Institutionen för datavetenskap
Department of Computer and Information Science

Final thesis

GPU-accelleration of image rendering and sorting algorithms with the OpenCL framework

by

Anders Söderholm & Justus Sörman

LIU-IDA/LITH-EX-G—15/064—SE

2016-02-03

Linköping University
Department of Computer and Information Science

Final Thesis

GPU-accelleration of image rendering and sorting algorithms with the OpenCL framework

by

Anders Söderholm & Justus Sörman

LIU-IDA/LITH-EX-G—15/064—SE

2016-02-03

Supervisor: Unmesh Bordoloi
Examiner: Unmesh Bordoloi

Abstract

Today's computer systems often contains several different processing units aside from the CPU. Among these the GPU is a very common processing unit with an immense compute power that is available in almost all computer systems. How do we make use of this processing power that lies within our machines? One answer is the OpenCL framework that is designed for just this, to open up the possibilities of using all the different types of processing units in a computer system. This thesis will discuss the advantages and disadvantages of using the integrated GPU available in a basic workstation computer for computation of image processing and sorting algorithms. These tasks are computationally intensive and the authors will analyze if an integrated GPU is up to the task of accelerating the processing of these algorithms. The OpenCL framework makes it possible to run one implementation on different processing units, to provide perspective we will benchmark our implementations on both the GPU and the CPU and compare the results. A heterogeneous approach that combines the two above mentioned processing units will also be tested and discussed. The OpenCL framework is analyzed from a development perspective and what advantages and disadvantages it brings to the development process will be presented.

Contents

List of Figures

Listings

List of Tables

1 Introduction

This thesis report is the culmination of a bachelor thesis work on GPU-acceleration with the OpenCL framework, at Linköping University. In this thesis the authors will examine the potential benefits of applying GPU-acceleration to common types of algorithms with a focus on performance when using a lower end GPU, the Intel HD Graphics 4600. This is in effort to determine and analyze how significant, if at all the gains in terms of performance are in a commodity desktop computer. The discussion section of this report will also contain an analysis of common hurdles and best practices that should be considered before one chooses to apply GPU-acceleration to a development project.

1.1 Motivation

As software development advances and programs require more and more computational capabilities the hardware side of things and more specifically the CPU has been struggling to keep up. In terms of clock frequency hardware designers has gotten closer to the threshold of what is realistically feasible on a single processor core, prompting the rise of multi-core processors [1]. All the while in commercial settings a significant source of processing power goes unused, the GPU. If more software developers made efficient use of the GPUs available in modern computers it would likely be possible to greatly increase computational performance when running everyday programs. Increased performance which in turn will lead to more advanced software as well as a better user experience.

1.2 Aim

The purpose of this thesis is to provide a proof of concept, that is showing that there is a significant increase in computational performance to be had by applying GPU-acceleration to commonly used algorithms which are computationally demanding. The intention is to find and select appropriate algorithms that normally has a singlethreaded sequential implementation and develop our own versions, GPU-accelerated multithreaded versions. This report will provide a detailed analysis of the results as the different versions are compared to each other both in terms of performance and difficulty of implementation.

1.3 Research questions

- Can GPU-acceleration be used to increase computational performance of merge sort algorithms when using an integrated GPU available in a commodity desktop computer?

It has already been well established that for specific types of algorithms which lends themselves to be parallelized to a considerable degree GPU-acceleration will provide performance improvements. A topic which has not yet received much attention from researchers is the effect in terms of performance when applying GPU-acceleration to algorithms that while parallelizeable might not be so to an optimal degree. The same holds true for the question of to which extent a common office workstation with an integrated GPU benefits from GPU-acceleration.

- Can GPU-acceleration be used to increase computational performance for image processing algorithms when using the integrated GPU available in a commodity desktop computer?

Image processing algorithms tend to be very suitable for parallel execution and can often benefit from being run on a highly parallel processing unit like a GPU. It is already known that using a high-end GPU to perform tedious filter processing or similar algorithms is beneficial but is an integrated GPU up to the task.

- When does the potential benefits of utilizing the GPU for general purpose computation warrant the increased complexity introduced to the development process?

While many algorithms has at least some minor potential for increased performance when implemented with GPU-acceleration some can not be parallelized or doing so would come with such a great amount of overhead that any gain in performance would immediately be lost. When determining the added value of GPU-acceleration one also needs to account for the added complexity that multi-core programming in general and GPU-acceleration in particular introduces to the software development process.

1.4 Delimitations

This report and the analyses herein will be limited to specific algorithms selected for the purpose of this research. While the algorithms in question will be selected based on qualifications believed to allow for extrapolation, variance in terms of how efficiently a given algorithm can be parallelized will still exist. The authors of this report will make use of GPU-acceleration via the OpenCL framework to implement two distinct type of computationally intensive algorithms, sorting and image processing algorithm(s).

1.4.1 Sorting

The results and conclusions of the sorting algorithm found in both the discussion and the results section will be focused on how our GPU-accelerated sorting compares to our sequential implementation. The authors does not intend to compare the implementation to any existing GPU-accelerated implementations.

1.4.2 Image processing

Much like the sorting delimitations, in the image processing part of the result section the focus will be on comparing a GPU-accelerated image convolution algorithm to our sequential implementation. The type of image processing that will be performed will be limited to image convolution with filters.

2 Background

This report is written as part of a bachelor thesis work at Linköping University at the Department of Computer and Information Science(IDA). The work was conducted at Mindroad AB in Linköping, Mjärdevi.

2.1 Hardware limitations

As has already been mentioned in the introductory section of this report, hardware designers are rapidly approaching the limits of the processing power that can be harnessed from a single core [1]. In fact, when talking about what is actually feasible in a home or office computer in terms of heat dissipation and component size the aforementioned limit has already been reached. Up through the late 1990s processor chip performance increased by roughly 60 percent annually, about 40 percent in the early 2000s and down to about 20 percent by the year 2004 [1]. This steadily rapid decline in processing power gained for the amount of time and resources spent developing new chips led CPU manufacturers down the path of multi-core processors.

With multi-core CPUs comes the potential for increased performance, however not all programs and their underlying implementation algorithms lend themselves well to being parallelized [2]. One of the more prominent challenges facing software developers in the years to come will be to efficiently implement their programs, taking advantage of all resources available to them. However an important aspect of software development is being able to recognize when not to make excessive use of multithreading and GPU-acceleration, as doing so could result in performance decrease instead of performance gain [2].

2.2 Craving for performance

With the great advances in single core computational performance that was had all the way up through the 1990s consumers came to expect a certain amount of performance gain when purchasing new hardware [2]. This consumer expectation coupled with previous years performance gains led to a very competitive market where increased performance by any means necessary became the standard [1].

2.3 Computationally intensive algorithms

The barrier for how fast a human can calculate advanced mathematical problems was reached a long time ago, computational hardware was needed to progress beyond this limitation. An historic example of using computer algorithms to solve problems that required so much computation that they could not be solved manually is the code breaking that occurred during the second world war [3]. This involved cracking ciphers produced by the infamous enigma machine, a task that would not have been possible without using early computers to sift through the millions of possible combinations. Since then computers have evolved and their computational power increased enormously but even more calculation heavy problems emerged as endless possibilities were discovered when employing this new type of hardware.

2.3.1 Sorting

A recurring problem in computer science is taking a list of unordered elements and generating a sorted list containing those same elements [6]. While this could appear to be an relatively simple task when the list contains a limited number of elements, creating an algorithm that remains efficient even as the list grows in size is quite challenging. There are many types of sorting algorithms available, all different from each other but there are some common principles that are followed.

One of the simpler sorting algorithms is bubble sort [7] which iterates from the beginning of the list and pushes the larger elements back in the list by swapping the elements. If we find that the element we are comparing with is larger than the one we are currently pushing back we stop pushing the element forward and start pushing the larger element instead. This type of sorting algorithm is called an “exchanging algorithm” because it exchanges an element in the list every iteration if needed. Another type of sorting algorithm which is very similar is the insertion type of sorting algorithm. Instead of continuously swapping elements it will iterate over the list and find the most suitable place for an element and insert it at that place. The selection type of sorting is also quite similar. It iterates over the list, finds the smallest element and puts it at the beginning of the list. This process is repeated with the second smallest element and so on until the list is sorted.

The Divide and conquer [7] method on the other hand differs from the previously mentioned methods in that it does not iterate over the list. It will instead repeatedly split the list up into several smaller lists until the sorting problem becomes trivial, then it will gather the sorted pieces to form bigger and bigger sorted pieces until we have a complete sorted list. The last common type of sorting algorithms are the so called distribution sorting algorithms, they are usually not meant to sort a list by themselves but they can be used to speed up another sorting algorithm by dividing the work into smaller problems. The divide and conquer principle is the currently the most popular sorting method because it is a lot faster than the other sorting methods mentioned above [7], although it is much more complex to implement than for example a simple insertion sort algorithm.

2.3.2 Image processing

Image processing is computationally heavy because the datasets that needs to be iterated over are often quite large [4]. When processing an image we often need to know information about each individual pixel as well as the pixels surrounding it. The image manipulation performance problem is a non-trivial one because the amount of pixels that needs to be iterated over can grow to an enormous amount rather quickly. Luckily these kind of problems are usually very parallelizable but they are still computationally heavy even if parallelized [5].

3 Theory

In this chapter the authors attempts to provide an overview of parallel computing, parallelizable algorithms and the OpenCL framework as well as a brief look at the CPU and GPU. All of these are important concepts to understand in order to appreciate the findings and analysis portions of this report later discussed in the section 6.

3.1 Hardware Differences

Some combinations of hardware might perform better than other combinations depending on the type of workloads involved. The reason for this is that the architectural design will differ from vendor to vendor and even between product lines from the same vendor. Some hardware is designed to work efficiently with large datasets, performing the same instructions on the whole set. Others are optimized for smaller datasets, being able to perform more advanced instructions on them very rapidly.

3.1.1 CPU

The CPU is a processing unit using the Single Instruction Multiple Data (SIMD) [8] architecture, this means that the processing unit can perform a single instruction on multiple data inputs. This type of processor is often very fast when it comes to performing sequential instructions, it is not uncommon to have clock frequencies of over three gigahertz(GHz). The drawback of the CPU design is that it is a general purpose processor, i.e. it should be able to perform many different kinds of processing fairly well. The downside of such a general purpose processing unit is that it likely will not perform as well as optimized hardware even if it running at a faster clock frequency [9]. On the other hand however it is very easy to develop for due to its versatile performance.

3.1.2 GPU

The other major type of processing unit is the GPU, which uses the Multiple Instruction Multiple Data (MIMD) [8] architecture instead of the SIMD employed by the CPU. This architecture can perform multiple instructions over multiple data inputs during a single clock cycle. Processors with this type of architectural design are usually called vector-processors because they are optimized to work with vectorized data structures. The clock frequency of this type of processor is generally not as fast as that of a SIMD processor, typically about five hundred to one thousand megahertz(MHz) depending on the manufacturer. The GPU is not as general purpose as the CPU but will often perform very well in its dedicated areas, these consists of handling large datasets and performing parallel tasks. The parallelization level that can be achieved on a GPU is far greater than that of the CPU and herein lies the strength of GPU-acceleration. A caveat of this being that poorly parallelized algorithms will often be slower on the GPU than a sequential implementation of the same algorithm on the CPU.

3.1.3 Integrated VS discrete

GPUs comes in two form factors, integrated and discrete. The integrated GPU is usually placed inside the CPU die to form an APU [10]. It should be noted that while APU is the proper term for this type of hardware where the CPU and GPU exists on the same die, some vendors, most notably Intel will still refer to this type of product simply as a CPU. The smaller integrated GPU is typically not meant to be particularly powerful, instead they are designed for low power consumption and to be small enough to fit on the same die space as the CPU. On the other hand the discrete GPU has the whole die to itself and can therefore be designed to be able to do more advanced and faster calculations compared to an integrated GPU.

The type of memory used also differ between the two [10], the integrated GPU is primarily using the systems own memory resources and will therefore often have a larger albeit not as fast a memory pool as the discrete GPU to work with. The discrete GPU is more often than not supplied with specialized high bandwidth, low latency memory. This memory is expensive and that is the reason for the integrated GPU memory pool not being as big as the system memory. When developing software one needs to be aware of one's target audience and what type of system this group will use to run the program. This is because the optimizations and design of the program differ a lot depending on what hardware the end user has in their machines.

3.2 Parallel computing

The term "Moore's law" was coined in 1965 and describes the advancements in computational performance and how it will double every two years. Up until the late 1990s this performance increase came in the form of putting more transistor onto a single chip [1]. As it turns out however it is not feasible to keep on adding more transistors forever. When hardware manufacturers began to push against the limits of how much performance could be had from a single core multi-core development really caught on. This trend lead to the ubiquitous adoption of parallel computing we see today and the massive increase in compute power available.

There are two main methods for utilizing multi-core processors, the first is having a Message Passing Interface(MPI) [8], which is very scalable, allowing the program to distribute the workload over several different machines. This is because the messages are passed between the connected computers via a network interface that is very easy to scale up and distribute, the computers does not even have to be in the same buildings to be able to cooperate efficiently with each other. The second method is the Shared Memory Architecture(SMA) [8], this method shares some of the data between the threads running in the system. The data is shared via a global memory which all threads can access. This method does not scale as well when using multiple interconnected systems [8].

3.2.1 Multithreading

To get greater performance out of a multi-core processor a programmer can make use of multiple threads to perform different operations. One of the problems inherent to multithreading is two or more threads manipulating the same data at the same time. This is solved by means of concurrency control which is crucial for any application using more than one thread [8, 10]. To implement concurrency control a software developer places locks and semaphores around data that several threads will want to access as this could potentially cause problems if these threads were to access the same data simultaneously.

3.2.2 Specialized hardware

The general purpose CPU is as its name indicates made for general purpose computations, which means that it is not specialized to only do one thing like for example an accelerator card specialized for physics. Therefore it will often not perform as well as a dedicated hardware solution like a Nvidia Tesla card [11] when performing computation for large scale physics simulations. These massive parallel calculations would take a much longer time to execute on the CPU. Some of these dedicated hardware cards can accelerate all manner of scientific applications, from protein folding to advanced physics simulations.

3.2.3 Clusters/Supercomputers

When a single machines compute power is not enough, not even if the system contains multiple powerful cores, a cluster can be made to boost the performance. A cluster is a group of computers [8] connected via a network that allows them to communicate with each other and thereby form a supercomputer that consists of multiple compute cores and that has a tremendous amount of primary memory. Due to the separate nature of the systems a message passing interface is required to allow parallel programs to utilize all the available compute power[8].

As an example a cluster can easily perform massive weather simulations where the map is divided into sections and one computer in the cluster might have one or more sections of the map that it is responsible for. The computers then send their results to its neighbors and depending on what their response is decide how the weather should change in its own section for the next iteration of the process.

3.3 Parallel problems

Solving problems faster without upgrading one's hardware is possible. You can sometimes split a larger problem down into several smaller problems which can then be solved concurrently instead of sequentially. This is what is referred to as a parallel problem, but not all problems are as inherently parallel as others. Some problems can not even be effectively parallelized due to the subtasks involved being strongly dependent on information about each other [10]. There are two main groups of parallel problems, data parallel problems and task parallel problems [10].

3.3.1 Data parallel

These types of algorithms can be executed in parallel due to how the data that is iterated over is structured. The data has to be structured in a way so that each parallel process does not write to another processes data without them being synchronized. Examples of algorithms that are generally data parallel in nature are image processing algorithms and vector math algorithms. One of the hurdles associated with these types of problems are memory constraints [10]. In fact when implementing efficient data parallel algorithms one will often find that so many operations can be done in parallel that the memory bandwidth is the limiting factor.

3.3.2 Task parallel

When an application run more than one algorithm and these algorithms do not rely on shared data, these tasks can be run in parallel to achieve better performance. The problems with these types of algorithms is that they usually need to be synchronized with each other at some point [10]. This is where load balancing comes into play. Load balancing refers to the act of actively distributing the workload over the available resources in an effort to avoid bottlenecking the performance. This means that the order in which the tasks are being run is a limiting factor in regards to performance [2, 10]. Finding the right sequence in which to run the tasks for maximum performance is left to the developer optimizing the implementation.

3.3.3 Embarrassingly parallel

Some algorithms are referred to as being embarrassingly parallel, the embarrassing part in these algorithms comes from the fact that they are well suited to a parallel implementation and does not necessarily relate to the amount of developer time spent implementing the algorithm [2, 10]. Generally what makes an algorithm embarrassingly parallel is a very limited need for communication between processes.

When we have a problem that could be solved with an embarrassingly parallel algorithm the new limiting factor is the supporting hardware [2]. In this case one need to consider how large a portion of the problem that is allocated to each process so that the hardware can work at an optimal level, if the portion is to big the level of parallelization is to low and we can not fully utilize the hardware. On the other hand if the portion is to small there will be overhead caused by the excessive scheduling and launching of threads[10].

