Linköping University | Department of Computer Science
Master thesis, 30 ECTS| Computer Science
Spring term 2017 | LIU-IDA/LITH-EX-A--17/019--SE

Analysis of GPU accelerated OpenCL applications on the Intel HD 4600 GPU

Arvid Johnsson

Supervisor, Jonas Wallgren (Linköping University)
Supervisor, Åsa Detterfelt (Mindroad)
Examinator, Ola Leifler (Linköping University)

LINKÖPINGS UNIVERSITET

Abstract

GPU acceleration is the concept of accelerating the execution speed of an application by running it on the GPU. Researchers and developers have always wanted to achieve greater speed for their applications and GPU acceleration is a very common way of doing so. This has been done a long time for highly graphical applications using powerful dedicated GPUs. However, researchers have become more and more interested in using GPU acceleration on everyday applications. Moreover now a days more or less every computer has some sort of integrated GPU which often is underutilized. The integrated GPUs are not as powerful as dedicated ones but they have other benefits such as a lower power consumption and faster data transfer. Therefore this thesis' purpose was to examine whether the integrated GPU Intel HD 4600 can be used to accelerate the two applications Image Convolution and sparse matrix vector multiplication (SpMV). This was done by analysing the code from a previous thesis which produced some unexpected results as well as a benchmark from the OpenDwarf's benchmark suite. The Intel HD 4600 was able to speedup both Image Convolution and SpMV by about two times compared to running them on the Intel i7-4790. However, the SpMV implementation was not well suited for the GPU meaning that the speedup was only observed on ideal input configurations.

Content

List of Figures

List of Tables

Acronyms

ILP – Instruction level parallelism

OpenCL – open compute language

SIMD – Single Instruction Multiple Data

MIMD – Multiple Instructions Multiple Data

EU – Execution Units

SpMV – Sparse Matrix Vector Multiplication

FFT – Fast Fourier Transform

APU – Accelerated processing unit

CSR – Compressed Sparse row format

GPU – Graphics Processing Unit

CPU – Central Processing Unit

GPGPU – General Purpose Graphics Processing Unit

PCIe link - Peripheral Component Interconnect Express, the link between the GPU and CPU

AMD – Advanced Micro Devices, a hardware company

OpenDwarf's – OpenCL benchmarking suite

SHOC – The Scalable Heterogeneous Computing benchmarking suite

Benchmark – An application specifically used for timing and analysis

1. Introduction

In order to solve harder problems engineers have always striven to make computers faster. Traditionally the way to do this has been to increase the clock speed of the CPU. However, the clock speed has more or less stopped to increase because of the power wall, instruction level parallelism (running independent instructions in parallel) and memory wall. The "power wall" means that the processor clock frequency cannot be increased because it would produce too much heat, the "ILP wall" means that you cannot achieve more than 3-4 parallel instructions on the same processor because of control and data dependencies and the "memory wall" means that the speed of the memory access simply lags behind enough to set a bound on the processor speed [1]. Because of this the focus has changed from single core speed to the parallelization of programs. Parallelization enables a processing unit with several cores to run several processes in parallel, which in theory equals a faster execution time. Moreover, these days most computers have some form of GPU. GPUs are massively parallel architectures which can be used in general computing for acceleration and freeing up resources for the CPU during everyday computing. When GPUs are used in this way to speed up the computation of general everyday applications it is often called GPGPU, General Purpose Graphics Processing Units.

1.1 Background and motivation

The company Mindroad in Linköping is a software company interested in getting a deeper understanding within the GPGPU-area of computing in general and specifically within OpenCL programming. In line with this, two years ago A Söderholm and J Sörman did the thesis work ”GPU-acceleration of image rendering and sorting algorithms with the OpenCL framework” [2] for Mindroad. In this thesis, the authors constructed a parallel implementation of the image processing algorithm "Image Convolution" and the sort algorithm "merge-sort" in the OpenCL framework. The applications execution time on the Intel i7-4790 CPU and the Intel HD graphics 4600 GPU were then measured. OpenCL enabled them to run the same code on the Intel i7-4790 CPU and its integrated Intel HD graphics 4600 GPU.

Previous studies, such as the study by V. W. Lee et al. [3] have proven that GPU acceleration (the concept of accelerating the execution speed of an application, by running it on the GPU instead of on the CPU) can accelerate the execution speed of applications well suited for parallel work, when a dedicated GPU is used. A Söderholm and J Sörman's [2] purpose was to find out if GPU acceleration can be effectively utilized (i.e. speedup is observed on the GPU compared to on the CPU) when an integrated GPU is used. The expected result was an observable speedup on the GPU, at least in the "Image Convolution" case. However, their result was that running the applications on the GPU was slower in every single case, independent of problem or partition size. In fact, the ratio between the execution time of the GPU and CPU implementations where constant. I.e. the CPU executed the application as much faster than the GPU for an image with a size of 480p as for an image with a size of 1440p, the same was true for different filter sizes.

Why the merge sort ran slower on the GPU can be explained by the fact that it is not very well suited for the GPU. But because Image Convolution is such a suitable problem for parallel work and the fact that GPUs are built specifically for parallel graphics computations, the expectation was that it would run faster on the GPU. This view is also supported by previous studies such as the studies by S. Kim et al. [4] and S. Azmath et al. [5]. In both of these studies the authors achieved GPU acceleration on low powered integrated GPUs with equally parallelizable algorithms. Furthermore B. R. Payne et al. [6] shows that the performance ratio between a GPU and a CPU performing convolution is logarithmically dependent on the image size. The larger the image is the better the ratio is for the GPU. This is because the larger the image is the more sequential work will have to be performed on the CPU, while on the GPU it mostly equals more parallel work. So the fact that A Söderholm and J Sörman's performance ratio was constant was peculiar, therefore Mindroad wanted to know out what caused these results. Because A Söderholm and J Sörman's thesis was on a bachelor level they did not have enough time to analyse their results more closely. Instead they explained their results by stating that the performance gap between the CPU and GPU was too big, basically, the GPU was not fast enough. This still does not explain their observed performance ratio between the GPU and CPU.

This thesis will be a continuation of the thesis introduced above; we will analyse their Image Convolution implementation and compare it to another convolution implementation, in order to find out what caused their results. Their merge-sort implementation will however not be tested because it is not especially well suited for parallel work. Moreover we will choose a benchmark application from a commonly used benchmark suite, to get a more general comparison between the GPU and CPU. The benchmark application is also used in order to rule out the possibility that the result depends on the programming skills of the author, rather than on the hardware.

Their results and code are interesting to examine in order to find out whether this specific GPU can be utilized for GPU acceleration or not. If it can, developers have an alternative to the more expensive dedicated GPUs when they need to speed up their applications. Furthermore, it can provide practical lessons for what a developer needs to consider when writing OpenCL programs for integrated GPUs.

What kind of benchmark application is used is an important choice. In order to have any chance at all to achieve speed up on the GPU it has to be well suited for parallel work. An algorithm well suited for parallelization on a GPU consists, in general, of a set of operations which will be applied to all data points in the problem. These are generally known as data parallel algorithms, because the parallelization comes from performing the same operation on multiple data in parallel. They are suitable for GPU execution because each data point can very easily be mapped to each thread.

Since the focus of this thesis is on GPGPU computing, the benchmark application should also be common enough to not be used only in scientific computing. The specific application is chosen based on its different characteristics. Of course since a pre-existing benchmark application should be used, the choice is also limited by which applications the existing OpenCL benchmark suits consist of.

In order to find out what caused A Söderholm and J Sörman's [2] results, the tests will be performed on the same hardware used by them. Therefore, we will run the applications on the Intel i7-4790 CPU and on its integrated Intel HD graphics 4600 GPU and see if we get the same result as the previous thesis. If we get the same result, we will analyse the outcome in order to get a better understanding of what causes it, besides that the integrated GPU's execution units are slower than the CPU's. If the GPU implementation has a faster execution time we will analyse why that is the case and what kind of parameters it depends on, partition, number of cores, problem size and so on. This means that more time will be put on the analysis compared to the previous thesis.

1.2 Purpose

The purpose of this thesis is to test the results of the thesis "GPU-acceleration of image rendering and sorting algorithms with the OpenCL framework" to see if our results will be the same, and if so, to find out what causes this result. Thereby providing more knowledge of which types of algorithms that can be successfully GPU accelerated, and what one needs to consider when writing an application in OpenCL for an integrated GPU.

Furthermore, we want to examine if running SpMV and Image Convolution equals a gain in performance on the Intel HD graphics 4600 GPU using the OpenCL framework. A performance gain is specified in this case as: when running the application on the GPU an execution time speed up can be observed compared to if it would have been run on the CPU. In previous works by e.g. S. Y. Kim et al. [7] and E. Ching et al. [8] they prove that the Intel HD graphics 4600 can be used to speed up datacentre MapReduce tasks but it does not speed up database queries. However, there have not been very much research on whether it can speed up general purpose applications such as convolution and matrix vector multiplications which this thesis focus on.

1.3 Delimitations

This project will handle the testing of Image Convolution and Sparse Matrix Vector multiplication implementations. This means that no implementation will be made from scratch. Moreover, analysis of the execution times will be performed in order to compare it to the result of the previous thesis and figure out what causes them. The specific focus is on the execution time, not on for example the power consumption.

