 rather than CUDA_NV4 is because there is no need for the programmer to hard code the number of threads per block. The number of threads per block increases or decreases automatically when the problem size changes. This is not the case in the CUDA versions where the total number of threads might be too many or too few depending on the hard coded number of threads per block and the problem size.

On the other hand, CUDA_NV4 can perform better if choosing the number of threads per block would be an option in the GUI when running the application. This way, there would be less chance of having too many or too few threads. Also, the user can optimize the performance since the maximum number of threads per block does not always result in the best performance. This is a downside with the OpenCL version. Due to lack of time, see Section 1.2, there is currently no possibility to change the number of threads per block, OpenCL will choose the maximum amount of threads per block, even though fewer might give better performance.

This master thesis has not implemented Noisegen versions for SkePU with OpenCL backend, therefore it is hard to tell which one of OpenCL and SkePU would be the best choice on GPUs from other vendors than NVIDIA. If a SkePU version with OpenCL backend would perform as good as with CUDA backend on a NVIDIA GPU, then the best choice would still be to choose the OpenCL version.

Unfortunately, lazy memory copying could not be exploited in the SkePU versions since the CPU must modify the data between the function invocations. If it could be utilized, then the SkePU functions might have resulted in a speedup closer to the CUDA and the OpenCL speedups.

One downside with all Noisegen version is that the memory accesses are not coalesced for reading and writing the calculation data. This is because each thread accesses one `noisestate struct` element each from the array consisting of each threads calculation data. The `noisestate struct` element for index 0 in the array is placed first in the global memory, then on the following memory addresses, the struct element for index 1 is placed, an so on. If then each thread accesses for example the first variable in its struct element, the accesses of consecutive threads will not be to consecutive addresses and the memory access are therefore not coalesced.

8.2 Edatacalc

For small problem sizes, the author would recommend using EV1. With small problems, EV1 results in lower execution time than the other versions, and it results in better sample and channel speedup. The author would recommend EV2a of the Edatacalc function for larger problem sizes (more than 2205 samples per channel and more than 10 sources). EV2a gives faster execution time and speedup when large number of sources are used, and the difference in sample and channel speedup are not that large when comparing to EV1.

One drawback is that it is not possible to decrease the amount of data that has to be copied to GPU global memory in the Edatacalc function. This is ashame because the copy takes about as long as it takes the original function with 8 threads to execute the function (for problem sizes smaller or equal to 96 channels and 2205 samples).

The original Edatacalc function was already optimized for CPU with the use of threads and SSE Intrinsics functions. Because of this, the original function executed with eight threads got better speedup for small problem sizes than the CUDA versions of the Edatacalc function. Also important to point out is that in the future, in order to simulate more advanced simulations, the number of sources should go towards infinity and EV2a is much more suitable to handle that large amount of sources. Also, more advanced simulations may require a larger amount of samples and then the first version might give better speedup than the original.

In EV2b, it is not possible to assign one thread to each iteration if the maximum number of samples, channels and sources are used, because the GPU cannot provide that many threads. This problem might appear in the other versions as well if the maximum problem size increases in the future. If there are not enough threads to process one for-loop iteration each, then some threads have to rerun the kernel. This will increase the execution time and decrease the speedup.

Chapter 9

Future work

If the CUDA versions shall be used in the future then it should be possible to change the CUDA kernel properties (number of threads per block and blocks) online, while the application is executed. Either the user should be able to change the parameters in the GUI, or there shall be a formula in the code which calculates the number of threads per block and number of blocks depending on the problem size that has been set in the GUI. It should also be possible to change the number of samples per thread in the GUI for the concerned versions.

In the future, solutions for making the memory accesses coalesced in both the Noisegen and the Edatacalc functions should be investigated. As for now, writing to the result array are not coalesced, but this could perhaps be solved if the algorithms were rewritten. Also, there might be a solution for making the memory accesses for the calculation data in the Noisegen function more coalesced. By rearrange the `noisestate struct` and the array which contains the calculation data for each thread so that the first variable in the `noisestate struct` for each thread are first in the array, then comes each threads second variable, and so on, the memory accesses could be more coalesced. There might be other ways to restructure the algorithms as well to make them more suited for GPU processing. This is also something that could be investigated in future work.