3.4 OpenCL

The need for a common standard of how to handle heterogeneous systems were needed as it became increasingly common to have multiple processing elements in a single computer [1, 10]. The development of OpenCL [12] began in June 2008 by the Khronos group and representatives from CPU, GPU and embedded-processor companies [12]. About 14 months later in August of 2009 version 1.0 of the OpenCL framework was released for public usage. The standard is meant to make development of programs that utilizes heterogeneous systems easier. The concept of the OpenCL framework was conceived by Apple who has retained the rights to the OpenCL trademark. Today's systems contain all sorts of different hardware, ranging from GPUs to field programmable gate arrays(FPGAs), hardware that often is not utilized to its fullest. The framework API is based on the C99 standard but wrappers exists for many other popular programming languages like C++, Python and Java [12].

As of today, the 1.2 version has gained widespread support among several major hardware vendors such as AMD, Nvidia and Intel, with most devices released by said vendors being compatible either with it or the previous version 1.1. Driver support for a new version is sometimes released a little slow. Currently OpenCL version 2.1 is released but support for this version is still lacking. Version 2.0 is supported by AMD, which are at the forefront when it comes to having OpenCL compatibility for their hardware. Intel now supports OpenCL version 1.2 on most of their new devices and has some prototype drivers for version 2.0 while Nvidia is only supporting version 1.1 on most devices and version 1.2 on some newly released devices.

3.4.1 Model

The OpenCL model provides a visualization of the computer system [10, 12]. In this visualization the CPU is referred to as the host, the coordinator of the computer system. The other hardware components in the system are devices, like graphics cards or other forms of dedicated accelerator cards. Each device has one or many compute units which can run a workgroup. Each of these workgroups consists of several work-items and each work-item represents the thread that actually executes on the device. The compute units contain one or many processing elements which the work-items utilizes. The source code for the operations performed by a work-item is called a kernel. This kernel code is compiled at runtime, while this runtime compilation does add some overhead to the application, it gains the ability to dynamically adapt to its environment and the same application can run efficiently on different types of hardware. An abstraction of this model is shown in figure 1.

3.4.2 Platform

OpenCL is a framework for enabling multi-core acceleration of CPUs, GPUs and other types of accelerators. While there is a defined C-style language API available when developing software that makes use of this framework the backend implementations of these API calls are left up to each individual hardware vendor [5, 10, 13]. A platform in the simplest sense is this vendor specific implementation of the OpenCL framework and it is through this layer that the host will communicate with the available devices. Because of this all platforms may not be compatible with all devices, this is especially true for devices manufactured by a vendor other than the platform developer [10]. Once the correct platform for the device one wishes to target has been selected it will be used in the creation of a context.

3.4.3 Context

This structure is the bridge between the host and the device in terms of memory [10, 13]. Here we can manipulate what data is allocated on the device and also get information about the current configuration of the devices in the context. Setting up the kernel arguments and output memory is also done in this

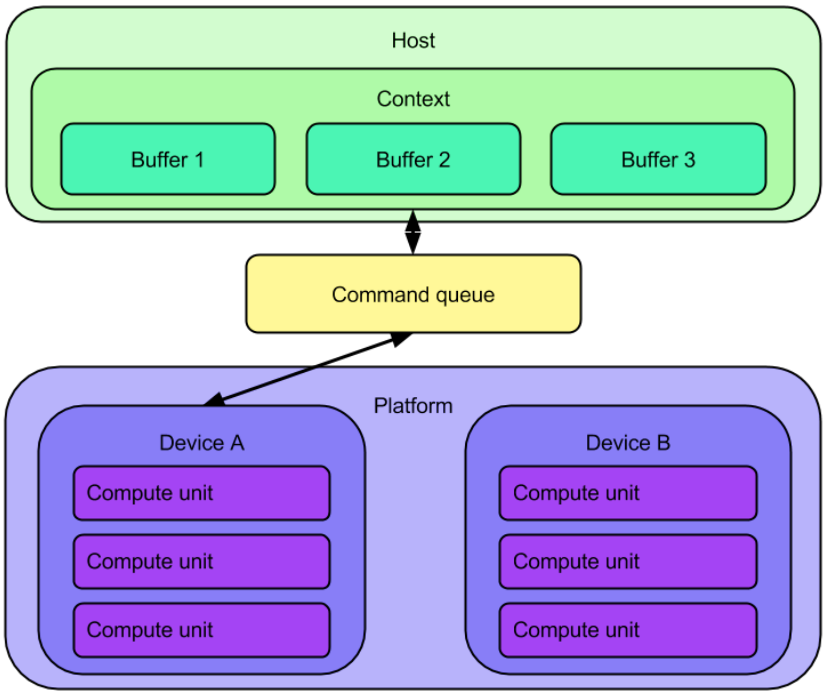


Figure 1: OpenCL model abstraction

structure. The context can be configured in several different ways. If we have several devices in the system we can create a context for all of them at the same time if they are in the same platform otherwise this is impossible to do. The other choice is to have different contexts for each of the different devices. Having either a single or multiple contexts is dependent on the design of the kernels and how they are run. If we need the CPU and GPU to cooperate we create a context containing both these devices[10] if it is possible. To have them in the same context often makes the development process easier because now we do not need to move data between the two if we were to share data.

3.4.4 Device

In OpenCL a device is the physical hardware represented as an abstract data structure [10, 13]. The information in this structure includes but is not limited to, the number of compute units it contains, the prefered workgroup size multiple and the maximum number of work-items per workgroup it supports. A device is primarily used to initialize the command queue and context. The information contained in this structure can used by programmers to optimize the algorithm for specific hardware [10, 13].

3.4.5 Command queue

To utilize the devices accessible in a given context we need a separate command queue for every device in the context [10, 13]. With this structure we can queue up kernels to be executed as well as setting the allocated buffers in the context to contain data of our choosing or send commands to the device to activate different features such as out-of-order execution if the device supports this feature. The benefit of performing out-of-order execution is that it improves load balancing by maximizing the utilization of the GPUs functional units. The drawback of out-of-order execution being that the developer manually has to ensure that operation dependencies are satisfied before execution [10]. When the execution of a kernel is completed we can then retrieve the data from the buffers to which it was allocated.

3.4.6 Memory

There are four different types of memory in OpenCL [10, 13, 14]. These four types are global, constant, local and private memory, this memory hierarchy and its connection to the host memory is illustrated in figure 2. All these types are found in different locations on the hardware and are handled in different ways. First is the global memory which is the shared memory buffer that all work-items are able to access. This memory is the slowest in terms of access speed but just because it is the slowest of the four does not mean it should not be used. Big data structures can be saved in this memory as well as data that needs to be shared between all the work-items. Second is the constant memory which is a special type of global memory. The data stored there can as the name indicates only be read from and not written to. In some hardware declaring a data structure as a constant will allow caching of the data and speedups may be gained compared to only using standard global memory.

Local memory, the third type, is much smaller than the global memory and is only shared between work-items in the same workgroup. The benefit of using this memory is speed, it is much faster than the global memory. The problem here is that if the workgroup uses too much of this memory it will be cached in the global memory and will therefore lose most of the speed that we were trying to gain by using it. The fourth and final type of memory is the private memory which is used exclusively by the work-item itself. This memory can not be shared in any way and usually consists of several different registers close to the physical core. Much like local memory, private memory is quite limited in terms of size, a work-item attempting to use more memory than is available will result in a loss of performance due to data being cached to global memory instead. Because of this the kernel should not use an excessive amount of private memory as this could have a severely negative impact on performance [10].

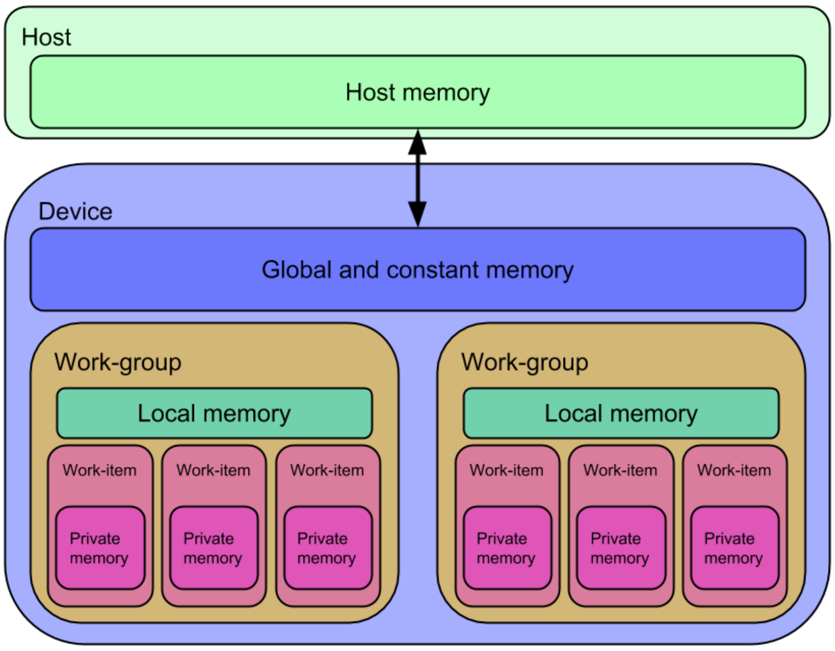


Figure 2: OpenCL memory hierarchy

3.4.7 Events

Events in OpenCL is a feature that allows the developer to ensure proper synchronization for operation execution [10, 13]. When one of several different enqueue commands is issued via a command queue an event is created, after the enqueued operation has been carried out the event will call back to inform that the operation has been completed. The OpenCL enqueue commands all take an optional parameter of event type. Utilizing this parameter when issuing commands via the command queue we can inform the host that the issued command is dependent on another operation and should not be executed before the callback from that operations event has received. This method of synchronization is referred to as event chaining and should primarily be used when the command queue is set to run out-of-order executions.

3.5 Performance optimizations

Implementing an algorithm with OpenCL is often not particularly difficult but making it run fast and efficiently in a manner that utilizes the available hardware resources to its full potential is another matter. There are currently several vendors with many different pieces of computational hardware available on the market. All of which prefers different types of optimizations be it in how they handle memory operations or in what data type is used in calculations.

3.5.1 Memory management

The first problem a developer faces when implement an algorithm in OpenCL is choosing how to handle the memory access patterns of the algorithm. The hardware might prefer reads of a specific size and pattern to perform at peak capacity. In order to utilize the memory efficiently, techniques such as data padding were useless junk data is interspersed together with the real data in order to ensure optimal read access might be used. Such techniques are very hardware dependent but should be considered if the application is only targeted to run on specific hardware. If one manages to find the ideal implementation for the targeted system, a substantial speedup may be gained. Using data types that are vectorized can also boost performance [10]. The memory bus is often capable of transferring large amounts of data as long as it is contiguous. The positioning of the data is also important, if you can use a memory module closer to the core then it is usually better to move data to it and use it from there if the data is to be accessed many times. Otherwise it will just bog the performance down. Another thing that is crucial to remember is that if we have a discrete GPU we do not want to use the systems own memory due to it being slower compared to the accelerator cards dedicated memory. However if the GPU is integrated within the CPU we can perform a so-called zero-copy [13] and thereby not creating an extra instance of the data in memory just for the GPU, minimizing the startup and teardown time of the operation.

3.5.2 Workgroups and work-items

Problem number two is determining an optimal workgroup size. The hardware manufacturer will recommend a preferred workgroup size multiple for all OpenCL compatible products. While this workgroup size is unlikely to be the optimal one for any given algorithm it will often perform decently well regardless of what hardware and kernel program you might be using. There are instances where the default preferred workgroup size multiple will greatly underperform [15], in these cases the design of the algorithm being used is most likely the main cause of this. The hardware usually performs well when using a workgroup size close to the recommended one, one power up or down usually results in a minor increase or decrease in performance. Setting the workgroup size to be at or close to the available minimum or maximum is usually a bad idea as this will often cause the algorithm run slow or not at all depending on the implementation [14].

3.5.3 Algorithm design

The third and final problem is the design of the algorithm itself. When first writing the loops and statements the code might look good on the screen but the compiler could have trouble optimizing the algorithm for the hardware [16]. The performance might be decent but if one more thoroughly inspects the degree to which the ALUs are utilized, one sometimes realize that the processing unit is only using a fraction of the compute power at its disposal. This could mean that the algorithm is memory bound and an alternative implementation, possibly using manual loop unrolling might be preferable [10]. The choice here is to either alter the way memory is handled or rethink how the algorithm works on the data. Changing to a more preferable data type like a vectorized one might be beneficial as doing so makes better use of the wide memory bus that is often available for data transfer between the global and local memory. Thereby having more data for the ALUs to process which will often lead to a higher utilization level of the device [10, 13].

3.5.4 Conclusion

Writing a parallel algorithm using OpenCL is often not exceedingly hard depending on the problem. To write an algorithm that utilizes the hardware's resources in an effective way requires extensive knowledge of the hardware used and many of the features in OpenCL. Although depending on the algorithm the most common way to have the algorithm run fast is just to choose a good workgroup size that will perform well on the targeted hardware.

Some other things to consider when designing an algorithm is that IF statements should be used with care because the GPU is not as good at handling branch misses as the CPU because of the lower clock frequency and therefore the penalty is much bigger [10]. A branch miss is when the processing unit makes a faulty guess on what code is supposed to be executed which happens often with IF statements. There are some other things that one should keep in mind as well and that is to unroll loops if possible, store small variables in local or private memory if possible because it is much faster than the global memory.

4 Method

In this section the authors will describe in detail the approach taken both in the algorithm selection process as well as in the implementation of said algorithms. A rundown of the hardware utilized for the implementation and benchmarking as they pertain to this thesis work will also be provided.

4.1 Algorithm selection

In order to provide a varied and representative analyses of GPU-acceleration on traditionally sequentially executed algorithms two distinctly different types of algorithms have been selected. The first type is a sorting algorithm, this type of algorithm was chosen as the sorting of data is a common problem in software development and computer science in general as many algorithms will make use of and depend on an efficient sorting component. At the behest of the company for which these analyses are carried out at least one of the selected algorithms should relate to either signal or image processing. Therefore the second type of algorithm chosen relates to image processing. The reason for this being that this type of algorithm is expected to provide significant performance improvements if implemented in parallel [4] even on an integrated GPU.

4.1.1 Sorting algorithm

Finding a sorting algorithm that could not only be parallelized but also stands to benefit from a parallel implementation was the first task of the selection process. Finding sorting algorithms that could be parallelized proved not to be particularly difficult. Almost all divide and conquer type algorithms can be parallelized in some manner but finding an efficient way to maximize the number of concurrent threads working on the same list is more challenging.

The first algorithms that were considered were bucket sort and merge sort [6, 7]. Bucket sort was considered a good fit for parallelization due to its distributed nature. The algorithm employs a distribution type of sorting which as the name implies distributes the list into smaller lists that can then later be sorted more efficiently. Bucket sort is usually implemented together with another sorting algorithm that sorts the buckets(the smaller lists), because it is less efficient when implemented by itself as it consumes too much memory due to recursion. Merge sort is a divide and conquer type of sorting algorithm. The essence of how it functions is dividing the list into equally sized smaller pieces until the sorting becomes trivial e.g. the pieces are split to the point where they only contain a single element. Then the algorithm starts to merge the elements in a tree like structure starting with the two first elements in the list. These two elements now form a small list of sorted elements. It will do the same to the next two elements in the list and merge them together to form a list of four elements, and so on until the list is completely sorted.

The problem with parallelizing the bucket sorting algorithm is the insertion into buckets. These buckets can only be accessed by one thread at a time, there may be many buckets and there needs to be synchronization when inserting into them. This will hinder the performance of the algorithm substantially. Merge sort on the other hand is a better fit for parallelization. As already mentioned the algorithm is a divide and conquer type algorithm which works by splitting the problem into smaller chunks that are easier to solve. Each of these small chunks can be run concurrently and thereby we can achieve a high level of parallelism.

During the literature study as we came to understand the pros and cons of the algorithms we concluded that bucket sort was not an ideal algorithm to parallelize. We turned our attention instead to merge sort that while more challenging to implement showed more potential for parallelism. There are already existing parallel merge sort algorithms, a prominent example of which is bitonic merge sort [17]. Bitonic merge sort is a parallel sorting algorithm that uses a fixed comparison network to sort the list, otherwise the sorting algorithm is very similar to the standard merge sort algorithm. In the end the standard merge sorting algorithm was selected to be implemented because of the time spent on the prior study of the algorithm and thought that had gone into it when trying to figure out how to utilize the memory properly when parallelizing the algorithm.

4.1.2 Image processing algorithm

The type of image processing algorithm select is with image convolution with filters [10, 18], a form of image processing that lends itself very well to being parallelized. The reason this type of algorithm is so well suited for a parallel implementation is that operations are performed on each individual pixel of the image and only limited information about the surrounding pixel values are needed. How limited is decided by the size of the filter matrix employed, for certain types of filters a simple 3x3 matrix will suffice, for others either a fixed larger masking matrix or one of varying size may be preferable. Oftentimes using a larger size filter can enhance the desired effect. This is illustrated in figure 3 where a sharpness filter of varying size is applied to an image.