1.4 Research questions

The research questions of this report are the following.

- Which are the most important factors determining the execution time of Image Convolution and SpMV on the Intel HD graphics 4600 GPU compared to the factors impacting the execution time on the Intel i7-4790 CPU?
- Do you observe any speedup when running Image Convolution or SpMV on the Intel HD graphics 4600 GPU compared to when running them on the Intel i7-4790 CPU, furthermore what causes this result?

2. Theory

In this chapter all theory regarding the platforms, OpenCL, parallel algorithms and analysis is presented.

2.1 Parallel Computing

Parallel computing is the computing model of running several instructions in parallel. This is achieved by running several threads on multiple cores simultaneously, instead of running everything sequentially on one core as in classical programming. In contrast to sequential programming where the main programming model is the von Neumann model, in parallel programming there are two competing architectures [9]. These are the distributed and shared memory architectures as well as a multitude of different models for both distributed and shared which are better suited for different problems.

2.1.1 Distributed memory and Shared Memory

Distributed memory systems are systems consisting of multiple processors with their own memory. These processors communicate with each other using a message passing interface over an interconnection network. In contrast, a shared memory architecture consists of multiple processors sharing the same memory, and a global clock controlling both the memory and the processors. Furthermore, there are no limits to how many processors which can access the memory simultaneously. The distributed memory architecture is often used in big clusters where you have several nodes of processors working together over an interconnection network, while in ordinary desktop computers the shared memory architecture is used. This thesis is performed on a shared memory architecture.

2.1.2 The data parallel computing model

While there are several parallel computing models, the one explained here is the data parallel model. This is because it is the one of the two supported by OpenCL which is highly suitable for GPU computing [10]. Furthermore, all the algorithms tested in this thesis are data parallel algorithms.

Data parallelism works very well with SIMD execution (issuing the same operation to multiple cores) because it revolves around achieving parallelism by executing the same operation elementwise on big sets of independent data. Because the data is independent of each other the operations can be performed in parallel.

2.2 CPU

CPUs are designed to be general purpose machines; a CPU should be able to handle both parallel and sequential programs and in general whatever problem issued to it. Therefore, CPUs rely on MIMD execution, MIMD means that different operations are issued to the separate cores. This is done so that the different cores in multicore CPUs can perform different operations on different data. Consequently one core can execute an entire program on its own or the cores can split the program between themselves and share the workload. Moreover, the cores have access to advanced operations such as branch prediction and are in general faster and more flexible compared to GPU cores [11].

2.2.1 Intel i7-4790 CPU

The Intel i7-4790 is one of Intel's fourth generation (Haswell) i7 processors [12] [13]. It has four cores, eight threads (two per core) and a base clock frequency of 3.6 GHz. If needed it can also speedup one of the cores to 4 GHz using the Intel turbo boost feature. Moreover, it has 3 levels of cache where the first (64kb) and second (256kb) is separate to each core, while the third (8 megabytes) is shared between the cores [14] [15]. The cache also has a bandwidth of 64 bytes per cycle meaning it can load one entire cache line per cycle. Moreover, the memory system supports both transactional memory and gather instructions. The support for transactional memory is made possible by the transactional synchronization extension. In practice, what it does is to make operations protected by an atomic lock. This means that as long as they do not perform conflicting operations on the data they can be executed simultaneously by multiple threads.

2.3 GPU

In contrast to CPUs modern GPUs are designed with the assumption that they will be assigned highly parallel tasks and they focus solely on high throughput. This means that they are not as flexible as CPUs but can be utilized for speedup of many parallel problems [16]. In order to achieve this, they mainly rely on three architectural ideas: SIMD execution, a very high number of small and simple execution cores and abundant use of hardware multithreading [17].

- With SIMD execution you issue the same operation to multiple cores that handle different data. In this way you can easily perform the same operation on big sets of data [17].
- In GPUs many simple cores are used instead of the CPUs few but complex cores. The main thing these small cores are sacrificing is the possibility of out of order execution and branch prediction, but the spatial savings for the chip results in more cores which outweigh these sacrifices. It also works well with SIMD [17].
- Hardware multithreading enables highly parallel computations to be decomposed in to multiple serial tasks which equal an even higher degree of parallelism [17].

2.4 Integrated GPU

As stated above the Intel i7-4790 is a CPU with an integrated GPU, meaning that the CPU and the GPU exist on the same die. The potential big advantage gained by using an integrated GPU instead of a discrete GPU is that the PCIe link between the CPU and a discrete GPU can be a performance bottleneck. This is because in all GPU acceleration you will need to transfer the data at least two times, one time from CPU-GPU and one time back. As shown by C. Gregg and K Hazelwood [18] this data transfer can have a significant impact on the execution speed specifically on the bandwidth bounded types of algorithms. However, an integrated GPU shares the same cache and memory as the CPU. This means that you completely circumvent the PCIe link, which for bandwidth bounded algorithms can equal a higher execution speed on an integrated GPU than on a discrete one [19].

2.4.1 Intel HD graphics 4600 GPU

The Intel HD graphics 4600 is the integrated, on die graphics processing unit of the Intel i7-4790 CPU [20]. It is constructed to be a power efficient, cheap but not as powerful alternative to a dedicated GPU. It has a base clock frequency of 350 MHz but can be boosted up to 1.2 GHz using the dynamic frequency scaling feature which can on the fly adjust the frequency of a processor in order to conserve power and/or heat. The 4600 is a part of Intel's 7.5 generation of processor graphics. The GPUs in Generation 7.5 are structured in slices, each slice has two sub slices each containing 10 execution units. Moreover, all slices have one common level 3 cache where each slice has access to a maximum of 256 kilo bytes. Within the level 3 cache there is also a shared memory location with a size of 64 kilo bytes for each sub slice which supports sharing local memory between kernels in OpenCL's work groups. Each cache line is 64 bytes and each sub slice has a data port with a read and write bandwidth of 64 bytes/cycle, supports gather scatter operations and coalescing memory accesses. Coalescing memory access means that given a SIMD read operation for 16 32-bit floating point values with different address offsets in the same cache line, the processor can coalesce the reads to one 64-byte read of the entire cache line. Each sub slice also has a sampler unit which handles a level 1 and level 2 cache especially used for images and textures. The sampler just like the data port has a read bandwidth of 64 bytes/cycle [14].

The 4600 has two slices, resulting in a total of 20 execution units. Each EU has 7 hardware threads resulting in 140 hardware threads. And each hardware thread can run a maximum of 32 software threads. This means that the entire chip can run 32 * 140 = 4480 concurrent threads. Moreover, each hardware thread has access to 128 32-byte general purpose registers, resulting in each EU having access to 28 kilo bytes of local storage.

2.5 OpenCL

As stated in section "1.1 Background and motivation", this thesis will be conducted on the Intel i7-4790 CPU and its integrated GPU Intel HD graphics 4600. In order to not have to use two different programming platforms which in itself can affect performance [10] and to replicate the results of A Söderhom and J Sörman [2], OpenCL is used. OpenCL is a programing standard, constructed with portability in mind meaning the same code can be run on different platforms, a CPU and a GPU in this case. Normally this is not possible because GPU and a CPU architectures are inherently different. To begin with CPUs use the MIMD model while GPUs uses SIMD. This means that the way in which parallel code is written differs a lot. Moreover, not only are there differences between CPUs and GPUs but there can also be major differences between different GPU models. This means the code will differ from model to model. This problem is solved by OpenCL.

2.5.1 Platform model

OpenCL is an open framework which produces highly portable code specifically for parallel programming, meaning you can use the same code across different platforms. The model, used by OpenCL, consists of a host and several devices. The host maps to the main controlling core of the program, e.g. if the program runs on a CPU it maps to the core that starts and merges the threads and if it runs on a GPU it will map to the systems CPU. Consequently, the devices can map to either the rest of the cores on the CPU or the GPU's cores. Note that the exact mapping is dependent on the relevant platform.

2.5.2 Memory model

OpenCL's memory model consists of two main memory types, Host memory and Device memory. Not surprisingly the host memory is the memory the host has access to and the device memory is the memory the devices have access to.

In turn the device memory consists of four memory regions where the kernels can allocate data depending on what kind of data it is and which kernels need access to it. There are two global memory regions which can be accessed by everyone in all work groups: Global memory and Constant memory. Global memory, as you might think is the standard global memory which all devices can read and write to/from. In the constant memory all data remain constant during execution time. Only the host can allocate data there and the devices can read from it. There is also a private memory region which is each kernels individual region. Lastly there is a local memory functioning like a shared memory between devices in the same workgroup meaning that all devices in the same workgroup can access it.

2.5.3 Execution model

The execution model consists of two entities which are mapped to the host and the devices in the platform. These are kernels and a host program, on the devices the kernels are run. The kernels are either mapped one to one or one to many, meaning that one kernel can run on either just one device or on several devices. However, on the host only one host program can execute at the same time. The host program is the main program that manages everything such as memory management and synchronization while the kernels handle the actual work in the computation. This work is executed in so called work-items more commonly known as software threads and they work together in so called work-groups.

