Performance Evaluation of a Signal Processing Algorithm with General-Purpose Computing on a Graphics Processing Unit

FILIP APPELGREN

MÅNS EKELUND
Performance Evaluation of a Signal Processing Algorithm with General-Purpose Computing on a Graphics Processing Unit

Filip Appelgren and Måns Ekelund

June 5, 2019
Abstract

Graphics Processing Units (GPU) are increasingly being used for general-purpose programming, instead of their traditional graphical tasks. This is because of their raw computational power, which in some cases give them an advantage over the traditionally used Central Processing Unit (CPU). This thesis therefore sets out to identify the performance of a GPU in a correlation algorithm, and what parameters have the greatest effect on GPU performance. The method used for determining performance was quantitative, utilizing a clock library in C++ to measure performance of the algorithm as problem size increased. Initial problem size was set to $2^8$ and increased exponentially to $2^{21}$. The results show that smaller sample sizes perform better on the serial CPU implementation but that the parallel GPU implementations start outperforming the CPU between problem sizes of $2^9$ and $2^{10}$. It became apparent that GPU’s benefit from larger problem sizes, mainly because of the memory overhead costs involved with allocating and transferring data. Further, the algorithm that is under evaluation is not suited for a parallelized implementation due to a high amount of branching. Logic can lead to warp divergence, which can drastically lower performance. Keeping logic to a minimum and minimizing the number of memory transfers are vital in order to reach high performance with a GPU.

Keywords: Parallelization, GPU, CUDA, RADAR, Optimization
Sammanfattning


**Keywords:** Parallellisering, GPU, CUDA, RADAR, Optimering
We would first like to thank SAAB and our manager Peter Sundström for the opportunity to conduct this interesting thesis at their offices. Further gratitude is extended to our supervisor Daniel Fransson, who helped us extensively during the entire thesis. A final thank you is directed towards our examiner Fredrik Kilander at KTH, who continuously assisted in the composition and writing of this report.
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1.8 Execution of CUDA code</td>
<td>23</td>
</tr>
<tr>
<td>2.2 Application of the algorithm</td>
<td>23</td>
</tr>
<tr>
<td>2.3 Abbreviations</td>
<td>25</td>
</tr>
<tr>
<td>3 Method</td>
<td>27</td>
</tr>
<tr>
<td>3.1 APOD</td>
<td>27</td>
</tr>
<tr>
<td>3.2 Testing</td>
<td>28</td>
</tr>
<tr>
<td>3.2.1 Automation</td>
<td>28</td>
</tr>
<tr>
<td>3.2.2 Measurement tools</td>
<td>28</td>
</tr>
<tr>
<td>3.2.3 Significance</td>
<td>29</td>
</tr>
<tr>
<td>4 Work</td>
<td>31</td>
</tr>
<tr>
<td>4.1 Assess</td>
<td>31</td>
</tr>
<tr>
<td>4.2 Parallelize</td>
<td>31</td>
</tr>
<tr>
<td>4.3 Optimize</td>
<td>32</td>
</tr>
<tr>
<td>4.4 Deploy</td>
<td>34</td>
</tr>
<tr>
<td>5 Results</td>
<td>35</td>
</tr>
<tr>
<td>5.1 Comparison of serial and final parallel version</td>
<td>36</td>
</tr>
<tr>
<td>5.2 Comparison of parallel versions</td>
<td>37</td>
</tr>
<tr>
<td>6 Conclusions and future work</td>
<td>41</td>
</tr>
<tr>
<td>6.1 Conclusions</td>
<td>41</td>
</tr>
<tr>
<td>6.2 Future work</td>
<td>42</td>
</tr>
</tbody>
</table>
1 Introduction

In 1965, in the journal *Electronics*, Gordon Moore shared his observations about computing power, which came to be known as Moore's Law. What Moore had noticed was that computing power had been increasing twofold for the past 6 years, from 1959 - 1965. Based on this, he made the claim that it would probably remain that way for 10 more years, until 1975 [15]. That statement made by Moore in the 1960's has held up very well and is still relevant in 2019. There is, however, skepticism regarding the future of Moore's law and with ever decreasing transistor sizes, perhaps the prediction made by Moore over 50 years ago is reaching its end [16]. How then will we go about improving the performance of computers in the future?