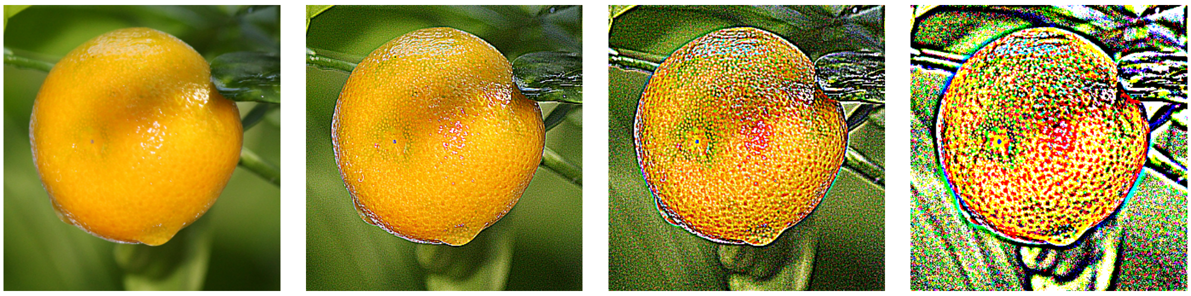

(a) Original image (b) 3x3 sharpen filter (c) 5x5 sharpen filter (d) 11x11 sharpen filter

Figure 3: A sharpness filter of varying size applied to an image (public domain image used)

The size of the filter used in combination with the resolution of the image it is applied to will determine the execution time of the algorithm. With the image resolution determining the amount of pixels and the filter size how many operations per pixel need to be performed.

The effect of the filters will also depend on the resolution of the image to which it is applied, a 3x3 box blur filter matrix will provide a slight uniform blurring of a 480p resolution image that is even more subtle than the effect shown in figure 6. Using the same filter on a 4K resolution image will make for a much more minute effect meaning that with some types of filters the filter size needs to increase in step with the image resolution. For other types of filters such as edge detection the application of a small filter on a large resolution image will still yield the expected result.

4.2 The hardware

In order to provide the reader with the requisite information needed to either replicate the findings presented in section 5 or to facilitate drawing their own conclusions, the authors will give a brief rundown of the systems used for this thesis work.

4.2.1 Intel i7-4790 CPU

The Intel i7-4790 CPU [19] is one of the fourth generation i7 processors from Intel. This model is a desktop variant with a locked clock frequency of 3.6 GHz but can boost up to 4.0 GHz when needed, although not all cores at once due to thermal limitations. The stepping schema of the processor is 2/3/4/4 which represents how many extra multiples the cores can boost compared to the Front-Side Bus(FSB). The first number represents how many extra multiples all cores can have, the second is how many half of the cores can have i.e. 4 cores on this particular processor, the rest of the schema follows the same pattern. The processor is equipped with 4 physical cores and Intel's Hyper-Threading Technology so the core count the user actually sees are 8. The memory configuration on the chip is an 8 megabytes of level 3 cache that is shared across all cores. The level 2 and level 1 cache is separate to each physical core with a size of 256 kilobytes and 32 kilobytes respectively. The level 2 cache is the cache that OpenCL sees as the local memory of each compute core and the first and third level is only handled by the hardware itself.

The OpenCL implementation for this processor sees 8 compute cores that has a maximum number of 8192 work-items in a workgroup per compute core [20]. The recommended workgroup size multiple is 128, which is quite high compared to that of discrete graphic cards.

4.2.2 Intel HD Graphics 4600 GPU

The Intel HD Graphics 4600 GPU [19, 21] is an integrated graphics processor designed to be a low power unit that runs at a clock frequency of 350 MHz. The processor will dynamically boost its clock frequency when needed, up to 1.2 GHz. Thereby making the chip perform better when needed while at the same time keeping power consumption to a minimum during low utilization. The are a total of 20 compute units in the GPU and each compute unit contains two Floating-Point Units(FPUs) which can perform one addition and one multiplication concurrently. The FPU handles both integers and floating-point values equally efficient. The SIMD width is the number of concurrent instructions that can be performed on GPUs. The Intel HD Graphics 4600 has a SIMD width of 4 but Intel has implemented an advanced SMT architecture that allows for dynamic scaling up to a width of 32. The memory on the GPU is divided into three levels of cache that is accessed via a 128 bit wide bus which makes transfers of 64 byte data reads and writes in a single clock cycle possible. The first and second levels of cache the programmer has no control over as they just cache data that is often accessed. The third level of cache is a 256 kilobyte cache that stores data for each separate slice. A slice is a subdivision of the GPU's total compute units which is a group containing 10 compute units. The Intel HD Graphics 4600 as a whole contains two slices and other models can have up to 4 slices. These two slices share a common cache of 64 kilobytes per slice making for 128 kilobytes of common storage for the Intel HD Graphics 4600 model. The hardware supports up to 7 threads per compute unit which means that for each slice there is support for up to 70 hardware threads to be run concurrently. So on the whole chip 140 threads can execute concurrently. Each hardware thread can run up to 32 different software threads e.g. work-items in OpenCL this enables us to run a total number of 4480 work-items concurrently.

Taking all these specifications into consideration when implementing the algorithm is vital as they impose a number of limitations on our implementation. The first limitation is the memory size, a good data fit is crucial so that the cache can work as optimally as possible without needing to bounce data to a higher level of memory. Because of this each workgroup should not use more than 256 kilobytes of local memory. The transport of data back and forth from global to local memory can also be made more efficiently by moving data 64 bytes at a time. Workgroup size needs to be carefully selected, a workgroup size of 32 would likely be preferable due to the hardware limitations and the multithreading capabilities of each compute unit. Although depending on the design of the algorithm this number could be adjusted to better fit the implementation. The last limitation to consider is to ensure that there are enough work-items executing concurrently. Using a number of work-items that is a multiple to the number of hardware threads is likely a good choice to ensure an even load across all cores. So the workload needs to big enough and heavily parallelized to be run efficiently.

4.3 The software

The algorithms that were selected for this thesis work and which implementations are described in this report was developed on machines running the Windows 7 operating system with all essential updates released until this point in time installed.

The C/C++ compiler used during the development of the aforementioned applications is the 64-bit MinGW compiler version 4.9.2 with posix threads [22].

Intel's OpenCL Code Builder[23] version 1.4.0.25 64-bit version was used extensively to compile and analyze the performance of the kernel code. The reason behind using a third-party program to compile and test kernel code is that an OpenCL application always builds its kernels during runtime, meaning that a kernel either executes or it does not, with no way for the developer to know why it did not. This proved very useful for catching common mistakes such as syntax errors and to verify that the kernels behaved as expected during execution time.

The implementation of the image convolution algorithm makes use two third-party libraries, CImg [24] and Image Magic [25] in order to facilitate loading of file formats such as jpg and png into memory as well as enabling the writing of the processed images to disk.

4.4 Implementation

In this section the authors will describe in detail how the selected algorithms functions and how they were implemented with the OpenCL framework. When applicable several different versions of the algorithm will be brought forward to have its pros and cons examined together with an explanation of why it was deemed either suitable or unsuitable for a parallel implementation.

4.4.1 Merge sort

The main principle behind merge sort is that we divide the sorting problem into smaller parts until the sorting problem is trivial to solve, i.e. the small sublists are of size one and then the sublist is sorted by definition. When we have many smaller lists we can now merge them in pairs in a tree like structure and put the smallest element of each merging pair first. Doing this for all the pairs will result in an sorted list. The implementation of this can be done in several different ways, below are some examples of how we implement the merge sort algorithm.

4.4.1.1 Recursive

One of the simplest and probably most naïve way to implement a sequential merge sort is by utilizing recursion. To implement this we need two different functions, the first function divides the list into smaller sublists if the sublists are not of size 1 because then they are considered to already be sorted. We divide the list into left and right tree branches with their own memory pools as demonstrated in figure 4.

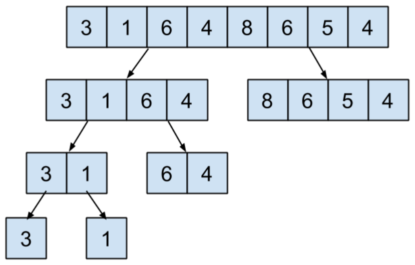


Figure 4: Recursive merge dividing sequence

When we have divided down to the bottom of the tree and we have sorted sublists of length 1 we need to merge them. Here comes the second function of the implementation, this function merges two sublists together. The merging is pretty straight forward, we allocate a result list that has space for both of the lists and goes through both lists and picks the smallest element in each list and inserts it into the result list. Continue this procedure until one of the lists are empty and then insert the rest into the resulting list. Then due to the recursion we will bounce up the whole tree of functions and after the last bounce we have a sorted list.

The implementation of the recursive method is not hard to follow or to implement if you are familiar with how recursion works. The main downside to the algorithm is that it uses a lot of extra memory compared to the amount of memory needed for the original list. The constant allocation of new memory space will take some time and the utilization of the CPU will not be very high because this algorithm is mostly bound by the speed of the installed main memory on the system and how the operating system allocates memory to the process. Also some overhead will occur as a result of the many nested function calls.

4.4.1.2 Iterative with swap

An iterative version that utilizes nested loops instead of recursion is usually beneficial because it avoids the overhead of multiple function calls caused by a recursive algorithm. Also an iterative algorithm is much easier to parallelize than a recursive algorithm.

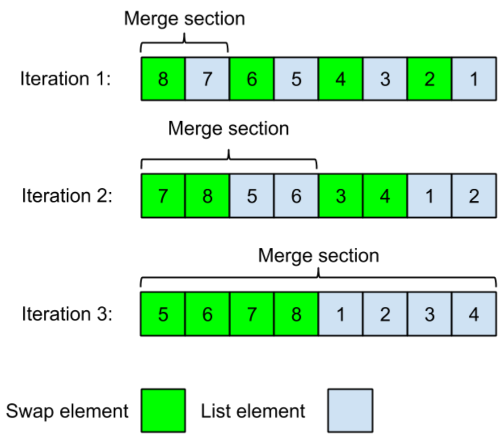


Figure 5: Iterative merge sort with swap

Now we will describe an iterative version of merge sort. In this iterative version we maintain a swap buffer that is used for storing the first half of all the merge sections. A merge section is a part of the list that is going to be merged into one bigger sorted section as shown in figure 5. It consists of two smaller individually sorted equally sized lists.

Step 1: The algorithm copies the elements in the first half of the merge section and puts the them into the swap buffer at the same index they had in the original list. Then the same procedure is done for all the others merge sections in the list.

Step 2: After the copying is finished we begin with the first merge section and merge the element from the swap buffer with the second half the merge section. This process is repeated until there are no merge sections left to merge.

Step 3: Then this process is repeated but the length of the merge sections are now doubled. This loop is run until the procedure has been performed with a merge section that has the same length as the complete list at which point the whole list is sorted.

By using this swap buffer instead of repeatedly allocating new memory for the merge sections the performance will increase and the memory usage decrease significantly. Although implementing the swap buffer rather than just allocating new memory is a little challenging, having the memory set up in this way will facilitate the implementation of the parallel algorithm.

4.4.1.3 Parallel implementation

By using the iterative version with swap as a base for the parallel version we can distribute all the merge section operations over the dispatched work-items. Thereby having a parallel algorithm where all merge sections are merged in parallel instead of sequentially. The merge section that a work-item is supposed to work with is calculated by having its thread id multiplied by two. A check is performed and if the offset corresponds to the index of the first element in a merge section we will perform a merge otherwise the thread will terminate its execution and do nothing this merge session. This is to ensure that only one thread merges the section.

The swap buffer is implement in the same manner as in the iterative version but we now can directly get the offset that we are supposed to work by using the one we just calculated. When the first half of the merge section is copied to the swap buffer we can start to merge the sections together. This is also fairly simple just like the swapping due to the already designated offset. Then after we have merged, all work-items needs to be synchronized to be able to begin the next run of swapping and merging with a greater length of the merge section. How the kernel is implemented is shown in listing 1.

```
basic merging kernel()
{
    if(my position is first in the merge section)
    {
        copy first half of the merge section to swap buffer

        while(sublists are not empty)
        {
            merge the sublists and insert into the list
        }

        while(swap buffer is not empty)
        {
            put rest into the list
        }
    }
}
```

Listing 1: Basic parallel merge sort pseudocode

This was the first implementation of our parallel merge sort. The problem that we encountered was that the kernel execution time can not exceed two seconds otherwise the Timeout Detection and Recovery (TDR)[26] daemon process will reset the drivers for the GPU and our kernel will halt its execution. Because of this we need to divide the later merge sessions into smaller chunks. The maximum number of merges that were allowed to be performed without hindering the performance of the earlier merge sessions but still prevent the halting of the later ones needed to be calculated. This was done by measuring the time each merge session took when performed with the basic implementation. After some measuring we decided to set the maximum number of elements that could be merged in a single chunk to 1/128 of the total list size. The merge chunk then took a little under half a second to be performed. This allows us to perform the merging within the time limit imposed by the TDR even on lower end hardware.

This solved the calculation part of the problem, during the first iteration of every chunk session we copied all the needed data to the swap and this took a considerable amount of time and we constantly hit the execution time limit even when having a relatively small merge size. The final version of the implementation takes the copy problem into account by using a separate kernel that only is used for copying the needed elements to the swap buffer. This can be done in parallel by having knowledge of the thread id and the current length of the merge sections. By doing this we minimize the time spent copying, also the kernel execution time goes down as it does not need to handle the copying. This enabled us to use a larger the chunk size, this will increase the performance by not having to save the state of the sorting as many times. The pseudocode of the final version is shown in listing 2.

One benefit with having the implementation in OpenCL is that we are not confined to only use the GPU. It is also possible to use the CPU in parallel, although the parallelism in this implementation is too fine-grained for the CPU to execute efficiently in the first merge sessions, but this implementation will work on both processing units.

```
chunking merge kernel with external copy()
{
    if(my position is first in the merge vector)
    {
        if(we are in a chunk)
        {
            get where we stopped last time
        }

        while(swap buffer is not empty)
        {
            merge the sublists and insert into the list
        }

        while(we have not inserted the chunk size number of elements from swap)
        {
            put elements of swap into list
        }
        save our state in the sorting process
    }
}
```

Listing 2: Chunking parallel merge sort with external copy pseudocode

4.4.1.4 Parallel implementation optimization

The choice of workgroup size is one of the first optimizations that should be done when the kernel code is finished, Intel recommends a workgroup size of 32 for the Intel HD Graphics 4600 which should be a good starting point. Although after some testing with the workgroup sizes we found that 32 runs well compared to most other sizes, a workgroup size of 16 runs faster with the implementation of our merge sort kernel on this specific hardware. The copy kernel on the other hand did not run as well with a workgroup size of 16 but performed very well with a workgroup size of 32. The execution time of the copy kernel is very short compared to that of the merge kernel so optimizing this kernel is not a priority as the impact will not be very significant.

The copy kernel is a solution that takes care of the problem with halting kernels caused by the sheer amount of copying that had to be done during the later merges. It is also a minor accidental optimization as it increases the parallelism of the copy sequence. The most basic merge kernel handled the copying to the swap buffer per work-item that were going to execute a merge. The copy sequence quickly became a bottleneck but it was not that hard to parallelize the copy across all work-items. The only catch is that the copying needs to be synchronized across all work-items and the only way to do this is to queue up multiple kernels where all work-items are synchronized between the executing threads as there is no way to synchronize between workgroups. The number of threads dispatched for this kernel is equal to half the size of the original list. The kernel copies two elements from the list to the swap if the thread's id is within the first sublist in a merge section. Thereby the kernel is running highly parallel and when a whole workgroup is within a sublist they combined copy 64 bytes of contiguous data which is what the hardware prefers, this will result in an efficient and fast copy. Pseudocode for this kernel implementation is shown in listing 3.

```
copying kernel()
{
    if(my position is in first part of the merge vector)
    {
        copy 2 elements from the list to the swap buffer
    }
}
```

Listing 3: Parallel copy to the swap buffer pseudocode

Most of the execution time spent in the merge sort is in the last few merge sessions when we can not utilize the whole GPU to its full extent, about 7% of the total execution time is spent when the GPU is performing at its peak and the remaining 93% is when it is underutilized. These numbers were calculated from the execution times presented in appendix F. This underutilization is the implementation's main bottleneck and it is a problem that as to be solved if the GPU is to competitively compete with the CPU. One possible solution is the use of cooperative merging in the later merge sessions. This would mean that the GPU would be utilized to a higher degree and this would greatly improve performance. The cooperation can be done in several different ways, some more complex than others. A version of cooperative merging has been described in [27]. In this thesis we implement a slightly modified version of this algorithm, below we will describe our implementation and the modifications made as well as briefly explain the standard cooperative merge sort algorithm.