The kernels are defined in a context which consists of the devices it executes on, the OpenCL functions, the actual kernel implementation as well as the memory it needs for the variables it is operating on.

Each kernel context is managed by the host program through a command queue which can be loaded with functions from the OpenCL API. Through the command-queue the host starts up new kernels, transfers data between for example host and device memory and manages synchronization between different kernels. Moreover each kernel can enqueue commands to a command queue specific for the device it executes on, through this command-queue, new child kernels can be started.

Regardless of the command queue each command transition between the six states Queued, Submitted, Ready, Running, Ended and Complete.

1. **Queued**: The command is placed in a command-queue.
2. **Submitted**: The command is submitted for execution.
3. **Ready**: The prerequisites are met and the command is moved in to a work-pool, here it will be scheduled.
4. **Running**: Execution starts and continues to run.

5. **Ended**: Execution is done.
6. **Complete**: The command and its child commands are done.

The commands in the same command queue execute in relation to each other either in order or out of order. However, different command queues in the same context execute independently from each other. In order execution means that the commands are executed in the same order as they were enqueued. Out of order execution means that the commands are executed in any order only constrained by synchronization and dependencies. The general communication and synchronization of commands are handled by event objects.

When a kernel is executed an index-space is defined, the parameters of the index space coupled with the argument values to the kernel and the kernel itself defines a kernel instance. When the kernel instance executes, it executes for each index in the index-space resulting in several instances of the kernel running concurrently. These separate kernels are the work-items, they all get an ID corresponding to their position in the index space and they are handled by the device in work-groups.

2.5.4 OpenCL for CPUs

While OpenCL have been developed for portability between different architectures the foremost focus and what it mostly has been used for is GPU architectures. This results in that while the code can run on CPUs as well as on GPUs the performance is not as portable [21]. In fact some of the specific optimizations for a GPU architecture in OpenCL can have the opposite effect on a CPU. Firstly, the work-items in OpenCL are quite small which suits the small execution units on a GPU, however mapping these simple tasks to a complex CPU core is somewhat of a mismatch. This mismatch leads to the CPU's cache utilization being restricted, because the work-item running on one hardware thread processes such a small part of the entire problem. Secondly, the SIMD units within the CPU cores needs vectorised input which you normally do not use in GPU code, meaning that some of the cores computational power will not be utilized. Furthermore, if the default data transfer model used on GPUs is used it will result in unnecessary data transfer between host and device. Perhaps the most common GPU optimization, coalescing memory access, also has a bad effect on CPUs because it is orthogonal to the CPU's cache accesses (GPUs reads from memory column wise while CPUs reads row wise) meaning it actually results in slower memory accesses on the CPU. Lastly, local memory which is prevalent on GPUs is non-existent on CPUs which means that these addresses gets mapped to the global memory, once again meaning that the intended optimization only results in extra overhead equal to the amount of memory being needlessly copied to the same memory. OpenCL code can of course also be optimized for CPUs which can have a negative impact on the performance if you run it on a GPU.

2.6 Parallel Algorithms and Data parallel algorithms

Parallel algorithms are algorithms designed to in one clock cycle perform multiple instructions. However, in order to gain performance from parallelizing an algorithm the algorithm in question needs to contain a lot of parallelization possibilities [22]. In other words the algorithm should handle a very big problem and have a suitable partition in order to be able to split the problem in parts, so that as many as possible can be run in parallel. Algorithms which are especially suitable for this are e.g. image processing and matrix operations. This is because of their inherent nature of having very big problem sizes and independent data. Moreover, if the algorithms are data parallel they perform the same operation on every input element (every pixel in the image case and every index element in the matrix case) which then can be mapped to each thread/work-item, in theory [23].

Here follows five different data parallel algorithms which all are typical benchmark applications which have been used to compare CPUs to GPUs by e.g. V. W. Lee et al. [3] and C. Greg and K. Hazlewood [18]. There are several more data parallel algorithms which could have been considered, however because of time limitations only five were evaluated. The reason for evaluating these five was the following. Firstly all of these have been tested before and this enabled a comparison of the results. Secondly they are all available in existing benchmark suites.

2.6.1 Image Convolution

Image Convolution is a data parallel image processing algorithm that coupled with a filter can alter the look of pixels in an image e.g. blur them, sharpen or mark the edges. The convolution itself is the action of producing a new value for a goal pixel. This is done by performing a mathematical operation on the value of the goal pixel together with the surrounding pixels and giving the result as a new value to the goal pixel. The specific filter used specifies what kind of operation that should be done on the goal pixel and its neighbours. For example, if an averaging filter is used the value of the goal pixel and its surrounding pixels are summed together, and then they are divided by the number of pixels used and this new value is given to the goal pixel. How many of the surrounding pixels needed to be used is determined by the size of the filter e.g. 3x3 (9 pixels) or 10 x 10 (100 pixels) [6]. This algorithm suits data parallelism very well because each operation is performed on one pixel, meaning that each thread can in theory handle one pixel each.

2.6.2 DFT and FFT

The Discrete Fourier transform is a way to transform an image from the spatial domain to the frequency domain. An image in the spatial domain is how we classically think an image is represented, by colour values at spatial locations in the image. An image in the frequency domain is represented by a set of sinus waves and their amplitudes and phase shifts. The reason for doing this transform is that some computations run much faster or are much easier to implement in the frequency domain. It is performed by applying the following function to the image represented as a vector with N elements.

$$F(u) = \frac{1}{N} \sum_{x=0}^{N-1} X_N e^{-\frac{j2\pi}{N}}$$

Where N is the number of samples or in this case pixels, x is the pixel value. The fast Fourier transform is simply a faster version of the discrete Fourier transform based on the divide and conquer method. Instead of applying this function and using all the pixel as the input you split it up and then put the results back together. If the size is optimized so that you always can split it in two parts you will get a final input size of one pixel, meaning that you perform the equation one pixel at a time and then combine the result to represent the entire image. This is very suitable for data parallelism, because in theory you could simply let each thread handle each pixel [24].

2.6.3 Sparse matrix vector multiplication

Sparse matrix vector multiplication is the operation of multiplying a sparse matrix (a matrix where most of the elements are zero) with a dense vector (a vector where most elements are non-zero). In normal matrix vector multiplication where both the matrix and vector are dense, you simply multiply each value in the matrix with the values on the corresponding indexes in the vector, adding those products together and place the resulting sum in a result vector. This means that you will access the same memory over and over for the different multiplications which means that the most prominent operation is memory accesses. This means that the algorithm is being bounded by bandwidth. SpMV is still bounded by memory but not to as large a degree because there are simply not as many elements to access because many of them are zero (irrelevant). However, you do need to store the row and column indexes instead to be able to perform the correct multiplications. This problem does also fit neatly with data parallelism. The algorithm splits up the matrix elements to all the threads, couple the matrix element with the elements from the vector they should multiply it with, and lastly the threads place the resulting sums in the same result vector [25] [26].

2.6.4 Radix sort

Radix sort is a fast stable counting sort algorithm instead of the more common comparison sorts. The biggest difference is that it sorts each number independently from the others without any comparisons. Instead, the correct position is calculated by counting how many smaller numbers there are. It does this by counting the number of occurrences of each number and placing the result in a new array, where the index corresponds to which value it represents. It then calculates for each index/value how many occurrences of smaller numbers there are. This is done by adding the values at all the indexes to the left of itself. Thereby knowing where to place the current number. This independent calculation makes it possible to implement it as a data parallel algorithm where each thread handles one value in the array to be sorted [27].

2.6.5 Histogram computation

Histogram computation is a way to calculate the number of results which fall into disjoint categories. It is a common operation in e.g. image recognition, pattern recognition or any sort of application which relies on statistical analysis. What a histogram computation does is to count every occurrence of a set of given disjointed categories. Each category is represented by a so called bin which holds a number (starting at 0), and for each occurrence of the respective category the algorithm adds 1 to the number in the categories bin. This can also be data parallelized on a GPU by simply giving each thread one point of the input and making it perform the addition in the appropriate bin. However, this can result in many stalls since the threads will try to access the same memory when adding in the bins. In order to resolve this a common solution is Replication. Replication means that the work-groups save their own private histogram which they add to. When every private histogram is done the private histograms are added together. This lessens the amount of stalls since there will be a smaller amount of threads accessing the same amount of memory locations [28].

2.6.6 Performance considerations of parallel algorithms

Because of the inherent nature of parallel algorithms, they are not analysed with the same metrics in mind as sequential algorithms [29]. First and foremost the reason for parallelizing algorithms is to gain a speedup in comparison to the sequential execution time, which is why this is normally the most important performance metric. The speedup is calculated with the following formula Ts/Tp = Speedup where Ts and Tp is the execution time for the sequential and the parallel implementation. There are some parameters which can have a bounding effect on how fast a parallel algorithm can run.

Firstly, how big part of the algorithm can run in parallel. Every algorithm have some part of the code that has to be run sequentially. The size of the sequential section will bound the theoretical gain of running on multiple processors.