A Central Processing Unit (CPU) is often referred to as the "brain" of the computer, and its job is to take input data, process it and transfer the result to memory or other devices [5]. This is the kind of chip with transistors Moore was talking about in his publication. Most computers today are also equipped with a Graphics Processing Unit (GPU), which traditionally has been used for drawing computer graphics. GPUs and CPUs look quite different from one another, since their tasks in a computer are not very similar. Drawing computer graphics is work which relies heavily on a large amount of mathematical operations. Usually, a programmer writes code which draws one pixel on the screen. The GPU then runs multiple instances of this code to create a picture on the monitor. This has led to GPUs evolving into devices that are extremely proficient at parallel computations [12]. With this massive, raw, computational power, GPUs have proven to be useful for
more general tasks than drawing graphics. To achieve a further increase in processing power, it is of relevance to understand under what circumstances GPUs can be used to improve the speed and performance of calculations which are traditionally executed on the CPU.

1.1 Background

In order to understand the thesis project’s problem and aim it is important to be familiar with following concepts.

1.1.1 Parallel Computing

Computer software has traditionally been created to compute a problem sequentially by breaking down the problem in a series of instructions and then executing them in order. Parallel computing is a type of computation where instructions can be executed simultaneously with the use of multiple computing resources [22].

During the 1960s parallel computing arose as a new field in computer science due to the increasing demand for high performance computing. The purpose of parallel computing is to achieve a speedup in latency or throughput relative to a sequential implementation [9].

In 1966 Michael J. Flynn presented four classifications of computer architectures [6], today the classifications are known as Flynn’s taxonomy, which is widely used in the field of computer architectures. The classifications are based on the number of instruction streams and data streams available to an
architecture. One of the classifications defined by Flynn is an architecture with a Single Instruction stream that operates on Multiple Data streams (SIMD) and is effective for vector operations such as graphics rendering.

1.1.2 Central Processing Unit

The CPU receives an instruction from memory, which it then decodes and executes. Once it has executed that instruction, any result is stored in another memory location [5]. Traditionally, a CPU consists of an Arithmetic Logic Unit (ALU), which does computations such as addition and subtraction. Further, the CPU has a control unit which handles instruction execution on the CPU as well as communication with other devices. Finally, the CPU has registers where calculated results and information to the ALU is stored.

1.1.3 Graphics Processing Unit

Even though GPUs and CPUs have different tasks, they still contain the same type of hardware [26]. CPUs can have multiple ALUs, but GPUs generally have a much higher count of ALUs than CPUs do. This allows the GPU to execute instructions on a high amount of ALUs at once, allowing for great SIMD performance in GPUs. It is worth mentioning though, that ALUs seen on GPUs are not as complex as the ones found on CPUs, neither do they run with as high of a clock frequency.
1.1.4 **Compute Unified Device Architecture**

Compute Unified Device Architecture (CUDA) is a parallel programming platform created by the graphics card manufacturer NVIDIA [4]. Modern computations are often very resource heavy, both in computation time and memory space, which leads to an increased demand for computing power. The CUDA programming model acts as a software layer to enable general purpose parallel programming with GPU resources. CUDA provides an Application Programming Interface (API) allowing developers to use the C programming language to write software that utilizes the GPU.

1.1.5 **Radio Detection and Ranging**

Radio Detection and Ranging (RADAR) was developed during the 20th century, and increasingly so during the second world war [7]. RADAR systems can be used to locate and track objects, through the use of electromagnetic energy. The systems send either continuous waves or quick bursts of electromagnetic pulses which bounce off of the targets, allowing a receiver to deduce their location and speed. Since RADAR signals are simply electromagnetic waves, those waves will have properties such as amplitude, frequency and pulse width. The attributes of these signals are relevant to this work since they form the input data of the algorithm that was parallelized.

1.2 **Problem**

In this thesis, an algorithm which stores and analyses RADAR signals has been studied. When the algorithm receives a new RADAR signal, it compares
the parameters of the new signal to other signals it has received before. Through the comparison, the algorithm decides if the newly received signal belongs to an emitter which has been seen earlier, or if this signal comes from an entirely new emitter.

What are the potential improvements to the algorithm’s execution time, when utilizing a GPU in combination with a CPU?

1.3 Purpose

The purpose of the report is to evaluate under what circumstances the utilization of a GPU together with a CPU, offers better performance than just a CPU. Further, the report investigates the possibilities of improving the performance of a correlation algorithm, with GPU programming. The correlation algorithm is supplied by SAAB AB and is being used in their RADAR systems.