The algorithm described in [27] works by splitting up the list into equally sized chunks that are then sorted. Then we use binary search to find where the splitting elements can be inserted into the other list and save that position. When we have done this for the complete list we have pairs of small sublists that can be individually sorted, with this algorithm many threads can concurrently operate on one merge section. The problem is that it is hard to understand and too complex to implement for this project. Our modified cooperating merge only uses two threads in one merge section instead of many. By having this simplification we can get a merge kernel that merges the two sublists from both the front and the back concurrently. The two threads will merge until they have processed half of the merge section each.

This optimization raised the degree of parallelism in the kernel which in turn lead to a huge performance increase. The copy kernel also needed to have some modifications done to it in order to be used with the cooperation kernel. The copy kernel is now copying the whole list to the swap buffer because of concurrency issues introduced by the cooperation. This is a problem in the multithreaded merging of the cooperation algorithm as there will be two threads writing to and reading from the list at the same time during the merge. By having the entire list in the swap buffer both threads can safely write to their own half of the merge section while remaining sure that the data read from the swap buffer is not modified by the other thread.

To conclude the optimization of the kernels the choice of workgroup size can largely dependent on the hardware and how the algorithm is designed. The use of local or private memory is in most cases be a good way to minimize the amount of reads and writes to global memory. The biggest improvement that can be achieved in most cases is to raise the degree of parallelism if it is possible.

4.4.2 Image processing

Using image convolution with filters we can manipulate an image using a constant algorithm to achieve a plethora of different outcomes based almost entirely on the filter being applied. Because of this potential diversity almost all modern image manipulation software these days allows the user to apply these types of filters to their images, this allows the users to instantly and drastically alter their images in a uniform manner. To demonstrate this technique we present an image passed through a 5x5 box blur filter in figure 6.

4.4.2.1 The basics of image convolution

There are a multitude of ways to store images in memory depending on the basic format used, which kinds of compression techniques are applied etc. As image compression and storage is not the focus of this report the following explanation of image convolution with filters will assume that images are stored as an array containing pixels with each pixel containing a red, a blue and a green color value ranging from 0 to 255.

```
for(each pixel)
{
  red = 0;
  green = 0;
  blue = 0;
  for(each pixel in the filter matrix)
  {
    red += filterPixel.red * correspondingFilterValue;
    blue += filterPixel.blue * correspondingFilterValue;
    green += filterPixel.green * correspondingFilterValue;
  }
  newPixel.red = red;
  newPixel.green = green;
  newPixel.blue = blue;
}
```

Listing 4: Basic image convolution pseudocode

As described in listing 4 above, to perform a basic image convolution one iterates over every single pixel of an image and for every iteration the red, green and blue color values of the pixels surrounding it are aggregated and will be used as the color values of the corresponding pixel for the new filtered image. This, the act of calculating a new color value based on the value given in a filter matrix multiplied by the corresponding surrounding pixels color values is the essence of image convolution with filters [28].

What quickly becomes apparent after one understands the basics of this algorithm is that a straightforward sequential implementation will be done with four nested for loops, rarely a sign of a quickly executing algorithm. An observant reader comfortable with multi-core programming might also soon realize that the pixels for the new filtered image need not be calculated in sequence to produce the same result, which is why this type of algorithm lends itself so well to a GPU-accelerated approach.

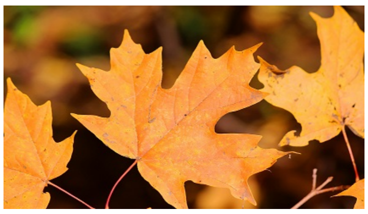
For the sake of simplicity there are some notable omissions and simplification in this section that would need to be accounted for in an actual implementation of this algorithm. Among these we find such things as:

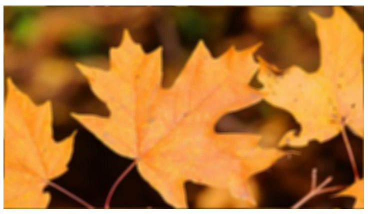
- Image constraints, how do we calculate the color values of pixels that are outside the image boundaries i.e. pixels that do not exists.
- Newly calculated color values that exceeds the minimum(0) or maximum(255) allowed color values.
- The brightness of the new pixels and by extension the whole new image will depending on the filter used often be significantly higher or lower than the original image.

The image boundaries problem is most efficiently dealt with in one of two ways, either you implement a check to see if the currently selected pixel in the filter is outside of the image before its color values are added to the calculation of the new pixel. We then either ignore these nonexistent pixels or use the color values of the next closest pixel e.g. a pixel with the position x=5 y=-2 would return the value of the closes existing pixel x=5 y=0. Because of a built-in feature in the OpenCL framework the latter option is used in our implementation of the algorithm.

To ensure that our new pixel only has color values within the 0 to 255 range we clamp them, first the maximum of their current value and 255 then the minimum of their current value and 0.

The issue of inconsistent brightness can be corrected by calculating the sum total of all the values in the filter and if this sum is greater than 1 we divide 1 by this sum and multiply the result by the new red, green and blue values, this should be done before the RGB value clamping.


(a) Original image


(b) Filtered image

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

(c) 5x5 box blur filter matrix

Figure 6: 5x5 box blur filter applied to an image (public domain image used)

4.4.2.2 Parallel image convolution

So far we have established that image convolution is a very compute heavy operation, exactly how long it takes to perform sequential and parallel image convolution with varying filter sizes and images of different resolutions will be thoroughly covered in section 5.2 of this report. How does one go about implementing a GPU-accelerated version of the same algorithm? In this section an implementation using the OpenCL framework will be presented.

As was briefly mentioned in the previous section much if not all of the compute heavy portions of the algorithm can be done in parallel. The OpenCL kernel code will not differ much from the code of sequential implementation but there are a few key points to be aware of.

- When queuing up a kernel in the host code the global workgroup size has to be a power of 2 e.g. if one wants to dispatch say 100 work-items the global workgroup size has to be at least 128. This means that an additional 28 unwanted work-items will have been dispatched.
- Operating system timeouts, several modern operating systems will automatically trigger a reset of the graphics driver if it deems your application to be hogging the GPU for too long.
- How to store an image using the OpenCL Image2D memory object. OpenCL features data-structures specifically designed to hold both 2D and 3D graphic objects for passing data to and from kernels.

Large workgroup sizes
The fact that the global number of work-items dispatch must conform to a certain scale, i.e power of 2 will likely not cause a significant decrease in overall performance if the number of work-items needed to perform an operation is relatively low. For example dispatching 32 work-items instead of 27 will probably have an unnoticeable performance impact. When however we start to deal with more computationally intensive kernels coupled with large global workgroups the overhead becomes greater. To help visualize this consider the image convolution of a 4K resolution image that is 3840x2160pixels, in an ideal scenario the NDrange object used to represent the global workgroup size would be initialized as NDRange(3840,2160). That is a total of 8294400 work-items, however those values(the width and

height of the image) are not a power of 2 so one must calculate the closest valid workgroup size. In this case this means initializing the NDRange object to a total of 16777216 work-items, leaving us with 8482816 unwanted work-items, more than double the amount needed. The most common way to deal with these unwanted work-items is the one employed in the kernel source code for our implementation, simply perform a check at the beginning of the kernel code to see if the current work-item actually has some work to perform and if it does not then exit the kernel immediately. This solution comes at a surprisingly low overhead cost.

System timeouts
The operating system's daemon process intended to make sure no one application excessively hogs the GPU to the detriment of the rest of the system can indeed cause problems for software developers wishing to unload computationally heavy work onto the GPU. The authors' experience of and solutions to this problem will be covered in section 6.5.1 which is dedicated to the issue of working around this background process.

Memory restraints
A problem that emerged after the basic GPU-accelerated implementation of the image convolution algorithm was with the processing of very large images. In order to provide as detailed an analysis as possible the authors had planned to compare the execution times of filters of varying sizes applied to images ranging in resolution from fairly small at 480p(853x480) all the way up to the very large 8K(7680x4320) resolution including the most common resolutions between the two extremes. For up to 4K(3840x2160) resolution images the application performed as expected, however the 8K images would not come out of the filtering process looking as expected.

After verifying that the fault did not lie with the algorithm or its implementation it was hypothesized that the unexpected result was a side effect of memory limitations of the OpenCL Image2D memory object used to store the images. While this did turn out to be the case it was not entirely obvious as to why. The OpenCL SDK comes with a handy application called CLinfo.exe which will provide the developers with printouts of very important information about their system including but not limited to preferred workgroup size multiple, maximum work-items, available platforms and devices etc. Among these important statistics one will find the maximum Image2D width as well as maximum Image2D height, both of these were reported as 16384. Looking at the OpenCL specification provided by Khronos [12] one will find that an Image2D object should support image resolutions of at least 8192x8192 pixels, as it turns out however the hardware vendors does not always comply with these numbers in their platforms implementation of the OpenCL framework. This in conjunction with the fact that the numbers shown in the CLinfo applications were incorrect meant that it took until a command for manually printing the systems maximum Image2D width and height was found until the actual limitation of our systems were discovered to be 4013x4014 pixels.

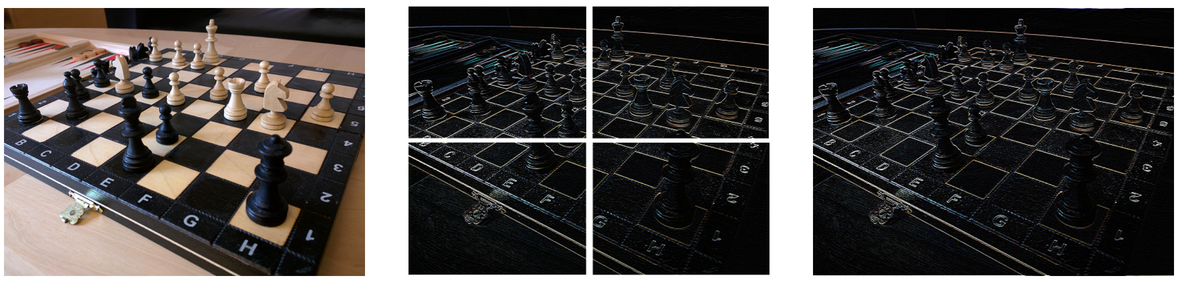
This problem was solved by having the application split the very large images into smaller subsections and perform the convolution on each piece separately before combining them back to a single cohesive image, the pseudocode for this process is presented in figure 5. This of course introduces some unwanted overhead but as becomes evident in section 5.2 Results of this report is still very much warranted.

```
if(image.width > 4000 or image.height > 4000)
{
  imagePart1 = Image(0  ,0  ,image.width()/2, image.height()/2);
  imagePart2 = Image(image.width()/2, 0, image.width(), image.height()/2);
  imagePart3 = Image(0, image.height()/2, image.width()/2, image.height());
  imagePart4 = Image(image.width()/2, image.height()/2, image.width(), image.height());

  applyFilter(imagePart1);
  applyFilter(imagePart2);
  applyFilter(imagePart3);
  applyFilter(imagePart4);

  for(each pixel in the original image)
  {
    replace with its filtered counter part from imagePart1,2,3 or 4;
  }
}
```

Listing 5: High resolution image splitting pseudocode

As shown in the pseudocode above, our implementation works around the issue of storing large images in a single memory object by splitting it into 4 equally large parts and performing the image convolution on each sub-image individually. As illustrated in figure 7 these four parts are then combined to create a complete filtered version of the original image.

An observant reader might realize that this solution coupled with the fact that as was described in section 4.4.2.1 color values for pixels outside the boundaries of the image are clamped to those of the closest pixel inside the image will likely cause artifacts in the new image along the edges of where the sub-images meet. To avoid this the actual implementation code creates the sub-images with 100 pixels worth of padding on the sides of the image that will be next to another sub-image. This padded section containing unwanted artifact will then be ignored during the combinatory step, resulting in a filtered version of the original image without unwanted artifacts.



(a) Original image (b) Split filtered image (c) Complete filtered image

Figure 7: Convolution performed on a very large resolution image (original photo by Justus Sörman)

5 Results

In this section of the report the authors will review the result and how the implementations performed on the hardware described in section 4.2. Some performance tests has been run on other systems with a more powerful GPU to show how much of a difference it can make as well as to allow for extrapolation for possible future work.

5.1 Merge sort

The total execution time of the merge sort algorithm differed a lot depending on the implementation and what hardware it was running on. The basic implementation that we described in section 4.4.1.3 and presented as pseudocode in listing 1 was not able to finish the computation in time before the TDR stepped in and reset the drivers for the GPU so the results from the basic kernel will not be present in this section. The setup time for the OpenCL framework will not be accounted for in the execution time of the different kernels partly because the setup must only be made once. This setup is not only very short compared to the total execution time it also remains the same for all the different kernels which is why the authors decided to omit this portion of time from the results. The last thing to mention is that all kernels described in this section uses zero-copy and no results for implementations without zero-copy will be presented. This is because the total execution time with and without zero-copy was not that different.

The normal chunking kernel and the optimized variant had very similar execution times when running on the CPU (about a 2% reduction in execution time for the optimized version with the largest list size of 16,777,216 elements). When running on the GPU the optimized variant was about 13-15% faster depending on the list size used, this can be seen in figure 8. The optimized cooperation kernels however had a significant performance increase compared to the normal chunking kernel. The speedup on the GPU was very impressive at about 112-115 % faster compared to the chunking implementation but the performance of the CPU with the cooperating kernel was about the same as the normal optimized kernel. The performance of the CPU can be seen in figure 9.

A modified version of the chunking merge sort where the amount of threads dispatched is the same as the amount threads needed to perform the computations was written. The intentions of this variant was to speed up performance by removing the overhead caused by dispatching more threads than needed. The early merges took longer and longer to execute until hitting a peak which was followed by a plunging execution time. This plunge was at a certain point in the merge sequence when the 512 threads were executing concurrently and the major peak when 16384 threads were executing concurrently. The execution time of this kernel is shown in figure 10 where different workgroup sizes were also examined.

The optimized variant with cooperation implemented has when compared to the normal optimized kernel a significantly faster execution time because the cooperation elevates the level of parallelism in the algorithm. Therefore execution is faster on the GPU where we can benefit substantially from the elevated level of parallelism while the CPU that is not very parallel at all does not gain as much from the cooperation optimization. In figure 8 the execution times of the different kernels are shown when executed on the GPU and in figure 9 on the CPU. The graphs takes into account different list sizes to show how the kernels scales when the workload goes up.

The execution times of the sorting on the GPU differed a lot depending on which implementation was used. The normal chunking algorithm performed well in the earlier merge sessions when the GPU was fully utilized but when underutilized the performance dropped significantly. This behavior can be seen in figure 11 along with some other kernels execution times. Otherwise the sorting algorithm scaled very well with different list sizes. We can see this represented in figure 8 where the execution times are compared to the list sizes.