Secondly, the load balance between the cores can slow down the execution time, if for example one core has a lot less to do than the others it will be an unused resource. This happens if one core is done a long time before the other cores and there is not any smart scheduler implemented which can give it more work. Then it will just be idle and wait for the other cores to finish. This can specifically have a big impact on the CPU implementation because of its MIMD architecture and the fact that it does not have a native queue like the GPU which simply schedules the first task in the que to a core when it goes idle. Instead, on the CPU the programmer has to pay close attention to the scheduling and partitioning of the problem [11].

Thirdly the communication and synchronization between threads can also slow down execution time by making threads wait for one another a high amount of time. This becomes apparent if for example, one of the cores has to send a lot of data to the other cores, or when a shared resource is used by all the cores. This results in a lock being needed and the individual cores needing a lot of time to process the resource. This is similar to the load balancing problem with the difference that it cannot be solved by a smart scheduler. Instead then the entire program will have to wait for that one core to finish before the others can continue if you cannot utilize some sort of lock free synchronization.

As mentioned earlier C .Gregg and K, Hazelwood [18] prove that data transfer between the GPU and CPU can have big performance ramifications on GPU programs meaning that the less data transfer the better for GPUs. This is especially true if an architecture with a discrete GPU is used and a PCIe express bus is used for the communication between the GPU and CPU. The communication cost is considerably lowered on a heterogeneous architecture with an integrated GPU but it still matters.

The data access speed to cache and other memory can also have a big impact on the performance of an architecture. Especially in parallel algorithms because their data sets are in general very large meaning that a large number of memory accesses will have to be performed [8].

Lastly when parallelizing a program some overhead always occurs, largely due to the creation of threads, intra processor communication and the joining of threads. This will affect the performance and how much overhead there is will differ from architecture to architecture, in general it is much more prevalent on accelerator cards like GPUs than on CPUs [19].

2.7 Benchmarking parallel programs

In order to eliminate the risk that the results will depend on our implementations of the algorithms, established benchmarks suites will be used to perform the tests. To test SpMV the OpenDwarf's benchmark suite will be used. OpenDwarf's is a benchmark suite constructed with the specific goal of supplying a common means for evaluating the multitude of different parallel architectures commonly used today. In order to achieve this it was built in OpenCL which can be run on most of today's parallel architectures and the benchmarks themselves were modelled from Berkeley's Dwarfs. Berkeley's Dwarfs is a set of applications (Dwarfs) which each captures a specific pattern of communication and/or computation commonly occurring in a class of common applications [30]. Another central idea utilized in OpenDwarf's in order to achieve its goal is that none of the Dwarf implementations are optimized for a certain platform. This means that all the kernels are standard implementations none using techniques such as shared memory which is specifically suitable for GPUs. However, variables such as work-group sizes are set automatically by the OpenCL runtime, depending on the current architecture. This way they do not favour a specific architecture they instead favour all [31].

In order to test convolution the code from the previous thesis is used as well as a standard implementation provided by AMD (the hardware company) which includes multiple kernels. Most of them are optimized for the CPU but one of them is a standard implementation without any optimizations.

2.8 Software engineering Research methodology

In the paper: Preliminary guidelines for empirical research in software engineering, B. Kitchenham et al. [32] presents a methodology for conducting experiments and empirical research within software engineering. Because our thesis can be defined as an experiment the relevant parts of their methodology has been followed. B. Kitchenham et al. [32] splits research in to six topics which all can be linked to the different parts of a study and constructs guidelines for each.

The parts included in this thesis are the following:

- Defining an experimental context
- Defining an experimental design
- Experiment and data collection guidelines
- Result interpretation guidelines

The experimental context consists of the background information of the industrial circumstances which the experiments are derived from, the research hypotheses and which theory pertaining to them and lastly the related research. Moreover, hypotheses and research questions should be stated clearly before any data is collected or analysed in order to know what to look for in the data and avoid bias in the hypotheses.

The experimental design consists of stating what population the subjects or objects are drawn from, in this case from what population the algorithms and data sets are chosen from. If this is not done it is not possible to draw correct conclusions from the experiment.

The guidelines for conducting the experiment and data collection states that all measures should be explained because there are not any overarching standard for all software tests. More specifically all analysis procedures should be clearly described.

Finally, the interpretation of the results. The most important part of the interpretation is that the conclusions should follow directly and in a clear manner from the results. Moreover what population the different conclusions pertain to should be defined as well as what kind of study it is.

B. Kitchenham et al. [32] describes a very general theoretical methodology for software experiments which we follow in order to present and handle the data in a suitable way. However, what to take in to consideration in this thesis practical analysis, is not handled. Therefore, in regards to the practical evaluation of the different implementations we follow the methodology used by V. W. Lee et al. [3].

From V. W. Lee et al. [3] we used the general methodology of how they performed the comparison between the CPU and GPU implementations. More specifically, how much the applications are optimized for the two platforms will be taken in to consideration during the analysis. Data transfer will also be taken into consideration because we want to evaluate the results as they would be in the real world. Moreover as stated by C Gregg and K Hazelwood [18] the data transfer can have a big impact. However this thesis is conducted on a CPU with an integrated GPU, while C Gregg and K Hazelwood used a dedicated GPU, it can be interesting to see how much impact data transfer has on our results compared to theirs.

2.9 Related works

Since the rise of multicore computers, GPU acceleration has been explored by many scientists. In “*Debunking the 100X GPU vs. CPU Myth: An evaluation of throughput Computing on CPU and GPU*” V. W. Lee et al. [3] examines how much performance that is actually gained by GPU acceleration on a NVidia GTX280 GPU, compared to running it normally on an Intel Core i7 - 960 CPU. The comparison was performed by running 14 different throughput computing kernels on the two processors. The kernels are modelled in order to capture the core computation and memory characteristics specified by the PARSEC and PARBOIL benchmark suit applications. Moreover their performance analysis outlines what the different kernels execution time was bounded by and lastly discusses how to optimize the code depending on which platform you use. Their conclusion showed that while GPU acceleration does equal a performance increase it is not in the orders of magnitude which is normally stated. According to their measures between the GTX 280 and i7-960 the performance increase lands around 2.5 times faster execution time on the GPU. They believed that a major factor to previous recorded performance increases depends on which CPUs and GPUs were used and what kind of optimizations that were used on the code.

In “*Power Efficient MapReduce Workload Acceleration using Integrated-GPU*” Kim et al. [7] uses the Intel HD graphics 4600 GPU to accelerate MapReduce tasks in the Apache Hadoop cluster framework for datacentres and compares it to an equivalent CPU implementation. The tests were performed on a 4-node cluster and a 1-node cluster and they used the HiBench benchmark suite for testing. The performance metrics they used were performance in time, IO overhead isolation and power consumption. To evaluate the performance in time they simply measured the execution times and compared them. In order to minimize variations they ran the tests multiple times and used an average for the actual comparison. Since MapReduce is very IO dependent, the tests they performed isolated the impact of the IO on the performance. More specifically they measured how fast they could send data to the map task isolated from all other components. Finally, they measured the amount of power consumption from all 4 nodes combined on the 4-node cluster and the power consumption of the CPU and integrated GPU on the 1-node cluster. Their result was that on the integrated GPU the MapReduce task hade ben converted from a Compute bound kernel to an IO bound kernel and that the GPU offered a significant speedup over the CPU.

In a similar study titled "*Evaluating Integrated Graphics Processors for Data Center Workloads*" S. Kim, I. Roy and V. Talwar [4] examine GPU acceleration in the integrated GPU on the AMD fusion architecture. Their goal was to evaluate it for Data centres however as a first step they evaluate it for the fusion architecture in isolation. The main difference from "*Power Efficient MapReduce Workload Acceleration using Integrated-GPU"* except that they use different hardware is that in this article they have more general tests targeted at GPGPU computing. More specifically they use the SpMV and Sort benchmarks from the SHOC (Scalable Heterogeneous Computing) suite which is a suite constructed specifically for GPGPU computing on heterogeneous systems (systems with multiple EUs). The comparison focuses on the energy efficiency and the raw performance (execution time) and is done on the architectures CPU compared to its GPU. They found that both SpMV and Sort ran faster and were more power efficient on the integrated GPU.

In *"Accelerating adaptive background modeling on low-power integrated GPUs"* Azmat et al. [5] examine whether you can accelerate the multi-modal mean (MMM) algorithm on Nvidias low powered ION GPU compared to running it on a single core Atom-330 CPU. MMM is an image processing algorithm which segments the background from the foreground and maintains running means of the values of the background pixels. The pixels have up to 4 modes depending on what kind of background it represent and each mode contains separate means of all colour components (i.e. R G and B) of a pixel. This algorithm enables modelling of dynamic backgrounds, e.g. swaying trees. They continue to detail all of the optimizations done on the CUDA platform however they do not mention what implementation the Atom CPU runs. Their results state that they experienced a 6x speedup on the GPU compared to the CPU however because we do not know the specifics of the CPU implementation we are not sure how dependable these results are.