1.4 Goal

The long term goal of this report is to allow for lower production costs of SAAB’s RADAR systems by replacing expensive hardware with a GPU.

1.5 Societal benefits, ethics, and sustainability

SAAB is a producer of high technology equipment in the defence and civil industry. The ethical aspect of working at a company operating in the military industry could be questioned in regards to human rights. To
ensure that SAAB meets the high ethical standards needed to operate in this business they have created a Code of Conduct (CoC) [2], where Trust and Responsibility are keywords. The CoC states that SAAB are actively working for zero tolerance against corruption and that they live up to the strict regulatory framework set up by the Swedish government. In addition to the CoC, SAAB also provide a CoC for suppliers which is based on the UN Global Compact’s ten principles [3] [25]. The purpose of the supplier CoC is to express what SAAB expects of a supplier in regards to Human rights, Labour, Environment and Anti-Corruption.

General-purpose computing on graphics processing units can be an intricate subject, information is often outdated or hard to find due to the rapid development of technology. Hopefully the work and the results presented in this thesis will be able to act as a foundation for other projects in this area.

The direct sustainability impacts of this work are mainly economical. If the outcome of this work is positive there are opportunities for SAAB to replace very expensive hardware with units utilizing GPUs, which would result in lower production costs.

1.6 Methodology

Determining aspects that affect performance of GPU programming has been partly handled through already conducted research, journals and other scientific literature. This is known as inductive reasoning, where a result or
understanding is reached through the study of already publicized data [10].

To draw specific conclusions about the SAAB algorithm, there was a greater focus on the performance of that particular algorithm, not so much on general GPU programming. To determine the performance of GPU programming in regards to the specific algorithm supplied by SAAB AB, multiple tests on a simplified algorithm were conducted. Tests on the original algorithm were never executed since the origin of some relevant parameters were inaccessible. Research done in this manner is often referred to as quantitive research [10]. Through quantitative research, a result is reached by conducting tests of some sort, through which data can be gathered with research instruments, and then presented in tables or charts [10]. A quantitative approach was taken to determine the difference in performance between the serial and parallel implementation of SAAB’s correlation algorithm. In this particular case improved performance was defined as a decrease in total execution time.

1.7 Employer

SAAB serves the global market with world-leading products, services and solutions from military defence to civil security. With operations on every continent, SAAB continuously develops, adapts and improves new technology to meet customers’ changing needs. SAAB has around 16,000 employees. Annual sales amount to around SEK 31 billion, of which they re-invest about 23 per cent in research and development. [1]
1.8 Disposition

In chapter 2, an extended background to concepts that are relevant to GPU programming are given. Further, CUDA and the application of the algorithm is explained in detail.

Chapter 3 describes how the work was handled, explaining the parallelization methodology, how test results were achieved, what tools were used and how the significance of the results was guaranteed.

In chapter 4, the actual work is described, showing how the algorithm was parallelized and optimized.

Chapter 5 presents results, showing comparison charts between the serialized version and the parallelized version, as well as charts showing how optimization to the parallel implementation led to improvements in execution time.

Finally, chapter 6 contains conclusions from the work as well as suggestions on how to improve the GPU results in the future.
2 Extended background

This section presents an extensive overview of the most important concepts to GPUs and general purpose GPU programming. Furthermore, it gives an explanation of the algorithm in question and its application.

2.1 GPU

The following subsections draw information from the book ”CUDA By Example”, published by NVIDIA [11].

2.1.1 Kernel

In regular C code, a programmer can write and execute a function, which is a group of statements that together execute a certain task. In just the same way, a CUDA programmer can write and execute a kernel. This fills the same role as the regular C function, but it is executed on a GPU instead of a CPU.

2.1.2 Threads

A kernel can be executed in parallel, meaning that the function executes many times simultaneously. This is done by the CUDA threads. The number of threads running a kernel is specified by the programmer in the kernel call. Every thread gets its own thread-ID during the execution of the kernel, meaning that the user can get information about every thread during runtime. This information can be useful during runtime since the programmer then can write thread-specific code.
2.1.3 Block

A CUDA block is a collection of threads. When executing a kernel, the programmer or user must specify not only thread count but also block count to get a total number of threads executing the kernel. If the user inputs 4 blocks running with 32 threads each, the total number of threads running the kernel is $4 \times 32 = 128$.