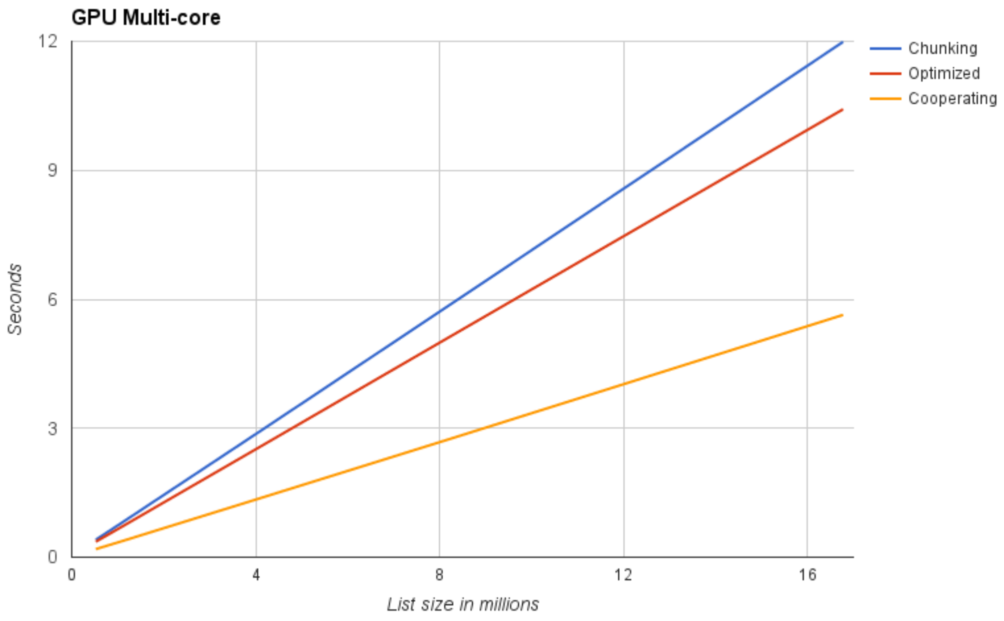


Figure 8: GPU multi-core execution times with different list sizes

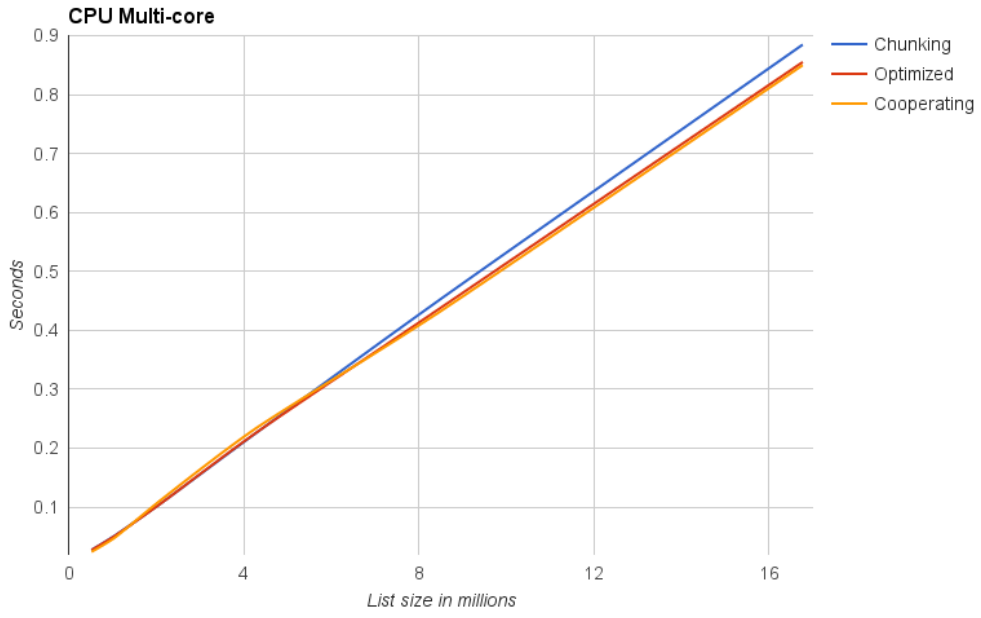


Figure 9: CPU multi-core execution times with different list sizes

The performance of the CPU followed the same pattern as the GPU when handling different list sizes as we can see in figure 9 and 8 but when comparing how they handled the different merge sessions of the algorithm the difference starts to show. The CPU is actually performing worse than the GPU in the first sessions of the execution due to not being able to handle the high degree of parallelism as well as the GPU. However in the later sessions when the number of elements that are being merged goes up the performance of the CPU goes up as well.

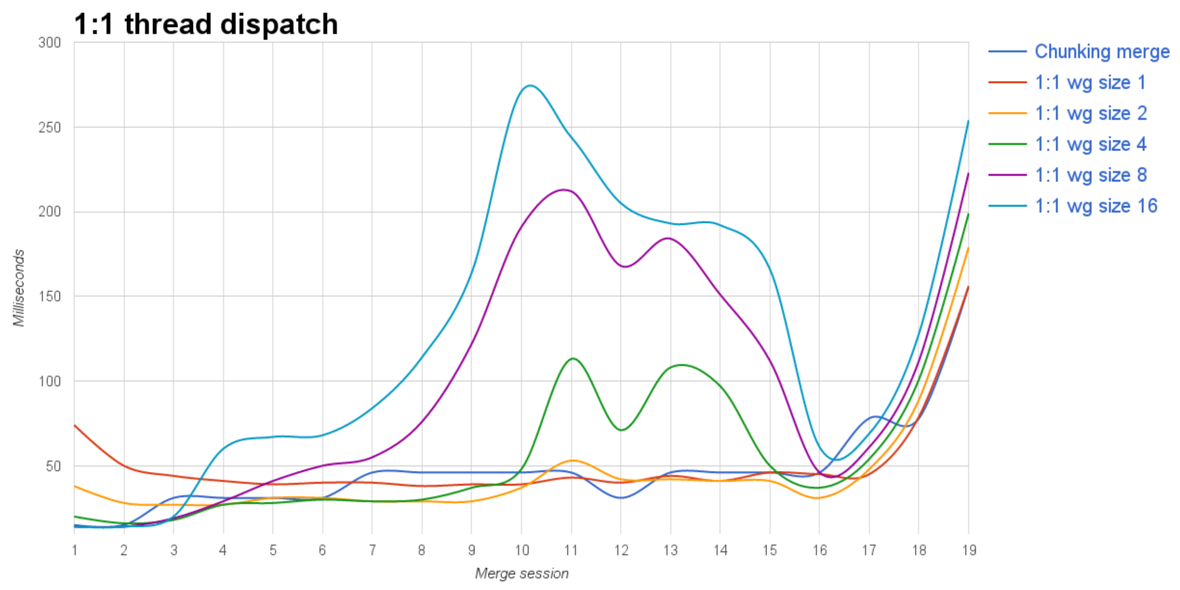


Figure 10: Performance test with 1 to 1 thread execution with different workgroup sizes

The hybrid solution where we use both the GPU and the CPU to perform the sorting is using one of the key features in OpenCL, which allows us to switch between the processing units. The hybrid approach is that we perform the sorting on the GPU when the number of executing threads is fairly large and when the GPU starts to get underutilized we start using the CPU instead to maximize the performance of each processing unit.

The performance of this implementation does not get a better total execution time than the multi-core CPU only execution as seen in figure 11. However if we only use the GPU for the first three merge sessions when it is performing better than the CPU and then switch to the CPU for the rest of the merge sessions we get a better result than the CPU for all merge sessions. This is because we use the GPU for the copying instead of the CPU which is performing better with this highly parallel kernel. If we compare the normal hybrid and the one that only uses the GPU for the first merge sessions they both have very good results compared to both the sequential and multi-core CPU.

To summarize, the GPU alone never beat the CPU in total execution time, not even when the CPU was only executing sequentially. The total execution times for the different implementations can be seen in appendix F. However the GPU during the merge sessions when it was fully utilized did beat the execution times of the sequential CPU during those same sessions. The multi-core CPU execution was the fastest when merging but not when copying. The hybrid kernel where the CPU merged and the GPU copied elements to the swap buffer performed better than the multi-core CPU only kernels even if the latter was using cooperation.

5.1.1 Sorting benchmark

The execution times of the sorting when performed with OpenCL varied slightly between test runs. Which is why during the benchmarking the same test was performed several times in order to get a representative average time. More than 100 test runs with all the different list sizes were performed. The benchmarks were done by having the main loop in the sorting program call the independent sorting functions with information about what device it was supposed to be running, the CPU and/or the GPU and if it was to use the optimized variant or not.

When the program had run the five different sorting algorithms 100 times each we divided the list

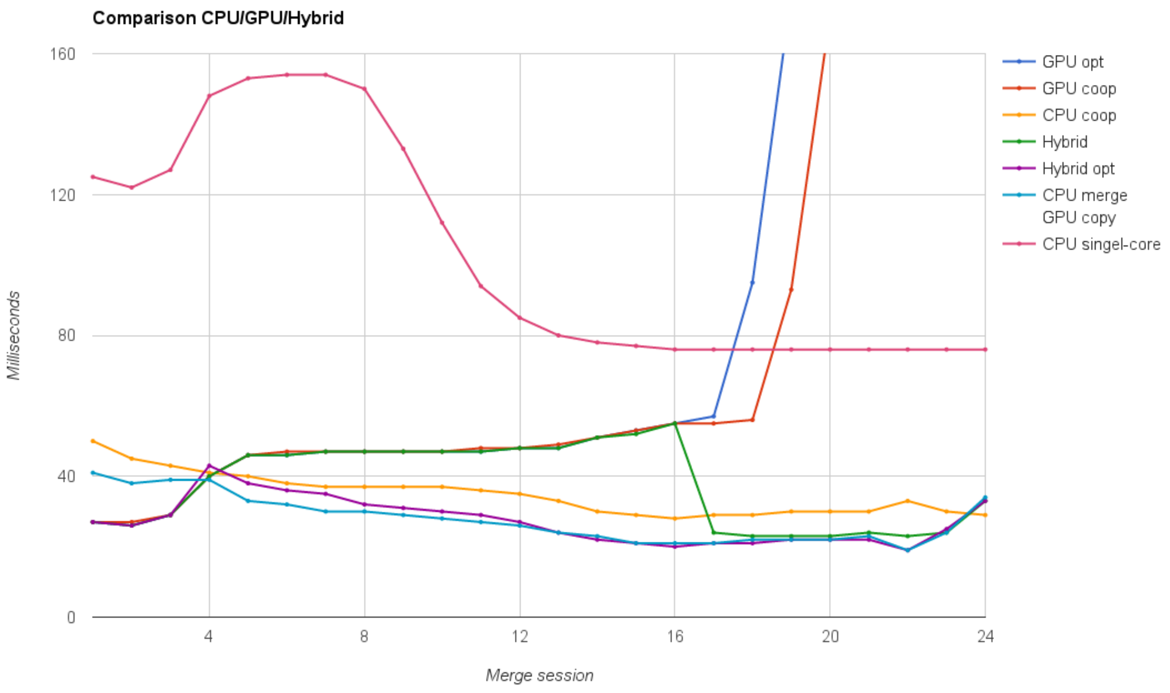


Figure 11: Comparison of CPU,GPU and Hybrid execution times in the different merge sessions

in half and re ran the test again with the new list size. This was repeated six times so the smallest list tested was only 1/32 of the original list size. The different list sizes were tested because we wanted to examine how well the algorithm scaled with different workloads. The total execution times were recorded as were the times of each merge session so that we could examine when the algorithm started to get inefficient and where it was efficient in its execution.

The largest list size on which the sorting was performed was 16,777,216 elements, this would result in the program taking up about 140 megabytes of RAM for both the list and swap buffer combined. This list size was chosen because the execution time was long enough to give us accurate benchmark result for all implementations. The total execution times that were recorded with this list size differed between 12 seconds to a little under 1 second depending on what processing unit(s) it was executed on.

5.1.2 Sequential merge sort benchmark

The benchmarking of the conventional sequential merge sort was performed in similar fashion with 100 sorting cycles for each list size. The sequential sorting on the CPU without the OpenCL framework did not fluctuate in execution time as much as the other benchmarks.

5.2 Image convolution

The performance of 5 different takes on an image convolution algorithm where measured and this section we will describe how well these implementations perform when compared to each other.

5.2.1 The benchmarking

Initial performance testing showed a propensity for significant variance in execution time when measuring the implementations for performance. To deal with this a script was written to execute each implementation with the same filter on an image of the same resolution one hundred times. To gain insight into if and when GPU-acceleration started to become viable this test was performed on all five implementations with five different sharpening filter matrices ranging in size from 3x3 to 11x11 on images of resolution 480p(853x480), 720p(1280x720), 1080p(1920x1080), 1440p(2560,1440),4K(3840,2160) and 8K(7680,4320). This benchmarking was performed once on two hardware/software wise identical systems. This made for a total of 30000 test runs which took several hours to perform on the systems described in section 4.2 this as will become evident further on in this section was in no small part thanks to the execution speed of the sequential implementation.

The tests measure time via the C++ standard library Chrono and the high_resolution_clock available therein, the test does not measure the time needed to configure OpenCL and building the kernels. This is because the time needed to do so was found in early testing to be negligible in all test but those with the smallest resolution images using the smallest filter matrices. This means that even when accounting for setting up the necessary OpenCL components every multi-core implementation still outperforms the sequential version. In an actual development scenario it is also not unreasonable to assume that the OpenCL setup would be done once at the start up of an application and then used multiple times before the application is shut down. For images larger than 4K in resolution the time needed to workaround both the TDR watchdog process and the limiting size of Image2D memory objects is included. That is the time needed to split up the image into four parts, apply the filter to each part and combine these parts back together.

5.2.2 Conventional sequential CPU implementation

This is an implementation of the basic image convolution algorithm described in section 4.4.2.1. This implementation serves as a baseline to which all other implementations are compared with the intent of answering the pertinent question "did we actually gain any increase in performance for the effort spent using GPU-acceleration?".

In table 1 the average execution times for the sequential CPU implementation is shown. Two things become very clear after reviewing these numbers, the first is that as expected the execution time does vary significantly depending on the size of the filter matrix. The second is that when one compares these results with those of the multi-core implementations, be it with the GPU or the CPU version we can see that they both outperform this sequential implementation

filter	480p	720p	1080p	1440p	4K
3x3	0.13762	0.3697	0.8278	1.7195	3.9876
5x5	0.31707	0.7560	1.7411	3.5143	7.8614
7x7	0.58897	1.3837	3.1234	6.1221	13.4873
9x9	0.97617	2.2955	5.0577	9.8043	21.7752
11x11	1.42849	3.3663	7.4389	14.2130	31.7865

Table 1: Sequential CPU execution times with varying filter sizes applied to images of resolution ranging from 480p to 4K

5.2.3 multi-core implementations

The basic multi-core CPU version is identical to the basic GPU-accelerated implementation except for the fact that the OpenCL setup part of the program creates the context with the CL_DEVICE_TYPE_GPU

filter	480p	720p	1080p	1440p	4K
3x3	0.0299	0.0608	0.1251	0.2205	0.4915
5x5	0.0368	0.0776	0.1595	0.2822	0.6202
7x7	0.0457	0.1006	0.2141	0.3783	0.8064
9x9	0.0609	0.1315	0.2810	0.4822	1.0435
11x11	0.0764	0.1684	0.3590	0.6166	1.3403

Table 2: Execution times in seconds on multi-core CPU without zero-copy

480p	720p	1080p	1440p	4K
0.0251	0.0479	0.1098	0.1909	0.4158
0.0311	0.0633	0.1448	0.2520	0.5456
0.4524	0.0898	0.1999	0.3418	0.7236
0.0570	0.1209	0.2649	0.4509	0.9717
0.0732	0.1569	0.3415	0.5841	1.2665

Table 3: Execution times in seconds on multi-core CPU with zero-copy

flag instead of the CL_DEVICE_TYPE_GPU one. They also have their threads dispatched to run the exact same kernel code. Both basic multi-core implementations has a local workgroup size of 16x16.

In an optimized version some minor changes has been made to improve the performance of the multi-core CPU implementation, most important of which is the use of zero-copy to avoid unnecessary overhead. The kernel code remains the same as it was in the basic GPU and CPU implementations. When analyzing the result of the benchmarking this small change consistently proved to provide a measurable advantage as shown in tables 2 and 3.

5.2.4 General results

To provide a good overview of how well the GPU-accelerated implementations fared against their CPU counterparts we provide two graphs that illustrates the average execution times of the different implementations. First on a 480p resolution image with filters of sizes varying from 3x3 to 11x11 in figure 12. This graph includes the execution time of the sequential CPU implementation which clearly deviates from the others, with the smallest size filter the sequential version is more than twice as slow as the slowest multi-core implementation.

Figure 13 illustrates the performance of the four multi-core implementations as it measures execution times on an 8K resolution image with filters ranging in size from 3x3 to 11x11. The execution times of the sequential implementation has been omitted in this graph simply because it would pull focus away from the other more pertinent implementations, for the sake of completeness these omitted execution times are shown in table 4.

filter size:	3x3	5x5	7x7	9x9	11x11
execution time:	15.9748	32.1769	55.9010	88.8090	130.220

Table 4: Sequential CPU execution times in seconds with varying filter sizes applied to 8K images

During the implementation stage when the earliest of tests where ran it was noted by both authors that using the system while running an OpenCL application could in some cases significantly affect performance. This could be a potential issue for developers wishing to deploy general purpose applications that makes use of OpenCL, a topic which is discussed in more depth in section 6.5. For the purpose of getting the most stable and unbiased results possible the benchmarking results presented in this section were gathered from a system where the benchmarking script was executed and then left uninterrupted for the duration of the tests.

The bulk of the performance testing was carried out on the systems described in section 4.2, this includes the over final 30000 tests runs that produced the data presented in this section. To verify that the results gathered during these final tests runs are viable for extrapolation a few thousand test runs were performed on systems with discrete graphics cards. The results of these tests are not included in this section as they are not truly relevant to this thesis work, although they did provide some insight into how one can optimize OpenCL for different hardware. This will be discussed further in section 6.5.