A new user function generator macro had to be written for the Noisegen SkePU versions. However, the need for more read/write input/output operands is a restriction that could be much harder to resolve for the Edatacalc function. This because the Edatacalc function reads from and writes to more parameters than the Noisegen function.

In the future it can also be interesting to test the OpenCL and SkePU versions on GPUs from other vendors than NVIDIA.

Bibliography

[1] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer and Kevin Skadron, *A performance study of general-purpose applications on graphics processors using CUDA*, 2008, Academic Press, Inc., ISSN: 0743-7315, Journal of Parallel and Distributed Computing, Volume 68, Pages 1370 - 1380, available at `http://www.cs.virginia.edu/~skadron/Papers/cuda_jpdc08.pdf`

[2] M. Cole, *Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming*, March 2004, Elsevier Science Publishers B. V. Amsterdam, The Netherlands, ISSN: 0167-8191, Parallel Computing, Volume 30, Issue 3, Pages 389 - 406.

[3] U. Dastgeer, *Skeleton Programming for Heterogeneous GPU-based Systems*, 2011, ISBN 978-91-7393-066-6, Licentiate thesis, Nov. 2011, Linköping University.

[4] U. Dastgeer, J. Enmyren and C. Kessler, *SkePU Skeleton Programming Framework for Multicore CPU and Multi-GPU Systems*, `http://www.ida.liu.se/~chrke/skepu/`, accessed 25/04/2013, 10:16.

[5] Javier Delgado, João Gazolla, Esteban Clua and S. Masoud Sadjadi, *An Incremental Approach to Porting Complex Scientific Applications to GPU/CUDA*, July 2010, In Proceedings of the IV Brazilian E-Science Workshop, XXX Brazilian Computer Science Conference, Belo Horizonte, Brazil, available at `http://users.cis.fiu.edu/~sadjadi/Publications/EScience2010.pdf`

[6] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson and J. Dongarra, *From CUDA to OpenCL: Towards a Performance-portable Solution for Multi-platform GPU Programming*, 2012, Elsevier Science Publishers B. V., ISSN 0167-8191, Parallel Comput., Volume 38, Pages 391 - 407, available at `http://www.netlib.org/lapack/lawnspdf/lawn228.pdf`.

[7] Eclipse Foundation, *Eclipse*, `http://www.eclipse.org/`, accessed 29/05/2013, 21:40.

[8] J. Enmyren and C. Kessler, *SkePU: A Multi-backend Skeleton Programming Library for Multi-GPU systems*, Sep 2010, ACM New York, USA, ISBN 978-1-4503-0254-8, HLPP '10: Proceedings of the fourth international workshop on High-level parallel programming and applications, Pages 5-14.

[9] Intel, *Intel® C++ Intrinsic Reference*, Document Number: 312482-003US, available at `http://software.intel.com/sites/default/files/m/9/4/c/8/e/18072-347603.pdf`, accessed 11/03/2013, 17:10.

[10] Kamran Karimi, Neil G. Dickson and Firas Hamze, *A Performance Comparison of CUDA and OpenCL*, 2010, CoRR, Volume abs/1005.2581, available at `http://arxiv.org/ftp/arxiv/papers/1005/1005.2581.pdf`

[11] C. Kessler and J. Keller, *Models for parallel computing: Review and Perspectives*, Dec 2007, Gesellschaft für Informatik e.V., Germany, ISSN 0177-0454, available at `http://www.ida.liu.se/~chrke/papers/modelsurvey.pdf`

[12] Khronos Group, *OpenCL - The open standard for parallel programming of heterogeneous systems*, `http://www.khronos.org/opencl/`, accessed 05/05/2013, 12:20.

[13] David B. Kirk and Michael Garland, *Understanding throughput-oriented architecture*, Nov 2010, Communications of the ACM, Magazine, Volume 53, Issue 11, Pages 58-66, available at `http://dl.acm.org/citation.cfm?id=1839694&bnc=1`, accessed 22/01/2013, 13:45.

[14] David B. Kirk and Wen-Mei W. Hwu, *Programming massively parallel processors : a hands-on approach*, 2006, Morgan Kaufmann Publishers Inc.

[15] Microsoft, *Visual Studio*, `http://www.microsoft.com/visualstudio/eng/visual-studio-update`, accessed 29/05/2013, 21:30

[16] NVIDIA Coporation, *CUDA 5*, `http://www.nvidia.com/object/cuda_home_new.html`, accessed 05/05/2013, 20:05.