2.1.4 Grid

A collection of blocks is known as a grid. Each kernel has its own grid, or set of blocks and threads.

![Figure 1: Host to device](image)

Figure 1 shows how the host launches kernels to the device, where the grids, blocks and threads are scheduled for execution.
2.1.5 Streaming Multiprocessor

When a kernel is executed, the blocks and threads specified by the user, are distributed among Streaming Multiprocessors (SMs). An SM creates groups of 32 threads called warps. The multiprocessors have the capability of creating multiple warps, in order to execute and manage hundreds of threads concurrently. The layout of the SM can differ depending on the GPU, but generally it contains a certain number of ALUs which handle arithmetic computation, a warp scheduler and some cache memory. This type of multiprocessor uses an architecture called Single-Instruction Multiple-Thread (SIMT), which is of relevancy when discussing the next topic, warps.

2.1.6 Warp

In NVIDIA’s programming guide [18], a warp is described as a group of 32 threads that start at the same program address, but are free to branch off if they were to gain different results from the code. This is what enables the earlier mentioned SIMT architecture. The more classical SIMD architecture allows a programmer to execute a single instruction on multiple processing elements. However, in CUDA, the programmer has the ability to execute instructions which are individually linked to each thread in the execution. During execution of a kernel, the programmer has access to the individual threads, which means that the code can be executed differently depending on the thread. Nevertheless, just because the programmer has the ability to create different execution paths depending on the thread, he or she may not wish to do so. If threads within a warp take different paths through the execution, i.e. diverge somewhere, we end up with a warp divergence [13].
The issues with warp divergence stem from the way CUDA handles these branching threads. If half the threads were to diverge from the others, the GPU would have to put the remaining half on hold, whilst executing the ones that branched off. Once that execution has finished, the GPU then goes on to finish execution of the remaining threads. Once both halves have been executed, the threads can rejoin and continue their parallel execution. As shown in figure 2, this leads to a loss of speedup since parallelism is lost.

![Figure 2: Warp divergence](image)

### 2.1.7 Stream

NVIDIA’s programming guide describes a stream as a sequence of commands executed serially. This means that a programmer which launches two kernels after one another will have those two kernels executed in serial order on one stream. However, the programmer may wish to have multiple kernels running simultaneously for extra parallelism. By launching multiple streams, the programmer can schedule multiple sequences of commands on the GPU at the same time, thereby increasing parallelism.
2.1.8 Execution of CUDA code

CUDA is an extension of C/C++ code, which traditionally is executed on a CPU, thus indicating an important relationship between CPU and GPU during CUDA execution. Because of the way CUDA is designed, it is entirely possible to write CUDA code which runs solely on the CPU, with no GPU interaction. Code written this way is said to run only on the host, while code that runs on the GPU is said to run on the device. As stated above, functions which execute on the device are known as kernels, and only when these are called does the device spring into action.

2.2 Application of the algorithm

RADARs are used to detect objects such as airplanes, boats, cars and other vehicles [7]. Due to strategic reasons in both military and civil applications it is important for an object to know when it is hit by a RADAR signal. With the right hardware it is possible to identify and locate the emitting source (emitters) of the signal by analyzing relevant parameters; frequency, amplitude, direction, and pulse width.

The algorithm referred to correlates new incoming signals with previously identified emitters. If a new signal shows a lot of similarities with a previously registered emitter’s signals, an assumption can be made that they originate from the same emitter. Similarities are calculated through statistics, the algorithm stores and updates identified emitters based on every signal and then calculates the standard deviation and mean for all parameters.
Each parameter of the new signal is individually correlated with the parameter distribution of previously seen emitters. If the parameters matches with a seen emitter, the correlation score for that parameter is low. If it does not match well, the correlation score is high. The correlation scores for each parameter are then summarized to return a total correlation score for the signal. The emitter that returns the lowest total correlation score is most likely to also be the emitter of the new signal. The mean value and standard deviation are then updated for that emitter, based on the parameter values of the newly seen signal.

Additionally there are certain conditions to the calculated score and the emitters. If the lowest score is higher than a certain threshold the emitter is not considered to be the transmitting source, a new emitter-slot is then created for the new signal. Furthermore, if a new signal is expected to belong to an emitter that has not been seen for a longer period of time, the match can be rejected. The old emitter is then deleted and the new signal is given a new emitter-slot.
2.3 Abbreviations

A comprehensive list of concepts explained in previous chapters

- **CPU** - Central Processing Unit, the classic processor in a computer.