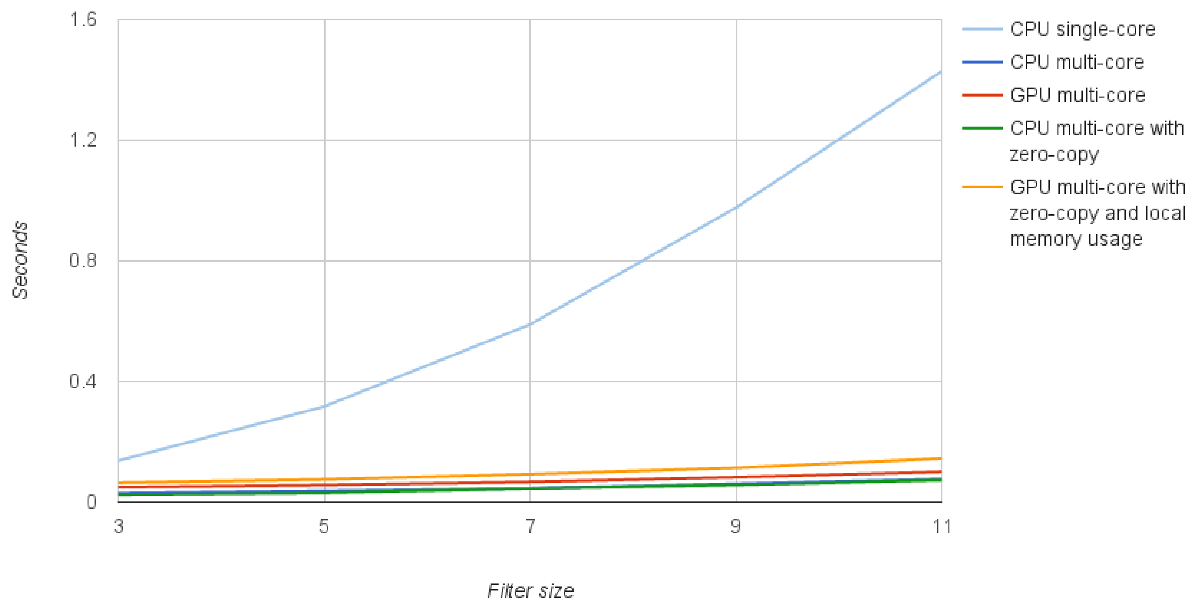


Figure 12: Average execution times of varying filter sizes on a 480p resolution image

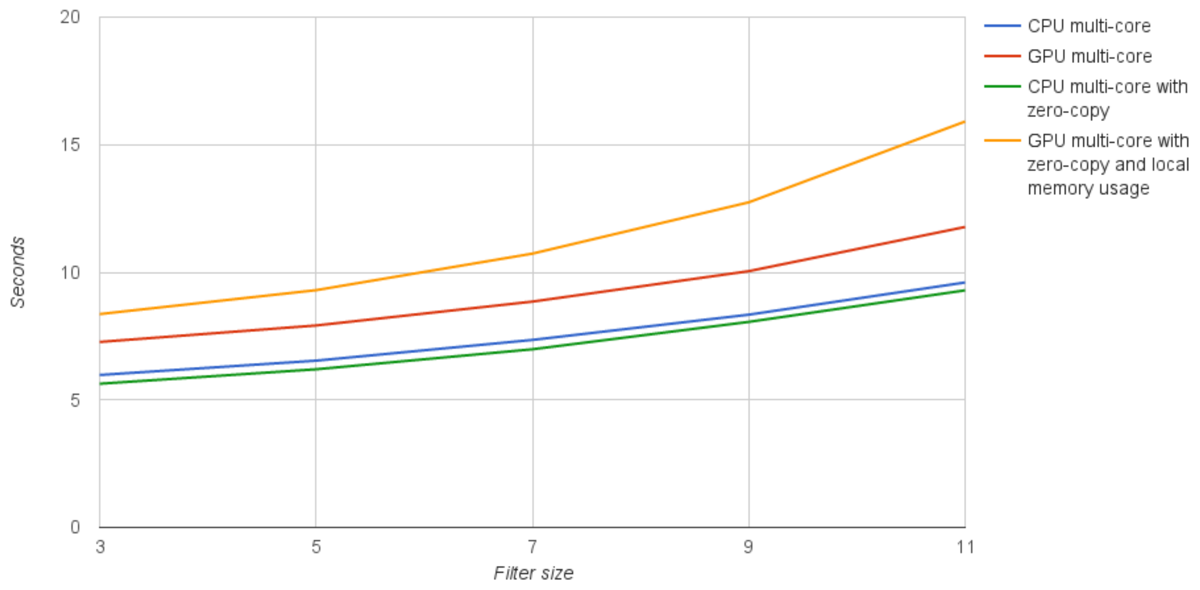


Figure 13: Average execution times of varying filter sizes on a 8K resolution image

6 Discussion

The results presented in section 5 will be analyzed and compiled, with the authors providing their reasoning and opinions as to why the data looks the way it does. The research question regarding the use of GPU-acceleration in general and the OpenCL framework in particular will be answered in this section based on the experiences gained by the authors during the literature study, algorithm implementation and result compilation and analysis.

6.1 Heterogeneous computation

The possibility to use both the CPU and the GPU together in a computation is a compelling thought. The implementation of the merge sort with both CPU and GPU executing parts of the code did not perform as well as we had anticipated that it would. The GPU performed at its peak when having more than 128 threads running concurrently and therefore we decided to use the GPU during the early merge sessions and then use the CPU for the later merge sessions. This because the CPU performed almost the same no matter which merge session it was in, although the early merge sessions are a little bit slower than the later ones. This algorithm is not task parallel so we could not run the CPU and the GPU concurrently. The performance of the hybrid version was faster than only running the GPU and slower than only running the CPU but not by much. This tells us that with a highly parallel workload the GPU if fully utilized is almost on par with the CPU when it comes to performance.

We realized fairly early on in the implementation process that the GPU would never perform better or even on par with the CPU when it came to parallel merge sort. We did however still anticipate that the GPU accelerated image convolution would prove to be more comparable. This is because of how well image convolution lends it self to a parallel implementation as have been mentioned many times throughout this report. As it turns out however not even the inherent parallelism of image convolution could make the GPU perform better than the multi-core CPU version.

If one has an algorithm that is task parallel then using OpenCL to make use of both the CPU and GPU in the computer system to perform the needed calculations will often provide some sort of performance increase. Depending on the task this speedup can vary from almost none to very high if there are specific parts of the tasks that are beneficial for the different types of processing units.

6.2 Results

The result of parallelizing an algorithm is not always what one might expect it to be. In most cases performance is gained as long as the overhead of conditional branches are minimized and the algorithm is well suited to be run in parallel.

6.2.1 Merge sort

The performance differences of the merge sort when optimized and not optimized were not that big or nonexistent when the workload was small. This is because the performance impact of the optimizations were so small as a result of none of the standard optimization techniques having any sort of positive impact on the performance. Although this might be because all the optimization techniques tested are designed for discrete graphics cards and not integrated ones. The Intel HD Graphics 4600 have some inherited caching hardware from the CPU that likely solved most of the read and write bottlenecks that usually occur when writing to a kernel. The only optimization technique that actually did have some impact was zero-copy. By not doing unnecessary copying from the host memory to a dedicated GPU memory buffer we reduced the memory consumption of the program by almost half but the execution times of the work-items was not significantly affected by zero-copy.

One optimization that was implemented was the usage of private memory caching. This optimization reduces the number of reads and writes to global memory. This was implemented by storing a small part of the list in the work-items own private memory, not the local memory because the private memory is faster and there was no need to share data between work-items. The cache size is preferable set to a multiple of 16 elements (64 bytes) because the hardware has a memory bus that supports this size of reads and writes in one clock cycle. Therefore when we fetch a part of both the swap buffer and the list into the kernel we fetch at least a multiple of 16 elements from each. Thereby filling the memory bus to its full width which is useful for hiding memory latency from the global memory. The same procedure was done when writing back a part of the list that we stored in private memory. This implementation removed some of the unnecessary reads and writes to global memory but inserted some minor overhead with conditional branches.

The performance after these optimizations was actually worse than without them. This is probably due to the sequential reading and writing to global memory that has a high chance of being cached by hardware caches. Therefore the caching from the lists was not enough to speed up the execution and the overhead bogged down the performance. Another attempt to optimize the kernel was to only use a small cache for the resulting list and write it back to global memory in batches of 64 bytes. This is generally a good idea but in our case inserted too much overhead and we did not gain enough by utilizing the bus to a higher degree.

The final optimized variant only has some arithmetic optimizations that were fairly simple to implement. Unexpectedly the overhead of conditional branches had a much bigger impact on the performance than we first thought when implementing the algorithm.

An implementation that was expected to have a large performance impact was attempted, merge sort with cooperation. This cooperation technique described in section 4.4.1.4 was used when running one of the multi-core non-hybrid implementations i.e. when using only a single device. This optimization was a great success when using the GPU that thrives on having a more parallelized workload. The CPU on the other hand did not see this major performance increase. This is due to the fact that the CPU is not a very parallel processing unit with only 4 physical cores. This means that it can only benefit from having this cooperation in very few of its final merges unlike the GPU which can utilize this technique for many merges.

The hybrid merge sort presented in figure 11 is a good comparison between the CPU and the GPU because it shows where during the sorting process the two devices performs at their peaks. Unfortunately the GPU could not keep up with the CPU even when the workload was highly parallel and optimized for the GPU. The performance of the optimized hybrid implementation is very close to the one using only the CPU as is illustrated in figure 11. The reason why the hybrid merge sort is faster than the CPU when both are actually using the CPU to merge is because the hybrid sort is using the GPU for the copying to the swap buffer which the GPU is much faster at doing than the CPU. Therefore the hybrid implementation where the GPU is used for copying and the CPU for merging is faster than the implementation where the CPU is responsible for both tasks.

An optimization that took the problem of launching a lot more threads than we needed into consideration was a fairly easy optimization. The only changes that were needed to be performed on the already existing kernel with chunking was the change in how the kernel found out which elements it was supposed to iterate over. This range we find by multiplying the work-item's global thread id with the size of the merge section. Another thing that needed changing was the host code so that the host only launches the correct number of threads, this number is the list size divided by the current size of the merge section. The performance of this kernel again was unexpected, we thought that this would increase the performance of the kernel by skipping the overhead of many threads just initializing their variables and then realize that they are not going to perform any merging and return. The performance was much worse than the arithmetically optimized kernel when we had an workgroup size of 16. The odd thing here is that when we minimized the workgroup size down to 1 the performance was almost on par with the arithmetically optimized kernel. A workgroup size of 1 is in almost all cases one of the most under performing workgroup sizes that you could choose. The reason behind this behavior the authors could not figure out because the hardware should have a much better performance when having a workgroup size of at least 4 as each compute unit has 4 ALUs. The most likely answer as to why the hardware acts this way with the kernel is that the scheduler gets some sort of hiccup when the threads wants to execute in this way.

The performance scaling on both the CPU and GPU was surprisingly linear when sorting with different list sizes. This is probably due to merge sort only needing a total number of comparisons of $\log_2(\mathrm{n})$ per list element which is not growing very fast compared to the size of the list.

The setup overhead of OpenCL is not that significant when compared to the execution time of the whole algorithm. As with all the OpenCL implementations the execution time differed some, the minimum setup time was 0.11 seconds and sometimes the system got an hiccup and the setup time spiked to 0.34 seconds, the average setup time was 0.115 seconds. These setup times are not included in the performance numbers in the result section because they are so small and if one were to reuse the kernels and buffers that is already declared then the setup time is close to zero.

6.2.2 Image convolution

The main focus of the image processing portion of this thesis work is on the GPU-accelerated implementation of the image convolution with filters algorithm. The host code begins with standard boilerplate code to setup the OpenCL framework. The image is then loaded into memory and converted into a format preferred by the OpenCL Image2D memory object, an array of cl_float4s. One element in this array represents a pixel with red, green, blue and alpha color values. The Image2D memory object is then enqueued to the command queue, the same goes for a 1D array representation of the filter. The kernel is built and all its arguments are assigned before being enqueued to the command queue. The image is then read back via the enqueueBufferRead command into a buffer containing an array of suitable size. At this point we force the command queue to synchronize via the finish command. Now with the completed filtered image stored as a 1D array of cl_float4s we convert them back into an image. Since we are working with a 2 dimensional image we use a 2 dimensional NDRange object when enqueueing the kernel to the command queue. Doing so allows one to fetch a given threads global x and y values inside the kernel code, these values are then used to represent a single unique pixel in the image. this also allows us to check if a thread is representing a non-existing pixel i.e one that is out of bounds of the image. If that would be the case the thread simple exits the kernel to avoid doing unnecessary work.

6.2.2.1 Optimizations

In an attempt to further increase performance an application based on the existing host code was developed. this new version utilizes zero-copy, a concept discussed earlier in section 3.5.1. The basic image convolution kernel accesses the image to which the filter will be applied as a function parameter to the kernel in the form of an Image2D memory object. This image data is then stored in global memory.

When performing the convolution with a 3x3 filter matrix every single work-item thread will make 9 access calls to this shared global memory. As the filters grow in size so will the amount of accesses to global memory, with an 11x11 filter performing 121 memory accesses during the convolution stage of the kernel. In section 3.4.6 the authors detail the different types of memory available to work-items, with global memory being the least error prone and simplest to manage though slowest in terms of memory access speeds.

A new version of the convolution kernel was developed in an attempt to minimize the amount of global memory accesses. This was achieved by having each work-item in the 16x16 workgroup fetch 4 pixels from global memory and then cooperatively create a 32x32 pixels large image that they all share in their local memory. The though behind this approach is that the image convolution part of the kernel never has to access the global memory. When applying a 3x3 size filter matrix each thread would then make 4 access calls to global memory and 9 to the local memory. This was not expected to provide significant performance gains, however for a 11x11 size filter matrix with only 4 access calls to global memory instead of 121 noticeable execution time speedup was anticipated. The decision to make the local image 32x32 pixels large was made to avoid any potential artifacts at the edges of the sub-images represented by the workgroups' local images in the final complete image.

6.2.2.2 Results

When analyzing the results presented in section 5.2.2 and in figure 12 in particular there are a few obvious conclusions to draw. Firstly if there was any doubt before about the parallelity of the image convolution algorithm seeing that even the smallest image with the smallest filter applied the sequential CPU version is handily outperformed by all other implementations should be enough to convince you otherwise. We can clearly see how poorly the sequential CPU implementation scales with increased filter sizes. In contrast the parallel implementations all scales better with increased filter sizes.

Secondly we can conclude that the multi-core CPU implementation always performs better than its GPU counterpart, looking at figure 13 we can see that even with the GPU fully utilized for a long period of time it still can not compete with the CPU. We came to this conclusion fairly early on in the testing process, the integrated Intel HD Graphics 4600 GPU is simply less powerful than the Intel i7-4790 CPU. The GPU did however have execution times many times faster than the sequential CPU implementation meaning that there is a definite possibility of being able to find algorithms suitable for a GPU-accelerated approach.

6.2.3 Caching in local/private memory

One of the most common optimizations to OpenCL kernels is the use of local or private memory to reduce the amount of reads and writes to the slower global memory. In both the merge sort and the image convolution kernels caching to the local or private memory was implemented but neither of these optimizations yielded any performance increase. This left the authors somewhat perplexed as the local and private memory is supposed to be significantly faster than the global memory so why did the implementations utilizing local and private memory not produce any performance increases? The answer could be that this type of optimization is one of the more complex ones to do properly. The use of the local or private memory for caching is hard to implement without introducing too much overhead and the same goes for the usage of IF statements in kernels it is therefore recommended to not overuse them. A branch miss on the GPU is very costly compared to one on the CPU.

6.3 Method

The most obvious limitation of our method as described in section 4 is the fact only one form of image processing and one method of sorting is evaluated. We believe that the results are suitable for general extrapolation of GPU-acceleration and the basic strengths and weaknesses of the OpenCL framework. Although one could argue that more research on many different kinds of algorithms applicable to a broader range of problems should be assessed before answering the research questions posed in section 1.3. The main reason behind limiting ourselves to these two types of algorithms is as with most thesis works a lack of time. In section 7.3 and 6.1 report we describe the heterogeneous approach, developing applications that simultaneously makes use of both the CPU and GPU. After this thesis work we can conclude that were we to redo this thesis work we would have put a larger emphasis on this sort of hybrid implementation.

Information about the systems on which the applications were developed have been presented in section 4.2 and 4.3, these sections contains detailed information about both the hardware and software employed during this thesis work for both development and benchmarking. Provided as appendices to this thesis report is the kernel code used by the OpenCL framework when running the applications. The host code used is in most cases omitted from this report, the reason for this is that this code does contain some boilerplate setup code for the OpenCL framework and also the fact that the actual application code has little impact on the execution time of the application. The basic theory behind the OpenCL framework and its components has been described in section 3.4 and their use has been described in the implementation sections of this report. With the above mentioned facts in combination with the rest of this report we are confident that any one attempting to replicate our applications will do so with great success.

As our performance tests has been run on multiple systems of the same specifications mentioned above with little to no discernible differences in results we believe that similar results would be had if one tried to replicate our performance tests.