[17] NVIDIA Corporation, *CUDA C programming guide*, `http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`, accessed 18/01/2013, 10:40.

[18] NVIDIA Corporation, *CUDA C best practices guide*, `http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html`, accessed 18/01/2013, 10:50.

[19] NVIDIA Corporation, *NVIDIA Quadro K5000*, `http://www.nvidia.com/object/quadro-k5000.html`, accessed 09/05/2013, 09:35.

[20] NVIDIA Corporation, *Tesla Kepler*, `http://www.nvidia.com/content/tesla/pdf/Tesla-KSeries-Overview-LR.pdf`, accessed 09/05/2013, 09:35.

[21] NVIDIA Corporation, *Tuning CUDA applications for Kepler*, `http://docs.nvidia.com/cuda/kepler-tuning-guide/index.html`, accessed 22/01/2013, 15:55.

[22] NVIDIA Corporation, *What is GPU-computing*, `http://www.nvidia.com/object/what-is-gpu-computing.html`, accessed 18/01/2013, 10:30.

[23] O. Rosenberg, Khronos Group, *OpenCL Overview*, `http://www.khronos.org/assets/uploads/developers/library/overview/opencl-overview.pdf`, accessed 05/05/2013, 12:20.

[24] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk and Wen-mei W. Hwu, *Optimization principles and application performance evaluation of a multithreaded GPU using CUDA*, 2008, ACM, ISBN: 978-1-59593-795-7, Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, Pages 73-82, available at `http://impact.crhc.illinois.edu/shared/papers/optimization2008.pdf`

[25] J. v.d. Sanden, *Evaluating the Performance and Portability of OpenCL*, August 2011, Master thesis, available at `http://parse.ele.tue.nl/system/attachments/20/original/Evaluating%20the%20Performance%20and%20Portability%20of%20OpenCL.pdf?1314101805`

[26] D. Schaa and B. Jang, *Programming with CUDA and OpenCL*, `http://www.ece.neu.edu/~dschaa/files/cuda_opencl_hms.pdf`, accessed 05/02/2013, 12:30.

[27] G. Vinod, K. K. Roshni, R. Abraham and G. Varkey, *Real-Time Multi-Channel Simulation of Sonar Signals by Spectral Shaping of White Noise Using an Array of Digital Signal Processors*, 2004, ISBN: 0-7803-8541-1, Underwater Technology, 2004. UT '04. 2004 International Symposium on, Pages 187-191, available at `http://www.cdactvm.in/Real-Time%20Multi-Channel%20Simulation%20of%20Sonar%20Signals%20by%20Spect.pdf`

Linköpings universitet

Avdelning, Institution
Division, Department

IDA,
Dept. of Computer and Information Science
581 83 Linköping

Datum
Date

June 2013

Språk Language	**Rapporttyp** Report category	**ISBN** -
☐ Svenska/Swedish	☐ Licentiatavhandling	**ISRN** LIU-IDA/LITH-EX-A–13/029–SE
☒ Engelska/English	☒ Examensarbete	**Serietitel och serienummer** Title of series, numbering
	☐ C-uppsats	**ISSN** -
	☐ D-uppsats	Linköping Studies in Science and Technology
☐ ______________	☐ övrig rapport	Thesis No. A–13/029–SE
	☐ ______________	

URL för elektronisk version

http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-94023

Titel
Title

Adaptation of algorithms for underwater sonar data processing to GPU-based systems

Författare
Author

Patricia Sundin

Sammanfattning
Abstract

In this master thesis, algorithms for acoustic simulations in underwater environments are ported for GPU processing. The GPU parallel computing platforms used are CUDA, OpenCL and SkePU. The purpose of this master thesis is to adapt and evaluate the ported algorithms' performance on two modern NVIDIA GPUs, Tesla K20 and Quadro K5000.

Several optimizations, described in existing literature for GPU processing (e.g. usage of shared memory, coalesced memory accesses), are implemented and multiple versions of each algorithm are created to study their trade-offs.

Evaluation on two GPUs showed that different versions of the same algorithm have different performance characteristic and execution with the best performing version can give better performance than the original algorithm executing on 8 CPUs. A performance comparison between CUDA, OpenCL and SkePU versions of one algorithm is also made.

Nyckelord
Keywords
GPU processing, CUDA, OpenCL, SkePU, Performance optimization, Underwater sonar data processing