In *"Unleashing the Hidden Power of Integrated-GPUs for Database Co-Processing"* Ching et al. [8] examine the potential of using integrated GPUs for the data bandwidth dependent work in Database processing such as queries instead of discrete GPUs. In their experiments, they used the Nvidia GTX 780 discrete GPU and the Intel HD 4600 integrated GPU. The main architectural differences are the memory interconnection speeds, the cache structure and the computational capacity. Both the interconnection speed and the cache structure favour the integrated GPU except the fact that the discrete GPU has access to a bigger cache. Regarding the computational capacity, it is a bit more complicated, the discrete GPU outperforms the integrated by nine times purely in floating point operations. However, the integrated one has many more data lanes to run instructions on, resulting in a bigger amount of parallelism which favours many small jobs. Because of the faster memory connections and the extra parallelism, the integrated GPU outperforms the discrete in these kind of bandwidth dependent workloads. Moreover, they also compared the integrated GPU to running the database queries on the i7-4770k CPU. In pure execution speed the CPU outperformed the GPU by a small amount (1-2 ms), but when normalizing it to the power consumption, the GPU outperformed the CPU by a vast amount.

In *"On the Efficacy of a Fused CPU+GPU Processor (or APU) for parallel computing"* Daga et al. [19] study the AMD Fusion accelerated processing unit which is a CPU with an integrated GPU. More specifically they compare this APU architecture to the more traditional with a discrete CPU coupled with a discrete GPU and why APUs has the potential to overtake discrete GPUs performance wise. Like in many of the other works referenced here they too state that the major benefit of using an APU architecture is that you can get around the PCIe performance bottleneck. However, they also state that you do not always get better performance. In order to gain performance from not having to send data over the PCIe the amount of data has to be quite large. For small data sets, there are no significant performance gains. Moreover, they tested the architecture with four benchmark applications (MD, FFT, Scan and Reduction) from the SHOC benchmark suite in order to model real world workloads. The tests were performed on the AMD Zacate APU, the AMD Radeon HD 5870 GPU and the AMD Radeon HD 5450. The 5870 is a high performance GPU while the 5450 is more or less a discrete version of the APU. MD executes fastest on the 5870 but it executes faster on the APU than on the 5450. FFT however is slowest on the APU because it depends too much on the computation and memory speed which is slower than both the others. However, both scan and reduction ran fastest on the APU given a large enough problem size.

In summation, from these previous works it seems to be possible to achieve speedup on less powerful, integrated GPUs. However even though the application is well suited for parallelism it is not guaranteed to run faster on the GPU. Whether the GPU or the CPU runs the application faster seems to depend a lot on the specific application and what it is bounded by. Heavily compute bounded applications seems to favour CPUs, since they require higher single thread speed which CPUs provides. On the other hand bandwidth bounded applications usually seem to favour GPUs. However note, this in not always true, sometimes the shear amount of data can enable the GPUs parallelism to execute it faster. Then again, sometimes integrated GPUs simply are not fast enough, or has to small cache to compete with CPUs.

3. Method

In this chapter the algorithm selection, the selected algorithm, implementation and how the analysis was done are presented.

3.1 Benchmark Application Selection Factors

That convolution would be examined was decided from the start since it is the one which A. Söderholm and J Sörman [2] examined. Besides, it is very well suited for data parallelism with the only apparent limiting factor on the GPU execution speed being the size of the filter. The bigger the filter the longer the processing of the pixel will take and the more memory each thread will need [6]. This means that for bigger filters it is computationally bounded [3].

Since there was not enough time to test all of the other algorithms, we choose the one best suited for the Intel HD graphics 4600 and which performance depended on other characteristics than convolution depends on (e.g. bandwidth or cache bounded).

The one algorithm best suited for the GPU was the one which we thought would have the biggest chance of running faster on the GPU. This was based on the following factors from the previous studies and the architectural limitations.

- Computationally bounded algorithms usually favours the CPU since they have a much higher clock speed.
- Bandwidth bounded usually favours the GPU since most of the time the GPU at least has as fast data access as the CPU (if not faster). When the bounding factor is the same the GPU's parallelism should outperform the CPU's sequential speed.
- Cache bounded algorithms favour the architecture with the biggest cache.

Since the Intel i7-4790 has a much higher clock speed (3.6 GHz compared to 1.2 GHz) and a much bigger cache (8 Mb compared to 256 kb) but the same bandwidth (64 bytes/cycle) we wanted a bandwidth bounded algorithm.

However the available implementations in OpenDwarf's will also be taken in to consideration. Even though we want the algorithm to be well suited for the GPU the implementation should not be specifically optimized for the GPU or the CPU. Since then the results would be more dependent on the specific implementation rather than on the hardware.

3.2 Benchmark Application Selection

While FFT is a widely used algorithm it is computationally bounded since the majority of its operations are computations. Furthermore, the OpenDwarf's implementation is optimized for the GPU, meaning that it would not be a fair comparison between the hardware architectures. Because of these two factors it was not chosen [3].

Both Histogram and Radix are cache bounded. Radix is cache bounded because each thread will need to save a big part of the original array in order to calculate the correct position of their values. Histogram is cache bounded because of the private histograms which are saved between the work-items in a work-group, needed to lessen the amount of stalls. Therefore neither Histogram nor Radix were chosen.

However SpMV is much like FFT one of the most widely used algorithms within its computing area. It is also used in many related studies such as in the studies by V. W. Lee et al. [3] and C Gregg and K Hazelwood [18] for conducting comparisons between CPUs and GPUs. Finally, it is bandwidth bounded and it does not have any other particular performance problems on GPUs [18].

3.3 Benchmark Application Implementations

Figure 1 shows the different implementations tested in this thesis, the Green boxes are the actual implementations.

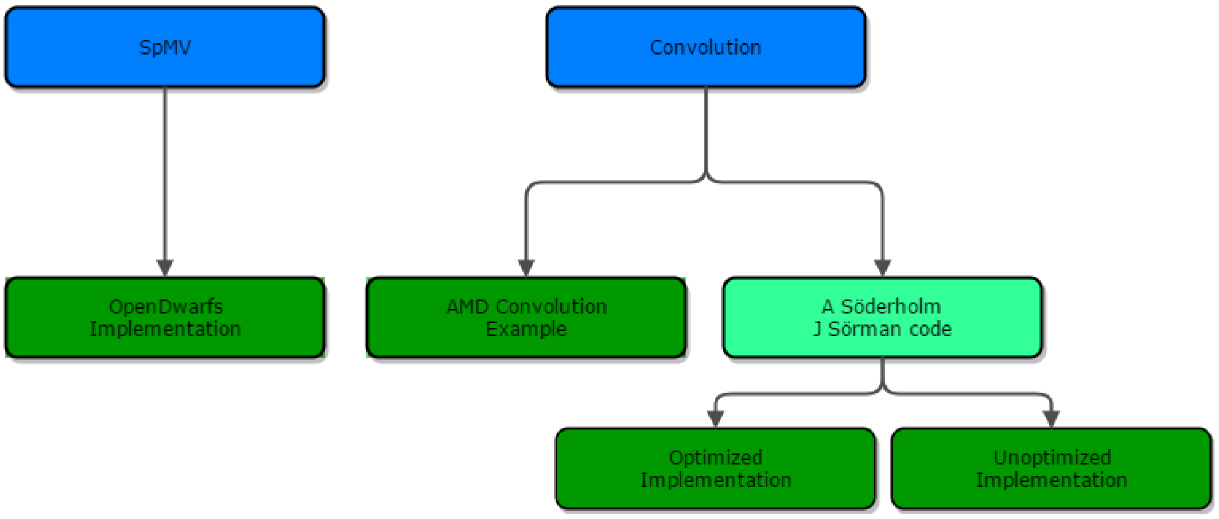

SpMV
Convolution
OpenDwarfs Implementation
AMD Convolution Example
A Söderholm J Sörman code
Optimized Implementation
Unoptimized Implementation

Figure 1: Blue boxes are the types of benchmark, the turquoise box is the code base it comes from and green boxes the actual implementation.

In this section the implementations will be describe as well as the implementation procedure.

3.3.1 Implementation procedure

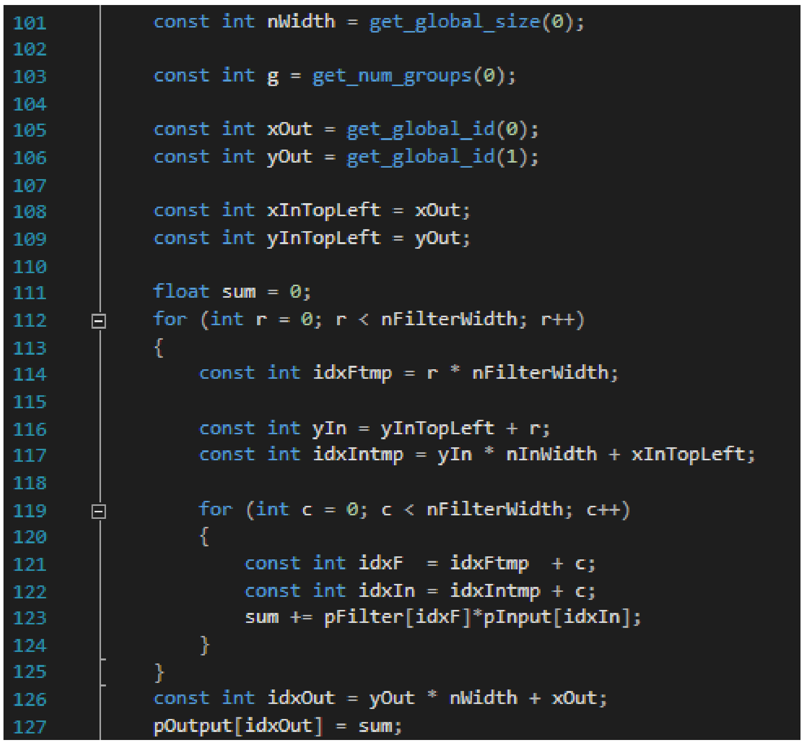
While none of the benchmark applications has been written from scratch by the authors they all needed some modification, because none of them were written for the exact system used by us. The AMD implementation was written for AMDs OpenCL implementation while we used Intel's, hardly any changes were needed because the implementations are very similar. However one or two functions during the OpenCL setup had to be changed to the Intel equivalent, but this should not affect the performance of the kernel in any major way.