The sources cited throughout this thesis report was for the most part selected during the literature study with a few exceptions added during the implementation stage when unforeseen gaps in knowledge became apparent e.g. Windows 7 timeout detection and recovery of GPUs daemon process. As many as possible of the sources cited comes from what we have deemed to be credible journals when dealing with papers and reports and, in the case of books ones that come highly recommended by the multi-core/OpenCL development communities. Also cited are several websites, among these are the Khronos group which has to be seen as the authority on the OpenCL framework, we also cite Intel and their specifications of the Intel HD Graphics 4600 GPU and the Intel Core i7-4790 CPU. Our thesis work is based on the assumption that the information provided specifically from these first party sources is correct and up to date. Some of the sources found in the reference section of this thesis report might not have an obvious connection to the research questions that we attempt to answer. In these cases the sources are used to provide an overview, both of why multi-core processors and GPU-acceleration exists as well as how they function.

6.4 The work in a wider context

This study has shown that the integrated the Intel HD Graphics 4600 GPU can compete with the Intel i7-4790 CPU if used correctly. In light of these findings and as an attempt to answer our third research question regarding when GPU-acceleration is applicable to software development we will give an example of one of the possible uses for GPU-acceleration. Distributed computing, distributed computing is the process of taking a workload that for various reasons can not be computed locally on a single computer or even a small cluster and distributing it onto a network of participating computers.

There are currently hundreds of projects utilizing this distributed computing model, the most successful of which are research projects that distributes a massive computational workload onto a network of participating volunteers [29]. A prominent example of these research projects is the folding@home project at Stanford university, they utilize distributed computing to simulate protein folding in an effort to understand and find cures for various diseases such as Alzheimer, Huntington's, Parkinson's among others. In this case the workload is not computed locally because of the prohibitively high cost of hardware and electricity as well as the complications involved with heat generation of large server farms [29]. It is estimated that this distributed approach allows the project to compute a workload 100 to 1000 times higher than could ever be done locally at a fraction of the cost [29].

The folding@home project currently have over 400.000 participating volunteers, with each volunteer computing a small portion of the total workload on their own private systems, these volunteers can choose to perform the computations on their CPUs, GPUs or even on their Playstation 3 consoles [29]. Out of these options the GPU has the most available raw compute power when measured in TFLOPS but most volunteers do not use their GPUs for these computations. One of the reasons for this is the varying quality of the client programs and the lack of support for individual graphics cards. Developing general purpose computational applications for the GPU has been a real obstacle as one would often have to support the many available GPUs on an individual basis[29]. This is an area where a framework such as OpenCL could really shine by providing a unified platform that would allow for a single application to support the majority of all computational devices used by the volunteers.

6.5 The OpenCL framework

An important aspect of this thesis work that has not been directly addressed thus far is the one of when GPU-acceleration in general and the OpenCL framework in particular should be used in software development. The pros and cons described in this section as well as the conclusions drawn by the authors will be based in its entirety upon the authors' own empirical experiences acquired during the research and implementation stages of the thesis work.

This thesis work started with the expectation of there being significant performance gain to be had by utilizing GPU-acceleration. Even on the less powerful GPU described in section 4.2.2 performance more or less equal to that of the CPU was more or less expected, causing the authors to plan for the majority of time to be spent in the implementation stage of the work optimizing and refining the basic implementation. As it turns out however using the Intel HD Graphics 4600 GPU to compete with the Intel i7-4790 CPU proved to be a tall order. Plainly put in our workstations the CPU is much more computationally powerful than the GPU.

The GPU-accelerated merge sort implementation even with a significant amount of time and effort spent on optimization could not truly compete with the standard sequential CPU implementation with an execution time almost half as long. This does not necessarily mean that the conclusion to draw here is that GPU-acceleration is not worth the time and effort to implement, the execution time while almost double that of the single-core CPU approach is still decent. If a situation occurs where the developer knows that the CPU will be fully utilized, getting the desired results even at twice the execution time might be more desirable than adding additional workload to the already busy CPU. In general though the real takeaway from the merge sort implementation is that any software developer should carefully consider how well the computational work that he or she needs to do can be executed in parallel. What seemed like a fairly good fit in the merge sort algorithm proved to have a significant bottleneck that was not feasible to workaround as well as having to be specially designed as not to incur the wrath of the Windows 7 TDR watchdog process.

The image convolution algorithm fared better in the sense that even on the smallest resolution image (480p) with the smallest filter matrix (3x3) the basic implementation outperformed the sequential CPU implementation by having an execution time almost three times faster. This was of course very encouraging as one would expect the GPU-accelerated approach to be faster at convolution with very large resolution images and very large filter matrices but perhaps struggle to perform competitively with the single core CPU implementation when dealing with smaller images and filters. Again proving that the type of computational work to be done is a very if not the most important factor to consider when utilizing GPU-acceleration.

A common theme that both of the authors encountered during the implementation and benchmarking stages of the work was that of the incredibly fast multi-core CPU approach. Using the CPU to execute the same kernels as the multi-core GPU it was established that the multi-core CPU implementation outperformed both the single-core CPU and multi-core GPU versions. This coupled with the fact that the OpenCL framework after some minor additions to the setup code on the host allows for dispatching of threads to execute the kernel code on both the CPU and the GPU interchangeably adds a great amount of flexibility for developers. Work could potentially be offloaded from the CPU and be assigned to the GPU dynamically or a heterogeneous approach like the one described in section 6.1 could be employed. It is also easy to imagine a scenario where the OpenCL framework is employed without the intention to utilize GPU-acceleration at all but just to implement a multi-core CPU algorithm.

One caveat to the benefits of using the OpenCL framework is that the applications produced might differ significantly in execution time depending on the system it is run on. This is because optimizations that provides increased performance on one system might result in a performance decrease on another. This was experienced during this thesis work when the merge sort implementations were tested on systems with discrete GPUs. On some systems it performed as expected, that is the more powerful discrete GPU outperformed the integrated Intel HD Graphics 4600. On another system with a discrete GPU however the execution time was markedly worse than on Intel's APU.

The conclusion to draw from this is that OpenCL works well when the target audience and the capabilities of their systems are well known and fairly uniform. In a similar vein if the target audience is the general public their system specifications will vary significantly. As discussed in section 3.4 different hardware vendors supports different versions of the OpenCL framework with Nvidia currently supporting version 1.1 on their discrete GPUs. This means that unless the target systems specifications are known beforehand an application targeting a wide audience would have to be developed using the lowest common denominator, that is version 1.1 of the OpenCL framework.

6.5.1 Operating system timeouts

For larger more complex kernels the execution time can be of considerable length. This does generally not hamper the application if the system running it has a discrete GPU dedicated to computational work and process acceleration. If however the GPU is providing output in the form of graphics rendering to one or more monitors the extensive execution time could prove quite problematic. The reason kernels with long execution time often prove to be problematic in a real world scenario is that today's operating systems carefully keeps tabs on the hardware in order to prevent the system from freezing or crashing. For the average user this is probably a good thing as it provides a more consistent and less frustrating user experience but for software developers it introduces restriction on their applications. An example of this type of restriction that was encountered early on in the implementation process was a Windows 7 watchdog/daemon process named "Time out Detection and Recovery" or TDR [26]. In essence what this process does is monitoring the amount of time the GPU takes to execute a given task and if it exceeds an operating system defined limit it will determine that the process executing on the GPU has timed out and reset the graphics driver. In Windows Vista and later iterations of the Windows operating system this timeout limit is set to 2 seconds [26]. It should be noted that this timeout limit applies to a single work-item and not to the application as a whole. Both authors of this report encountered TDR in their efforts to implement their respective algorithms. For the image convolution algorithm using very large filter matrices triggered the graphics driver reset and for the parallel merge sort the final few merges of very large lists proved time-consuming enough to activate TDR. When the graphics driver is reset during the execution of an active instance of our processes it effectively crashes the application, an unacceptable outcome for any software developer. This is an obstacle that has to be considered and dealt with for any GPU-accelerated software that does not exclusively target platforms where a dedicated accelerator card is guaranteed.

Several possible solutions were considered and tested in order to get around this problem, the first of which was to limit the number of simultaneously executing threads to a maximum of 128. This number was not arbitrarily chosen but based on the fact that the Intel HD Graphics 4600 supports 140 hardware threads. We had anticipated that imposing such a restriction would serve as an acceptable workaround, trading some performance in exchange for a guaranteeing that the GPU would always have some threads available and thus never timeout. As it turns out OpenCL will actively override manual attempts to limit the number of simultaneously executing kernels and try use all available resources and thus block the GPU, meaning that this approach was not a viable solution to our problem.

After consulting the available documentation a second solution was considered, by using an OpenCL concept known as device fission a device can be split into sub-devices. If some of these sub-devices were to be left unused by the applications it would likely leave the GPU with cores available for graphics rendering and solve our timeout problem. Sadly it was discovered that the Intel HD Graphics 4600 does not support this particular feature of the OpenCL framework meaning that this also was not a viable solution. The lack of hardware support for certain OpenCL features also implies that software developers should have a firm grasp of the types of systems employed by their target audience in order to not rely to heavily on framework features that might not be available on all platforms.

It should be noted that it is possible to manually override this 2 second timeout limit by editing certain registry keys in the Windows operating system, so although it is not impossible to workaround this issue we decided against manually editing these registry key values in effort to stay as close to a real life scenario as possible. A scenario where having our program changing these values would be at best seen as intrusive and at worst as malicious. Microsoft also specifically states that these values should not be edited outside of targeted testing and application debugging. In the end we came to the conclusion that the only truly effective way to guarantee that the TDR watchdog process would not cause problems is to significantly reduce the execution time of our work-items.

7 Conclusions

This section will contain the conclusions made by the authors about their respective algorithm implementations based upon their software development experience with the OpenCL framework during this thesis work. By its very nature any thesis work will be limited in scope, this fact will inevitably leave authors with unanswered questions and ideas that were never tested either due to time constraints or because they were deemed to be outside of the scope of the research question the work is based on. We will present some of these ideas and unanswered questions in section 7.3.

7.1 Merge sort

The solution to the problem with the diminishing numbers of merge sections in the later merge sessions is to implement the full cooperation algorithm when the number of merge sections becomes less than the number of available processing elements on the GPU. The only catch to this algorithm is that it is very complex and hard to implement. The execution times of the early merge sessions using the full cooperation implementation will be a little slower than those of the merge sort with chunking implementation but it will be a lot faster once the latter is unable to fully utilize the GPU. This implementation on the hardware we are testing on would likely not yield enough of a performance increase to beat the multi-core CPU but it would probably beat a sequential implementation.

The optimal execution strategy we found to be running the first sessions on a highly parallel GPU and when the number of processing elements in the GPU exceeds the number of merge sections a faster CPU with fewer cores could take over the execution. This strategy only works if the GPU is actually faster at merging the smaller sublists than the CPU. This was the case of our study but only in the first couple of sessions because the high powered CPU became faster compared to the GPU as the length of the merge sections became greater. However when using the GPU for the copying to the swap buffer and using the CPU for the merging the performance is actually better than that of the CPU only implementation. This is because the CPU is much slower than the GPU when performing the copy as seen the figure 11.

To answer the question of if it is viable to use the integrated Intel HD Graphics 4600 GPU for speeding up the execution time of a merge sort and offloading the CPU we unfortunately found that the GPU alone will not have enough compute power to speed up the execution time for this type of algorithm. But if we use both the GPU and CPU to execute the merge sort we can gain more performance out of the system than we could have using only the CPU. So yes we can say with certainty that a hybrid solution can gain performance by using the GPU in a smart way to handle smaller more efficiently than the CPU could.

7.2 Image convolution

In section 4.1 we described the image convolution algorithm and why we believe it made for such a good candidate for GPU-acceleration. Looking at the data resulting from our benchmark testing we can conclude that we were right in this assumption. It may have been overly optimistic to think that the integrated Intel HD Graphics 4600 GPU could perform competitively compared to the Intel i7-4970 CPU but as indicated by the data the vital portions of which are presented in section 5.2 it was not that far-fetched after all. All things considered the GPU performed admirably and showed what we believe to be satisfactory enough execution times to warrant further exploration and demonstrated that there truly is viable computational power going unused even in systems with comparatively weak GPUs.

The implementation stage of this thesis work in general and the optimization process in particular provided great insight into the OpenCL framework and its place in software development. If the application being developed performs a lot of computationally intensive work one should ask one self if the work could be done in parallel as described in section 3.3.1 and 3.3.2. If the algorithm could fit into one of those two camps then the next question should be GPU or CPU, how to decide between the two is dependent on how the application is to be deployed i.e. what is known about the intended user base. What kind of hardware will they have and how important is it to maintain a consistent execution time, as was discussed in section 6.5 the GPU-accelerated implementations tend to have a significant variations in execution times depending on the actions of the end users during run time.

7.3 Future work

After having completed the implementation, compiled and analyzed the results what immediately comes to mind as an area for future work is a more heterogeneous approach to parallel algorithm implementation. This sort of hybrid approach did receive some attention in the sorting portions of this report with some interesting results. These results as presented in figure 11 show that dividing the workload over both the CPU and the GPU can make for a viable implementation. In this case deciding when in the implementation to switch from the GPU to the CPU was quite straightforward, as was seen in figure 11 there is a point when the GPU implementations takes a huge performance hit. Depending on the type of algorithm that is to be implemented this breaking point might not always be as static as in the hybrid merge sort example, being able to dynamically determine when to switch from one device to another could prove to be of tremendous importance in the future if one wishes to achieve peak performance.

As a possible example referencing the work performed in this thesis report it could be possible to take a similar approach to that of the image convolution performed on very large images. In this sort of convolution as was described in section 4.4.2.2 the image is divided into several parts which are then individually filtered before being joined back together. In this thesis work this was not done with the intention to improve performance but as a means to workaround the width and height limitations of images inherent to the GPU and OpenCL platform available to us. One potential method of increasing performance could be to apply the same type of logic i.e. image splitting and performing small individual filter convolutions although modified to dispatch some portions of the workload to be done on the GPU and others on the CPU. As became evident in the benchmarking comparison of the GPU-accelerated and the CPU multi-core versions presented in section 5.2 and illustrated in figure 13 the two versions does differ in execution time by several seconds. Because of this dividing the workload at a straightforward 50:50 ratio is likely not the best solution, again hinting that a dynamic approach for workload allocation which is able to take the system's specifications into account is the way forward with heterogeneous computing.

7.4 Final thoughts

One of the main points that the OpenCL framework has going for it is widespread support amongst the major hardware vendors, although the degree to which they support OpenCL varies. After the initial learning curve that comes with any new framework or library we both found OpenCL to be fairly straightforward when it comes to the basic implementation of data parallel algorithms. We where also pleasantly surprised by the ease of which OpenCL allows for switching between the GPU and the CPU, allowing for smooth heterogeneous implementations.

We found that a less powerful integrated GPU most definitely can be used to improve the performance of compute heavy image processing algorithms. Our performance evaluations show that when comparing a basic sequential CPU implementation to its GPU-accelerated counterpart the latter will always outperform the former. Though the multi-core CPU version still reigns supreme it was proven to our satisfaction that the GPU, even if integrated will have a role to play in general purpose computation.

The answer is a lot less clear-cut when it comes to the question of using GPU-acceleration to improve the performance of sorting algorithms. The type of sorting algorithm implemented during this thesis work was merge sort, at first glance a good algorithm to parallelize but later showed itself to have major bottlenecks in how much it could be parallelized during the later merge sessions. Therefore the implementation of an hybrid solution that takes this phenomenon into consideration and uses the processing unit that is the best suited for the current workload should be the next logical step.

8 Glossary

CPU: Central Processing Unit, used for general purpose processing in computer systems.

GPU: Graphics Processing Unit, traditionally used for calculations involved with graphics rendering.

GPGPU: General Purpose computing on Graphics Processing Units, the concept of using the GPU to perform calculations not related to graphics rendering.

APU: Accelerated Processing Unit, a CPU with added processing functionality, most often seen as a CPU with a GPU included on the same die.

GPU-acceleration: Making use of the GPU to perform processing calculations.

ALU: Arithmetic logic unit, a core part of the CPU that performs operation on integer values.

FPU: Floating-Point Unit, same as the ALU except it operates only on floating-point values.

OpenCL: Open Computing Language, a framework that enables developers to use the GPU for general purpose processing.

SIMD: Single Instruction, Multiple Data, a description of a processing unit that is able to perform a single operation on a larger dataset.

MIMD: Multiple Instruction, Multiple Data, multiple processors able to simultaneously perform operations on separate datasets.

SMT: Simultaneous multithreading, is an advanced hardware scheduling procedure that Intel has implemented under the name Hyper threading

Moore's law: In 1965 Gordon Moore observed that the amount of transistors on a single chip roughly doubled every two years, this has since come to known as the law that state that processing power will double every two years.

MPI: Message Passing Interface, a method for interprocessor communication.

SMA: Shared Memory Architecture, a method for interprocessor communication.

Singlethreaded: An application or part of an application that is executed sequentially on a single thread.

Multithreaded: An application or part of an application that is executed in parallel on multiple threads.

Pixel: The building blocks of digital images, an 128*128 resolution image is composed of 16384 or 128*128 pixels.

Vectorized data: Data that is now seen as an array that usually is not in the array format.

API: Application Programming Interface, the interface through which a programmer accesses functionality of a framework.