- **GPU** - Graphics Processing Unit, traditionally used for rendering computer graphics, used for general purpose tasks in this thesis.

- **CUDA** - Programming platform created by GPU manufacturer NVIDIA, allows for easy GPU programming.

- **Kernel** - Similar to a classic function written in C++ code, but executes on the GPU instead of the CPU.

- **Threads** - The workers executing in parallel on the GPU, the number of threads launched for the kernel is specified in the code.

- **Block** - A collection of threads, the number of blocks to launch is specified in the code.

- **Grid** - A collection of blocks, grids are used for the programmer to easily access specific threads during run time.

- **Streaming Multiprocessor** - General purpose processors in the GPU that schedule the threads for execution.

- **Warp** - Threads that are scheduled for execution are done so in groups of 32, which is called a warp.

- **Stream** - Streams allow for parallel execution of kernels.
3 Method

This section will present the methods used to parallelize the algorithm and evaluate the differences in performance between the serial and the parallel implementations of the algorithm. The code was written in the programming languages C++ and CUDA C++.

3.1 APOD

The conversion from a serial to a parallel implementation of the algorithm has been done in accordance with NVIDIA’s C best practices guide [17]. The purpose of the manual is to help developers achieve the best possible performance with established parallelization and optimization techniques. To ensure that the conversion is done in a systematic fashion the guide also provides a four step design cycle: Assess, Parallelize, Optimize, Deploy (APOD).

The first step is to assess the algorithm and identify areas that can be parallelized. In the second step the developer does the actual parallelization, a serial implementation might have to be re-factored so that a parallel version makes sense. When the second step is complete the current implementation can be optimized. Optimization can be done in many different ways, such as hiding data transfers, avoiding warp divergence, or fine tuning floating point operations. To get an indication of where to start the optimization work, NVIDIA suggest using the CUDA profiler.
3.2 Testing

The algorithm is used in a real-time system where data is collected continuously. The only performance metric identified as relevant for this thesis was time. Multiple tests were taken with different problem sizes. Problem sizes are defined as the maximum number of emitters that can be stored. As testing proceeded, a range of problem sizes was set, to assure that performance could be accurately judged as the workload changed. This range was determined to start at $2^8$ and exponentially increase to $2^{21}$. For each problem size 100 tests were completed.

3.2.1 Automation

To ensure repeatability and to reduce time spent testing, all tests were done using an automated setup. The setup consists of two scripts that parameterize both the serial and the parallel implementation of the algorithm. The common parameters were problem size and number of runs, additionally the parallel implementation had number of blocks and number of threads as parameters.

3.2.2 Measurement tools

Computers and operating systems have different approaches towards tracking CPU time \[24\], to avoid uncertainty regarding these differences all tests were conducted on the same hardware and with the same operating system.

The serial version of the algorithm was implemented with ordinary C++ code. With the release of C++ version 11 a library named Chrono is
available, the Chrono library provides a high resolution clock that can record precise measurements in microseconds. Chrono was used to measure both the serial version and the parallel version of the algorithm.

The CUDA Toolkit has a profiling tool called *nvprof* that enables developers to understand what happens in a CUDA application. Nvprof is command-line based tool that collects data of CUDA activities on both the host (CPU) and the device (GPU) [19]. The profiler collects data of, among other things, kernel execution, memory transfer and API calls. This tool was used during the optimization phase of the work.

### 3.2.3 Significance

The emitters used in all tests were based on random numbers. Identifying what type of probability distribution the data has was of relevance in order to assure that results were of significance and not due to chance. The most common continuous probability function is the normal distribution [8], therefore, testing for normality is a good place to start when determining the probability distribution. The Shapiro-Wilk test is one of the most well know normality tests [21] and will be the one used in this thesis. In addition to the Shapiro-Wilk test, results were visualized to assess what possible distribution the results could have.
4 Work

This section presents how the work progressed and the parallelized version was implemented, based on the APOD method.