The benchmark application from the OpenDwarf's suite needed more modification because they were written for Linux while we used Windows. In order to avoid as much Linux specific code as possible the individual SpMV application were singled out and separated from the rest of the suite. However once again all the Linux specific code (using Linux specific libraries etc.) was located in the setup of the program which should not affect the performance of the OpenCL kernel. Although, there is one difference from running the original benchmark application, being OpenDwarf's has their own kernel timers which are based on Linux functions such as **gettimeofday(),** we instead used OpenCL's built in event kernel timers.

3.3.2 AMDs Convolution kernel

AMD provides a comprehensive convolution example in OpenCL which is mainly built for CPUs. In this example there are multiple OpenCL kernels each with its own type of optimizations for the CPUs as well as a standard kernel without any optimizations. There is also a standard convolution written in C which is used to validate the results of the kernels. However the only OpenCL kernel that we actually used in this thesis is the standard convolution in order to avoid favouring any of the two platforms.

In the ordinary C implementation the convolution consists of four nested for loop. The two outer loops iterates over the pixels in the image while the two inner loops iterates over each pixels filter area and performs the actual calculation. AMDs standard OpenCL convolution kernel showed in figure 2 is very similar to their C convolution. The only important difference is that in the OpenCL implementation one kernel handles one input pixel each, meaning that the two outer loops can be left out. The correct input pixel is chosen for each thread depending on the threads global index which correlates to the index of the pixel in the image. The pixels corresponding filter is then iterated over and the calculations are performed.



```
const int nWidth = get_global_size(0);

const int g = get_num_groups(0);

const int xOut = get_global_id(0);
const int yOut = get_global_id(1);

const int xInTopLeft = xOut;
const int yInTopLeft = yOut;

float sum = 0;
for (int r = 0; r < nFilterWidth; r++)
{
    const int idxFtmp = r * nFilterWidth;

    const int yIn = yInTopLeft + r;
    const int idxIntmp = yIn * nInWidth + xInTopLeft;

    for (int c = 0; c < nFilterWidth; c++)
    {
        const int idxF  = idxFtmp  + c;
        const int idxIn = idxIntmp + c;
        sum += pFilter[idxF]*pInput[idxIn];
    }
}
const int idxOut = yOut * nWidth + xOut;
pOutput[idxOut] = sum;
```

Figure 2: AMD convolution kernel:

3.3.3 A Söderholm and J Sörman's Convolution kernel

A Söderholm and J Sörman [2] developed two convolution kernels, one standard and one which is optimized for GPUs in that it utilizes the GPUs local memory.

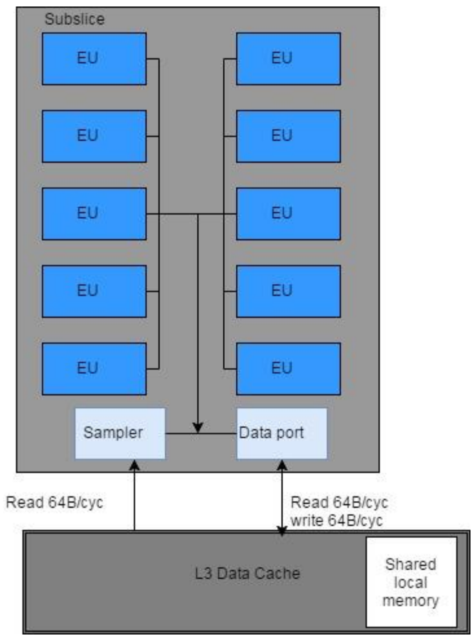
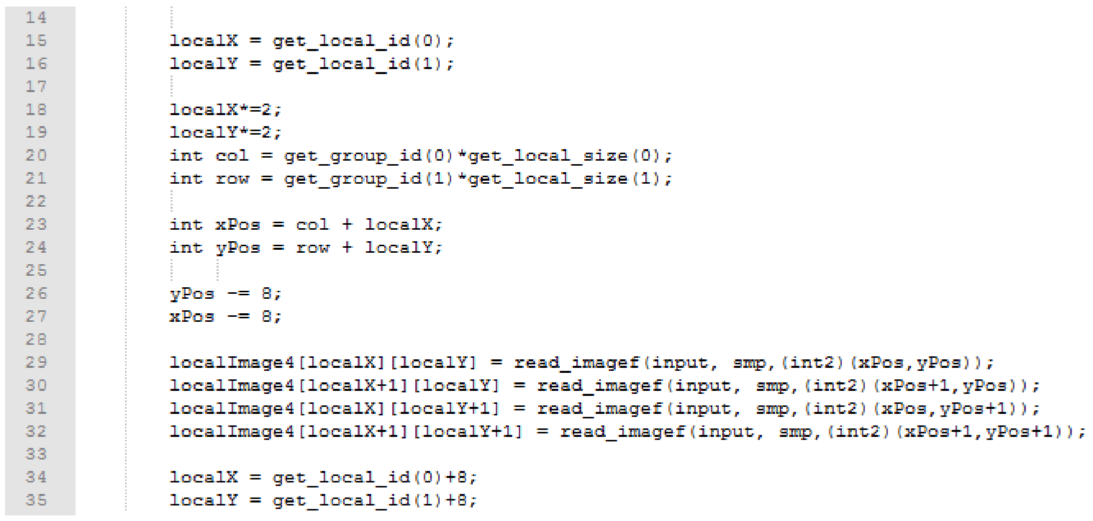


Figure 3: EU memory access

As mentioned earlier the local memory is faster than the global memory, however as you can see in figure 3, it resides within the level 3 cache, meaning it has the same bandwidth as rest of the cache. The reason for that the local memory is faster is that it is more highly banked. This means that it will achieve max bandwidth even when the data is not 64-byte aligned or contiguously adjacent in memory.

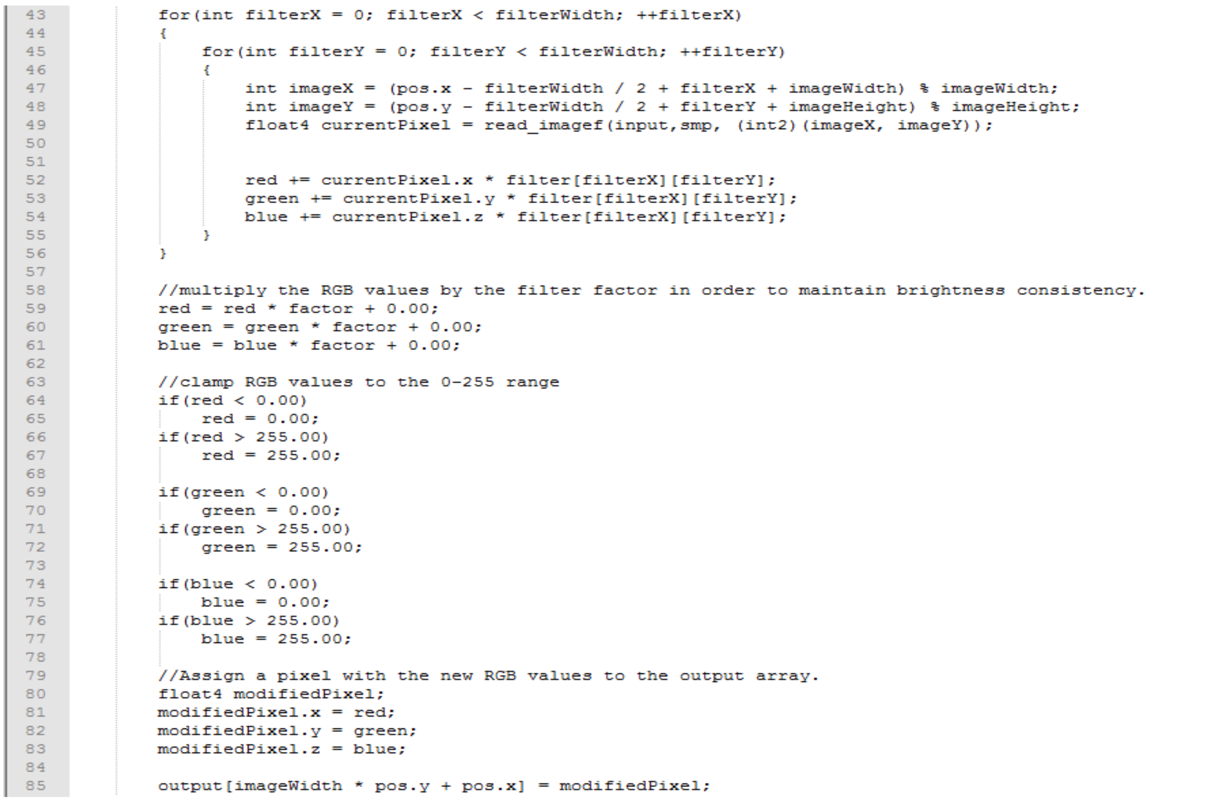
In order to utilize the local memory the first thing the optimized version does is to load the image to local memory, this is shown at line 29-32 in figure 4. Each work-group shares a 32x32 array in local memory and each kernel loads four pixels to this array. Then the operations are performed in the local memory instead of the global memory.



```
14
15  localX = get_local_id(0);
16  localY = get_local_id(1);
17
18  localX*=2;
19  localY*=2;
20  int col = get_group_id(0)*get_local_size(0);
21  int row = get_group_id(1)*get_local_size(1);
22
23  int xPos = col + localX;
24  int yPos = row + localY;
25
26  yPos -= 8;
27  xPos -= 8;
28
29  localImage4[localX][localY] = read_imagef(input, smp,(int2)(xPos,yPos));
30  localImage4[localX+1][localY] = read_imagef(input, smp,(int2)(xPos+1,yPos));
31  localImage4[localX][localY+1] = read_imagef(input, smp,(int2)(xPos,yPos+1));
32  localImage4[localX+1][localY+1] = read_imagef(input, smp,(int2)(xPos+1,yPos+1));
33
34  localX = get_local_id(0)+8;
35  localY = get_local_id(1)+8;
```