Cache: Small and fast memory for storing often accessed data to speedup memory access.

TDR: A Windows watchdog process that resets the graphics driver if an application timesout, Time out Detection and Recovery.

Framework:A support structure which an application is built around.

Zero-copy: A technique that skips unnecessary copying of data.

Heterogeneous computing: The usage of different type of processing units in a system.

SDK: Software Development Kit, is a sett of tools and APIs to help developers in the program development process.

FLOPS: FLoating-point Operation Per Second, an old form of benchmarking when the number of FLOPS were a good reference point in how a chip was performing.

FPGA: A Field Programmable Gate Array is a circuit configurable via a hardware description language.

FSB: Front Side Bus, an interface that allows the CPU to communicate with the northbridge

RGB: Red, Green and Blue, often used to describe the color value of a pixel.

9 References

[1] Geer, David. 2005. “Industry Trends: Chip Makers Turn to multi-core Processors.” Computer 38 (5): 11–13. doi:10.1109/MC.2005.160.

[2] Matloff, Norm. 2012. “Programming on Parallel Machines” https://archive.org/details/ProgrammingOnParallelMachines (accessed May 22, 2015)

[3] O’Regan, Gerard. 2009. A Brief History of Computing. Vasa. doi:10.1007/978-1-4471-2359-0.

[4] Park, In Kyu, Nitin Singhal, Man Hee Lee, Sungdae Cho, and Chris Kim. 2011. “Design and Performance Evaluation of Image Processing Algorithms on GPUs.” IEEE Transactions on Parallel and Distributed Systems 22 (1): 91–104. doi:10.1109/TPDS.2010.115.

[5] Trabelsi, A, and Y Savaria. 2013. “A 2D Gaussian Smoothing Kernel Mapped to Heterogeneous Platforms.” In 2013 IEEE 11th International New Circuits and Systems Conference (NEWCAS), 1–4. IEEE. doi:10.1109/NEWCAS.2013.6573641.

[6] “The Art of Computer Programming: Volume 3: Sorting and Searching (2nd Edition): Donald E. Knuth: 9780201896855: Amazon.com: Books.” 2015. Accessed May 21. http://www.amazon.com/The-Art-Computer-Programming-Searching/dp/0201896850.

[7] Mark Allen Weiss. 2006. Data Structures And Algorithm Analysis In C++. Vasa. doi:10.1002/1521-3773(20010316)40:6¡9823::AID-ANIE9823¿3.3.CO;2-C.

[8] Hwang, Kai. 1993. ”Advanced Computer Architecture”. John Wiley & Sons, Inc., Hoboken, New Jersey

[9] Schenk, Olaf, Matthias Christen, and Helmar Burkhart. 2008. “Algorithmic Performance Studies on Graphics Processing Units.” Journal of Parallel and Distributed Computing 68 (10): 1360–69. doi:10.1016/j.jpdc.2008.05.008.

[10] Gaster, Benedict, Lee Howes, David R Kaeli, Perhaad Mistry, and Dana Schaa. 2012. “Heterogeneous Computing with OpenCL.” Heterogeneous Computing with OpenCL. doi:10.1016/B978-0-12-387766-6.00027-X.

[11] Nvidia Tesla ”Nvidia Tesla Overview” http://www.nvidia.com/object/tesla-servers.html (accessed May 22, 2015)

[12] Khronos group. ”OpenCL Reference” khronos.com. https://www.khronos.org/opencl/ (accessed May 21, 2015).

[13] AMD. 2013. “AMD Accelerated Parallel Processing OpenCL Programming Guide,” developer.amd.com http://developer.amd.com/wordpress/media/2013/07/AMD_Accelerated_Parallel_Processing _OpenCL_Programming_Guide-rev-2.7.pdf (accessed May 22, 2015).

[14] Tompson, Jonathan, and Kristofer Schlachter. 2012. “An Introduction to the OpenCL Programming Model.” http://www.cs.nyu.edu/ lerner/spring12/Preso07-OpenCL.pdf (accessed May 22, 2015)

[15] Lawlor, Orion Sky. 2011. “Embedding OpenCL in C++ for Expressive GPU Programming.” First International Workshop on DomainSpecific Languages and HighLevel Frameworks for High Performance Computing. WOLFHPC 2011, September.

[16] Komatsu, Kazuhiko, Katsuto Sato, Yusuke Arai, Kentaro Koyama, Hiroyuki Takizawa, and Hiroaki Kobayashi. 2010. "Evaluating Performance and Portability of OpenCL Programs." Science And Technology 2: 52. http://vecpar.fe.up.pt/2010/workshops-iWAPT/Komatsu-Sato-Arai-Koyama-Takizawa-Kobayashi.pdf.

[17] Peters, Hagen, Ole Schulz-Hildebrandt, and Norbert Luttenberger. 2011. "Fast in-Place, Comparison-Based Sorting with CUDA: A Study with Bitonic Sort." Concurrency Computation Practice and Experience 23 (7): 681–93. doi:10.1002/cpe.1686.

[18] Payne, Br, So Belkasim, and Gs Owen. 2005. "Accelerated 2D Image Processing on GPUs." Science–ICCS 2005, 256–64. http://link.springer.com/chapter/10.1007/11428848_32.

[19] Intel Corporation "Intel Core i7-4790 specification" ark.intel.com http://ark.intel.com/products/80806/Intel-Core-i7-4790-Processor-8M-Cache-up-to-4_00-GHz (accessed May 22, 2015)

[20] AMD "AMD OpenCL SDK" http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/ (accessed june 5 2015)

[21] Intel corporation. "The Compute Architecture of Intel® Processor Graphics Gen7.5" software.intel.com https://software.intel.com/sites/default/files/managed/f3/13/Compute_Architecture_of_Intel_Processor _Graphics_Gen7dot5_Aug2014.pdf (accessed May 22, 2015)

[22] MinGW "Minimalist GNU for Windows". http://www.mingw.org/ (accessed June 1, 2015)

[23] Intel corporation. "OpenCL Code Builder". https://software.intel.com/en-us/opencl-code-builder (accessed June 1, 2015)

[24] CImg "The CImg Library is an open-source C++ tool" http://cimg.eu/ (accessed June 1, 2015)

[25] Image magic, "Image format conversion" http://www.imagemagick.org/script/index.php (accessed June 1, 2015)

[26] Microsoft corporation. "Timeout Detection and Recovery of GPUs (TDR)" msdn.microsoft.com https://msdn.microsoft.com/en-us/Library/Windows/Hardware/ff570088%28v=vs.85%29.aspx (accessed June 1, 2015)

[27] Satish, Nadathur, Mark Harris, and Michael Garland. 2009. "Designing Efficient Sorting Algorithms for Manycore GPUs." Proceedings of 23rd IEEE International Parallel and Distributed Processing Symposium, 1–10.

[28] "Standard implementation and explanation of sequential image convolution with filter matrices" http://lodev.org/cgtutor/filtering.html (accessed July 04, 2015)

[29] Beberg, Adam L., Daniel L. Ensign, Guha Jayachandran, Siraj Khaliq, and Vijay S. Pande. 2009. "Folding@home: Lessons from Eight Years of Volunteer Distributed Computing." In 2009 IEEE International Symposium on Parallel & Distributed Processing, 1–8. IEEE. doi:10.1109/IPDPS.2009.5160922.

Appendices

A Source code of the basic image convolution kernel

```
__kernel void convolution(__read_only image2d_t input, const int imageWidth
    , const int imageHeight, __global int* inFilter, const int filterWidth,
     const float factor, __global float4* output)
{
   /****
     A sampler is used to set the formatting for how an Image2D
     memory object is read, disabling normalized coords allows
     us to access the pixels of the image by absolute x and y
     coordinates e.g. 852,436. Address clamping means that if
     we select a pixel outside the boundaries of the image the
     closest pixel inside the image will be selected instead.
   ****/
   const sampler_t smp = CLK_NORMALIZED_COORDS_FALSE | CLK_ADDRESS_CLAMP |
       CLK_FILTER_NEAREST;

   //The global id represents pixel coordinates in 2D space
   int2 pos = {get_global_id(0),get_global_id(1)};

   /****
     Because the number of enqueued work-items has to be a power of 2 there
     will almost always be superfluous work-items. We therefore check that
     the current work-item "pos" actually represents a pixel in the image
     if it does not we immediately exit the kernel.
   ****/
   if(pos.x < imageWidth && pos.y < imageHeight)
   {
      //Take 1D filter representation and convert to 2D
      int filter[15][15];
      for(int x = 0, counter = 0; x < filterWidth; ++x)
      {
         for(int y = 0; y < filterWidth; ++y, ++counter)
         {
            filter[x][y] = inFilter[counter];
         }
      }

      float red = 0.0;
      float green = 0.0;
      float blue = 0.0;

      /****
        With the "pos" pixel as the center we traverse the surrounding
        pixels within the filter radius and calculate new RGB values
        for the "pos" pixel.
        ****/
        for(int filterX = 0; filterX < filterWidth; ++filterX)
        {
            for(int filterY = 0; filterY < filterWidth; ++filterY)
            {
                int imageX = (pos.x - filterWidth / 2 + filterX + imageWidth) %
                     imageWidth;
                int imageY = (pos.y - filterWidth / 2 + filterY + imageHeight)
                    % imageHeight;
                float4 currentPixel = read_imagef(input, smp, (int2)(imageX,
                    imageY));

                red += currentPixel.x * filter[filterX][filterY];
                green += currentPixel.y * filter[filterX][filterY];
                blue += currentPixel.z * filter[filterX][filterY];
            }
        }

        //multiply the RGB values by the filter factor in order to maintain
            brightness consistency.
        red = red * factor + 0.00;
        green = green * factor + 0.00;
        blue = blue * factor + 0.00;

        //clamp RGB values to the 0-255 range
        if(red < 0.00)
            red = 0.00;
        if(red > 255.00)
            red = 255.00;

        if(green < 0.00)
            green = 0.00;
        if(green > 255.00)
            green = 255.00;

        if(blue < 0.00)
            blue = 0.00;
        if(blue > 255.00)
            blue = 255.00;

        //Assign a pixel with the new RGB values to the output array.
        float4 modifiedPixel;
        modifiedPixel.x = red;
        modifiedPixel.y = green;
        modifiedPixel.z = blue;

        output[imageWidth * pos.y + pos.x] = modifiedPixel;
    }
}
```

B Source code of the Merge kernel with chunking

```
__kernel void merge_sort_chunk_optimized(__global int* input_list, __global
    int* swap_space, __global int* merge_size, __global int*
   compute_chunk_offset, __global int* max_compute_size)
{
    int my_item = get_global_id(0) * 2;
    int merge_length = *merge_size;
    int sub_list_size = merge_length/2;
    int swap_counter = 0;
    int list_counter = 0;

    int chunk_offset = *compute_chunk_offset;
    int max_chunk_size = *max_compute_size;
    int current_chunk_size = (sub_list_size <= max_chunk_size ?
        sub_list_size : max_chunk_size);  // get the smallest of them.

    int result_offset = 0;
    int list_offset = 0;

    // The swap buffer is already fixed for us so just calculate the merge
    if(my_item % merge_length == 0)
    {
        if(chunk_offset != 0)
        {
            // We are in a chunk fetch the state from last session
            result_offset = swap_space[my_item + chunk_offset - 1];
            list_offset = result_offset - current_chunk_size * (
                chunk_offset/max_chunk_size);
        }

        // Merge the two chunks together.
        while(swap_counter != current_chunk_size && list_counter +
            list_offset != sub_list_size)
        {
            // Compare the elements.
            if(swap_space[my_item + swap_counter + chunk_offset] <=
                input_list[my_item + sub_list_size + list_counter +
                list_offset])
            {
                 input_list[my_item + result_offset] = swap_space[my_item +
                     swap_counter + chunk_offset];
                     ++swap_counter;
            }
            else
            {
                 input_list[my_item + result_offset] = input_list[my_item +
                     sub_list_size + list_counter + list_offset];
                ++list_counter;
            }
            ++result_offset;
        }
        // put the rest of the swap chunk into the list if any
        while(swap_counter != current_chunk_size)
        {
            input_list[my_item + result_offset] = swap_space[my_item +
               swap_counter + chunk_offset];
            ++swap_counter;
            ++result_offset;
        }
        // save the state of the sort in the swap
        swap_space[my_item + swap_counter + chunk_offset - 1] =
           result_offset;
    }
}
```

C Source code of the copy kernel

```
__kernel void copy_to_swap(__global int* input_list, __global int*
   swap_space, __global int* merge_size)
{
    // Concurrent copy to swap.
    int my_item = get_global_id(0) * 2;
    int merge_length = * merge_size;

    if((my_item % merge_length) / (merge_length/2) == 0)
    {
        swap_space[my_item] = input_list[my_item];
        swap_space[my_item+1] = input_list[my_item+1];
    }
}
```

D Source code of the Merge kernel with cooperation

```
__kernel void merge_sort_coop(__global int* input_list, __global int*
   swap_space, __global int* merge_size, __global int*
   compute_chunk_offset, __global int* max_compute_size)
{
    int my_item = get_global_id(0) * 2;
    int merge_length = *merge_size;
    int sub_list_size = merge_length/2;
    int swap_counter = 0;
    int list_counter = 0;

    int chunk_offset = *compute_chunk_offset;
    int max_chunk_size = *max_compute_size;
    int current_chunk_size = (sub_list_size <= max_chunk_size ?
        sub_list_size : max_chunk_size); // get the smallest of them.

    int result_offset = 0;

    // The swap buffer is already fixed for us so just calculate the merge
    if(my_item % merge_length == 0)
    {
        if(chunk_offset != 0)
        {
            // We are now in a chunk fetch the result and list offset from
                last session
            result_offset = chunk_offset;
            swap_counter = swap_space[my_item];
            list_counter = result_offset - swap_counter;
        }

        // Merge the two chunks together.
        int old_offset = result_offset;
        while(result_offset != current_chunk_size + old_offset)
        {
            // Compare the elements.
            if(swap_space[my_item + swap_counter] <= swap_space[my_item +
                sub_list_size + list_counter])
            {
                input_list[my_item + result_offset] = swap_space[my_item +
                    swap_counter];
                ++swap_counter;
            }
            else
            {
                input_list[my_item + result_offset] = swap_space[my_item +
                    sub_list_size + list_counter];
                ++list_counter;
            }
            ++result_offset;
        }
        // Save the current state of the merging
        swap_space[my_item] = swap_counter;
    }
    else if(my_item % sub_list_size == 0)
    {
        if(chunk_offset != 0)
        {
            result_offset = chunk_offset;
            swap_counter = swap_space[my_item + sub_list_size - 1];
            list_counter = result_offset - swap_counter;
        }

        // Merge the two chunks together.
        int old_offset = result_offset;
        while(result_offset != current_chunk_size + old_offset)
        {
            // Compare the elements.
            // swap >= list
            if(swap_space[my_item - 1 - swap_counter] >= swap_space[my_item + sub_list_size - 1 - list_counter])
            {
                input_list[my_item + sub_list_size - 1 - result_offset] = swap_space[my_item - 1 - swap_counter];
                ++swap_counter;
            }
            else
            {
                input_list[my_item + sub_list_size - 1 - result_offset] = swap_space[my_item + sub_list_size - 1 - list_counter];
                ++list_counter;
            }
        ++result_offset;
        }
        // Save the current state of the merging.
        swap_space[my_item + sub_list_size - 1] = swap_counter;
    }
}
```

E Source code of the Copy kernel with cooperation

```
__kernel void copy_to_swap_coop(__global int* input_list, __global int* swap_space)
{
    int my_item = get_global_id(0) * 2;
    swap_space[my_item] = input_list[my_item];
    swap_space[my_item+1] = input_list[my_item+1];
}
```

F Table of run times in milliseconds for different merge sort kernels

merge session	GPU opt	GPU coop	CPU coop	Hybrid	Hybrid opt	CPU merge GPU copy	CPU singel-core
1	27	27	50	27	27	41	125
2	26	27	45	26	26	38	122
3	29	29	43	29	29	39	127
4	40	40	41	40	43	39	148
5	46	46	40	46	38	33	153
6	46	47	38	46	36	32	154
7	47	47	37	47	35	30	154
8	47	47	37	47	32	30	150
9	47	47	37	47	31	29	133
10	47	47	37	47	30	28	112
11	47	48	36	47	29	27	94
12	48	48	35	48	27	26	85
13	48	49	33	48	24	24	80
14	51	51	30	51	22	23	78
15	53	53	29	52	21	21	77
16	55	55	28	55	20	21	76
17	57	55	29	24	21	21	76
18	95	56	29	23	21	22	76
19	177	93	30	23	22	22	76
20	344	171	30	23	22	22	76
21	673	331	30	24	22	23	76
22	1327	636	33	23	19	19	76
23	2600	1238	30	24	25	24	76
24	4976	2384	29	33	33	34	76
total:	10953	5672	836	900	655	668	2476