4.1 Assess

The inherent nature of the correlation algorithm does not lend itself towards GPU parallelization due to the large amount of sequential logic the algorithm relies on. Further, the first problem sizes were quite small which led to GPU resources being left unused. However, the comparison between the newly received signals and the previously seen emitters was still assessed to be a parallelizable part of the code. This meant that all of the previously seen emitters now could be correlated with the new signal, all at once.

Since the new signal is correlated for each of its parameters separately, the correlation results for each parameter must be summarised for a total correlation score. This part of the code was also parallelized, meaning that all resulting correlation values for the new signal could be calculated simultaneously.

4.2 Parallelize

The two parts identified as hotspots for parallel work were the correlation and the summing of calculated scores. By refactoring the serial correlation into four kernels, one for each parameter, and the summing of scores into one kernel, the GPU could then be used in the execution of the algorithm.
When parallelizing an algorithm with a GPU one must consider the memory management on both the host (CPU) and the device (GPU). CUDA provides an array of different allocation and data transfer functions. A very modern one is \texttt{cudaMallocManaged} which creates a unified memory space where all processors, both CPU and GPU see a coherent memory image with a common address space \[20\]. The \texttt{cudaMallocManaged} function is widely used and recommended because of its simplicity. No thought has to go into where certain data is allocated or when data has to be transferred between host and device and vice versa, \texttt{cudaMallocManaged} just works \[14\]. Because of its simplicity, the first parallel version was implemented with \texttt{cudaMallocManaged}.

As explained, parallelization involves transferring data between the host and device. To ensure that no hiccups occurred in the parallelized version, the result of the parallel implementation had to be guaranteed. Therefore, instead of using randomized values, a test case which utilized set values was executed. With the use of set values in both the serial and the parallel implementation, the validity of the parallel version was guaranteed with the serial and the parallel algorithms returning the same value.

\section*{4.3 Optimize}

Following NVIDIA’s best practice guide the parallelized versions were profiled with the \texttt{nvprof} profiling tool. Profiling the algorithm gave indications that memory management with \texttt{cudaMallocManaged} took a large portion of the
total run time. CudaMallocManaged allocates space in a unified address space, meaning that it is an address space were both the host and the device can read from and write to. The function *cudaMalloc* instead allocates memory explicitly in the device memory. CudaMalloc only returns a chunk of memory with a size specified by the user, but does not actually populate it with any data in that memory space. To use the allocated memory, the user has to manually copy data over to the device location, from the host. Doing this manually requires extra work over using *cudaMallocManaged* but can lead to improved performance. A major factor in the speed improvements with cudaMalloc is due to unified memory introducing a need for page fault handling. The transfer speed for unified memory therefore ends up being about half of the speed offered by explicit memory allocation [23].

As the problem size increases, the algorithm can benefit from a higher concurrent thread count. Essentially, a higher thread count allows for higher utilization of the GPU’s computational capabilities. However, there are limitations to the number of threads that can be launched. The GPU used during this thesis has access to 28 streaming multiprocessors (SM), all of which can run 2048 simultaneous threads. Assuming each block carries 512 threads, one SM can support 4 blocks, 4 * 512 = 2048. Since the GPU has access to 28 SMs, the theoretical maximum number of blocks that can be used is 4 * 28 = 112. By launching more threads, the GPU utilization could therefore increase to match the problem size. The number of threads and blocks launched with each kernel was therefore dynamically increased with problem size. The maximum number of threads and blocks launched was
reached when problem size reached 65 536, or $2^{16}$. After that, the GPU was using all of its processing power, running 112 blocks with 512 threads per kernel.

To further improve performance, streams were introduced in the code. As previously explained in section 2.1.7, streams allow the user to run multiple sequences of kernels simultaneously. This meant that the four scoring kernels which were used for the four parameters could be run all at once.

4.4 Deploy

The algorithm used in this thesis is a simplified version of the actual algorithm used by SAAB, therefore the parallelized algorithm was not deployed in any production environment.
5 Results

In this section the results of this thesis are presented. The results are best presented in the form of graphs. To manage the large range of values some of the axes are logarithmically scaled. In section 5.1 the comparison of results between the serial version and the final parallel version are presented. In section 5.2 comparison of results between different parallel implementations are presented. The Shapiro-wilk test indicated that it its very probable that the results are normally distributed. To strengthen the case that the data is normally distributed one can look at the histogram in figure 3. Based on the indication that results are normally distributed and that results have a standard deviation of less than 5% of the mean, it is deemed that the results are of significance.