Figure 4: Write to local memory in optimized kernel

In both the optimized and unoptimized kernel, the image is stored in a two dimensional float4 array meaning that at each index one pixel is stored and each pixel consists of 3 RGB values, one for red, one for green and one for blue. After the image has been loaded the computation is performed by simply looping over the filter and image, multiplying the corresponding pixel values together and adding the results to one value for each RGB value. After the computation they make sure that the RGB values are between 0-255. Lastly the goal pixels RGB values are given the resulting values of the summations and multiplications and it is placed in the output. This process is showed in figure 5.



```
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;
```

Figure 5: Convolution computation

However one thing in their setup code was changed, namely the time measuring code, they used C++ high_resolution_clock to measure the kernel which seemed to give weird results. For example when running the original code on the GPU with a 480p image and a 3x3 filter both VTune (OpenCL profiler described in section 3.6) and the built in OpenCL timers returned a total time of 0,009 seconds, while the high_resolution_clock returned a total time of 0,06 seconds. Since high_resolution_clock is a standard C++ timer it measures the time it takes for the CPU to get from the code line before the kernel invocation to the code line after the kernel invocation. This is not necessarily the same amount of time as the GPU executes and sends data. Therefore we used OpenCL's built in profiling timers which measures the actual GPU execution time, as well as the data transfer times giving extra and more accurate information. However since data transfer from the CPU is not needed when the computation is done on the CPU the data transfer was only taken in to consideration when computing on the GPU.

They also have a specific setup code for testing 8k images since this GPU's OpenCL Image2d memory was capped to 4013x4014 pixels. Because of this they load the image to four different memory objects and each part is computed on its own and lastly the individual results are combined.

3.3.4 SpMV in Open Dwarfs

Open Dwarfs has one implementation of SpMV in OpenCL using the compressed sparse row format. In the CSR format (illustrated in figure 6) you take all the non-zero elements from the sparse-matrix and place them in a dense vector. Moreover a vector with the column index of each element and a vector that specifies at which position each row starts in the dense vector are created. The last entry in the vector specifies the position where the next row would start if there were any more values. Open dwarfs has a separate program that generates CSR formats, these are then supplied as input to the SpMV program which in turn specifies the vectors and sends them as input to the kernel.

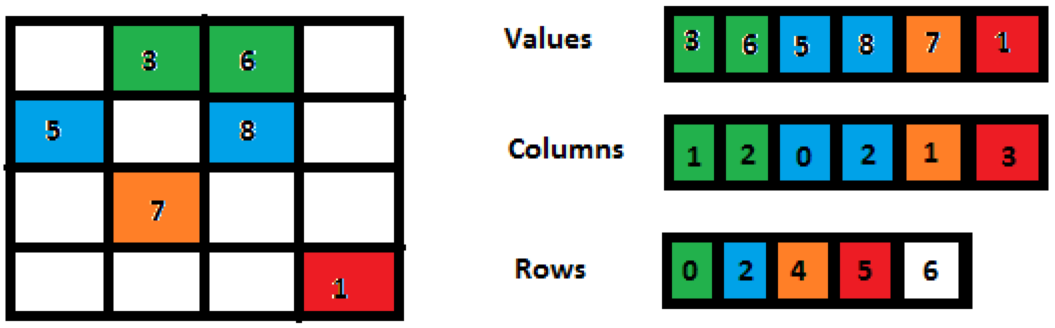
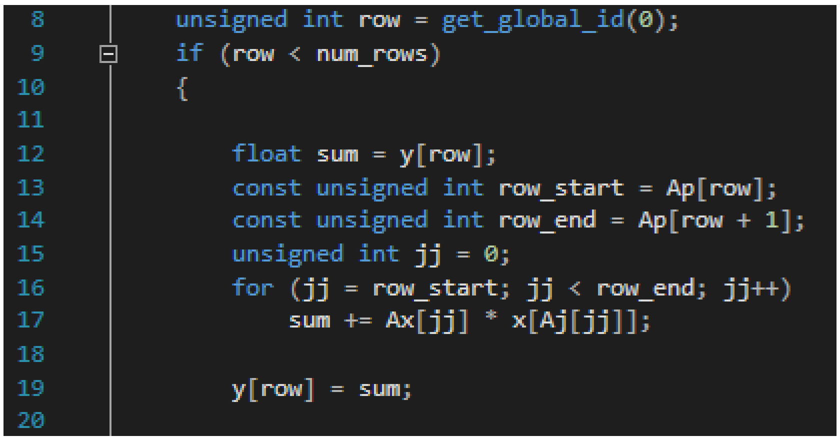


Figure 6: The CSR format

In the OpenCL kernel (shown in figure 7) each work-item is given the row corresponding to its global index and the indexes of the row is taken from the vector with row indexes. The start value for the vector (the start of the row in the dense vector) is stored at the corresponding position to the work-items index and the end value (the start of the next row in the dense vector) is stored at the next position in the vector. Then that part of the dense vector is iterated over and the matrix value is multiplied with the corresponding vector value by using the stored column index. Finally, the individual products of one row are summed up and saved in the results vector.



```
    unsigned int row = get_global_id(0);
    if (row < num_rows)
    {

        float sum = y[row];
        const unsigned int row_start = Ap[row];
        const unsigned int row_end = Ap[row + 1];
        unsigned int jj = 0;
        for (jj = row_start; jj < row_end; jj++)
            sum += Ax[jj] * x[Aj[jj]];

        y[row] = sum;

```

Figure 7: The SpMV kernel

3.3.5 Data transfer

In 3.2.2-3.2.4 the OpenCL kernels have been described since that's the code which mainly impact the performance. However, all OpenCL programs has quite a lot of setup much of it is more or less the same standard boilerplate code needed by all OpenCL programs. But when working with integrated graphics there are two ways to handle the data transfer from CPU to the GPU. Either, you simply send it from the CPU to the GPU as you would when using a dedicated GPU and thereby using the GPU's specific memory space. Otherwise, you can skip the data transfer and instead use the CPU's memory space. This is possible since the GPU is located on the same hardware die as the CPU which means it has access to the CPU's memory as well.

A Söderholm and J Sörman and the OpenDwarf's benchmark application transfers the data while the AMD example uses the CPU's memory. If you do not transfer the data you of course avoid the data transfer time however in some cases when the data is not transferred the computation takes a bit longer. A Söderholm and J Sörman's code is such a case, it probably has to do with the fact they handle an actual image which the GPUs sampler is specifically built to handle. The OpenDwarf's benchmark application transfers the data so that the code can run on both dedicated and integrated GPUs. We did not manage to modify the code to use the CPU's memory therefore we do not know if there is any speed increase or not when the data is located on the GPU. The AMD example is originally built for CPUs which is why it does not transfer data. We tested to transfer the data but there was not any significant speedup.

3.4 Problem Configurations

Since one of the research questions is to examine which factors that impact the performance of the application on the two different execution units, the applications will have to be tested with several configurations. In this section first all the different configurations, which will be tested, are presented. Then the general configurations that impact all applications are explained in detail and lastly the application specific ones are explained.

3.4.1 Configuration summary

The following configurations (illustrated in figure 8) will be tested.