Figure 3: Histogram over result distribution
5.1 Comparison of serial and final parallel version

In figure 4, the performance of the serialized version and the final parallel version are compared. The focus lies on smaller problem sizes, ranging from the problem size of $2^8$ to $2^{14}$. From this graph it is evident that the CPU performs better at problem sizes smaller than $2^9$. At the smallest problem size of $2^8$ the parallel version is 120% slower than the serial version. However, at larger sizes the GPU outperforms the CPU by a growing margin as size increases. In this graph, both axes are represented logarithmically.
Figure 5 shows a comparison between the serial version and the final parallel implementation at larger problem sizes. Figure 5 maintains the logarithmic representation of size on the x-axis, but starts from $2^{16}$ instead of $2^8$. This way, the graph clearly displays the performance improvements achieved with a GPU at larger problem sizes.

### 5.2 Comparison of parallel versions

Figure 6 displays results between different parallel implementations. In figure 6, the x-axis is represented logarithmically while the y-axis is non-logarithmic. This graph is limited to problem sizes from $2^8$ to $2^{11}$ in order to give a clear understanding of performance differences at small problem sizes. The algorithm utilizing unified memory was roughly three times slower than
the versions utilizing explicit memory allocation and data transfer, at the smallest problem size.
Figure 7 displays the performance difference between the two different implementations with explicit memory allocation and data transfers. The version with streams offer slightly more performance at smaller sizes and performs, on average, about 12% better than the non-stream version at these sizes. The x-axis remains logarithmic while y-axis is non-logarithmic.
6 Conclusions and future work

The problem of the thesis was to understand how GPU parallelization could lead to improvements in the algorithm’s execution time. Both conclusions regarding the thesis project’s results and ideas for future work are presented below.

6.1 Conclusions

The parallel GPU implementation shows potential as size increases, and only performs worse than the CPU at problem sizes smaller than $2^9$. Involving the GPU in the execution requires extra memory allocation and data transfer which is not necessary during the CPU execution. Despite the GPU performing the actual calculation faster than the CPU, there is too much overhead cost from involving the GPU. A test run at size of 256 showed that the total overhead cost to utilize the GPU added up to about 70% of the measured execution time. These are mainly from memory transfer and memory allocation on the GPU.

Furthermore, the streams did not offer as much performance gain as initially expected. This is because of the overhead costs from the GPU. As mentioned, the actual kernels running on the GPU only represented about 30% of the actual execution time, at the smallest problem size. Therefore, by adding streams, all that was achieved was a slight speed-up in the actual execution, but no speed-up in the parts that took the largest amount of time, the memory allocation and data transfer. Because of this, the actual speed
increase was only a few per cent at smaller sizes. Then, what about larger sizes, when the actual calculation should represent a larger percentage of the execution time? Unfortunately, as problem size increased the streams stopped offering greater performance. This occurred when problem size reached roughly $2^{14}$. The reason for this is simple, running four kernels at once requires four times the resources one kernel does. This essentially means that the GPU is scheduling four times as many threads for execution. At problem sizes larger than $2^{14}$, the GPU runs out of physical resources to manage four streams at once, which is why there is no large improvement from using streams. Either, the resources have to be limited for the GPU to handle four streams at once, or the streams have to be serially executed, which eliminates their purpose.

Replacing current solutions with GPUs could allow for lower production costs. Hopefully this result can act as a basis for continued work and development within GPU programming at SAAB.

### 6.2 Future work

As explained in section 4, the original algorithm does not lend itself to GPU parallelism, because of the logic involved. GPUs thrive in optimal SIMD conditions, where every thread is guaranteed to do the same work as the others, and when this is no longer ensured, a lot of performance is lost. It is important to remember that the test algorithm probably performs slightly better than the original algorithm would with a GPU. This is because the original algorithm contains more logic, if and else-statements, and would
therefore result in more warp divergence. If the algorithm could be rewritten to better suit a parallelized implementation, performance could improve.

Allocating and moving data to and from a GPU is a costly operation. Rewriting the algorithm should therefore also prioritize making as few CUDA API calls as possible to memory copies and memory allocations. Finally, there are a lot of overhead costs involved with using a GPU, which clearly shows at smaller problem sizes. Therefore, utilization of a GPU can only be recommended to SAAB at larger problem sizes, ranging from $2^{10}$ and upwards.
References