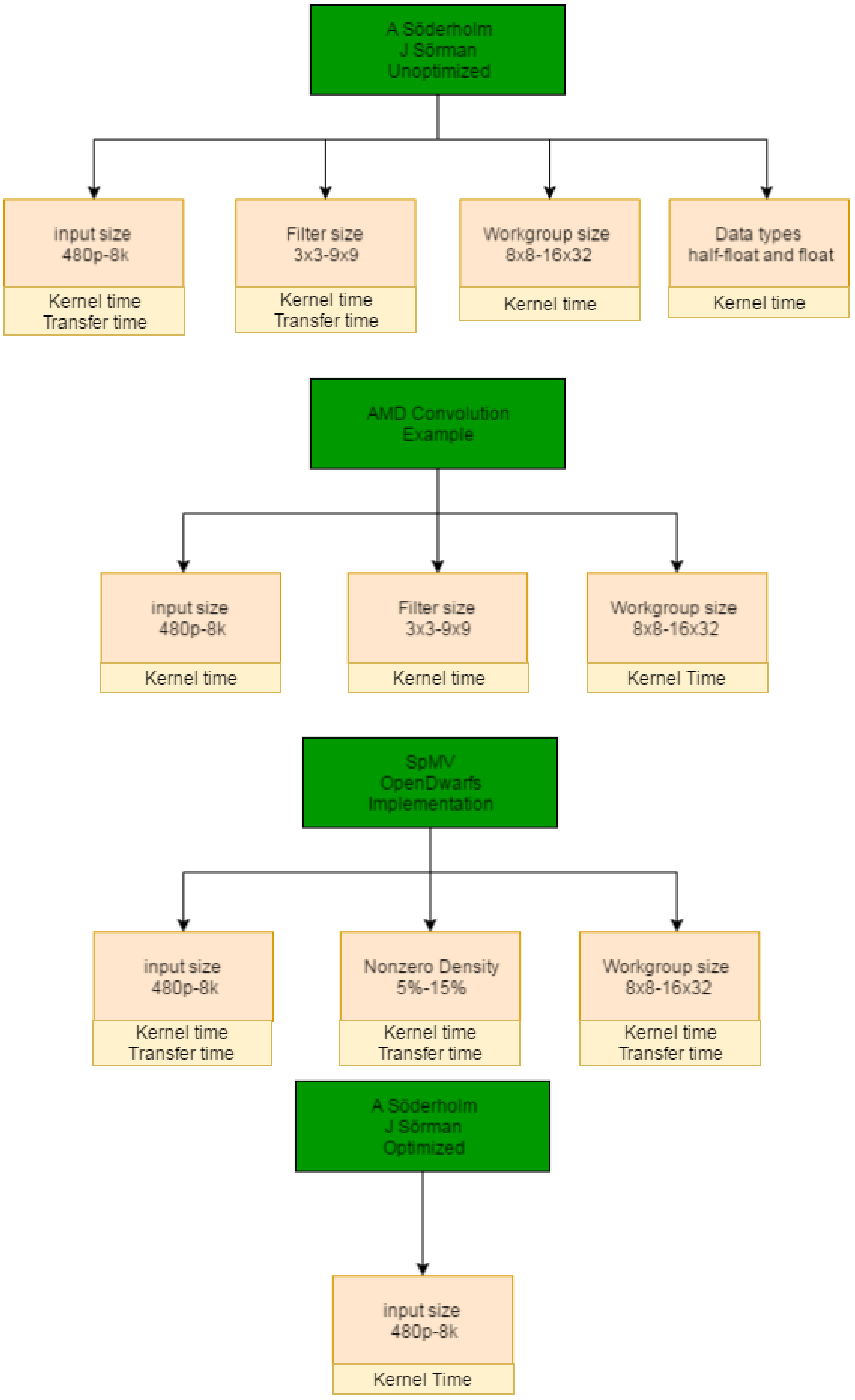


Figure 8: The green boxes are the implementations the orange boxes are the tests, the yellow ones details what part of the code that is timed.

As you can see only one of the parameters are tested on the Optimized version. This is because the optimized version is not important to the parameter tests at large. Since as specified in section “3.1 Benchmark Application Selection Factors” optimized versions gives more information about the specific implementation instead of the hardware. The reason it is tested is instead to examine A Söderholm and J Sörman’s [2] results in which case, altering the input size is enough.

In the following sections the parameters are explained.

3.4.2 General configurations

The general parameters of the applications are the following.

- The data type of the input.
- The size of the input.
- The work-group sizes

The data type could impact the performance of all of the applications, since they all perform mathematical operations on their data and bigger data-types take longer for the execution unit to operate on. Since a larger data type will mean more sequential execution, it should execute faster on the CPU compared to the GPU. However as noted in figure 7 the data type is only tested on the unoptimized implementation. This was because it was the only implementation where you could change the data type without breaking the computation, why is explained section “3.4.3 Application Specific Configurations”.

As was stated in the introduction, B.R. Payne [6] showed that in convolution on their GPU the speedup depended logarithmically on the input size. More specifically meaning that the relationship GPU_time(x)/CPU_time(x) is described by the function f(x) = k*$\log_2 x$ where x is the input size and k is a constant. Moreover bigger input should in general be better for the GPU. While this should be true for our convolution implementations it is probably not true for the SpMV implementation used in this thesis, since each kernel gets an entire row to compute. This means that the bigger the problem gets not only are more work-items needed but each work-item will have to compute a bigger chunk of data. More work-items are favoured by the GPU but bigger work-items are favoured by the CPU. Moreover since the CPU is about 3-4 times faster than the GPU the larger work-items will most likely cancel out the benefit the GPU gains from the larger amount of work-items. This means it would be a linear relation between the CPU-GPU execution time favouring the CPU as a function of the input size. I.e. GPU_time(x)/CPU_time(x) is described by f(x) = k*x where x is the input size. Since the input size increase equals a linear increase in the amount of operations performed by each kernel and a linear (but smaller) increase of the amount of kernels.

This means that larger input for SpMV should be executed faster on the CPU compared to on the GPU.

Lastly, the workgroup size: the workgroup size is the parameter through which you can impact the utilization degree (how many work-items that work concurrently) on the GPU and the CPU. If the workgroup-size is not chosen with care the hardware could be underutilized which of course impacts the performance. The number of work-groups is the input size divided by the work-group size, meaning the bigger the work-groups the fewer there are. Intel states that the number of work-groups should always be greater than the number of execution units since a work group can in theory be executed on one EU. They also state that they should be as large as possible in order to reduce overhead and that the size should be a multiple of 8. If it is a multiple of 8 the work-groups can utilize the built in vectorization module making it easier to parallelize. Intel generally recommends a size of about 64-128 work-items. As long as the sizes does not deviate to far from those sizes (not bigger than 256) and is devisable by eight the speedup should not vary with more than 0.05 units.

The sizes of the work-groups tested are the same for all the applications. They are 32x16, 16x16 and 8x8. 16x16 is the size which was used in the former thesis, 8x8 and 32x16 are used because we want one size quite a lot bigger size then Intel's recommended as well as one of Intel's recommended sizes. This means that as long as 32x16 is the only size impacting the performance of the applications in any significant way the hypotheses holds. If 8x8 or 16x16 impact the performance in a significant way the hypothesis does not hold.

3.4.3 Application specific configurations

Convolution

The input used for A Söderholm and J Sörman's [2] implementation are images, however what kind of images they are (what they depict) does not have any real impact on the performance as long as they are standard pictures (i.e. not any extremes such as an entirely black picture which would mean that all the pixel values would be 0). This is because the convolution consists of a number of addition and multiplications and the images are represented in some sort of array of pixel values, each value representing what colour the picture has. This means that different pictures will only result in different numbers to perform the same operation on and it does not take noticeably longer for a computer to add/multiply two number of different sizes as long as they are of about the same magnitude. However, if there are many pixels with the value zero in one image and in another all pixels have the value 255 there could potentially be a computational difference. This means besides standard images, which should not have any specific impact on the performance, images either resulting in an array of only zeros or in arrays with only 255's will be tested.

The AMD implementation is in a way a more theoretical implementation than the two others, by which we refer to the fact that it does not perform the operation on actual images, but instead on a float array with randomized values. This should however not be a problem since as stated above the exact values should not impact the performance (i.e. what an image depict should not impact the performance).

A specific parameter to test for convolution is the size of the filter, because it equals more operations. The larger the filter is the faster the code will run on the CPU compared to the GPU [6]. This is because each kernel will have more work to do which means you will gain more from single core speed which the CPU has more of [3]. The time increase should be more or less linear to the filter size increase, since a larger filter equals a linear increase in sequential operations. I.e. the relationship GPU_time(x)/CPU_time(x) is described by f(x) = k*x where x is the filter size.

The sizes of the tested inputs are 480p (853x480), 720p (1280x720), 1080p (1920x2080), 1440p (2560x1440), 4k (3840x2160) and 8k (7680x4320) and the filter sizes 3x3, 5x5 and 9x9. This enables us to compare our results with the former thesis results, because they used the same sizes.

The data types which are tested are cl_float, cl_half_float, because they are of distinct sizes which is needed to test our hypothesis regarding data-types. cl_float and cl_half_float are specific image channel types which can be used in the former thesis convolution example since they use real images. However in the AMD example you cannot test different data types because it uses an ordinary float array. If it is changed to an int array the operations fail and the intel HD 4600 does not support double precision operations (using double instead of float) therefore only float have been tested.

SpMV

As described in section "3.1.5 SpMV" in OpenDwarf's the CSR format is used as input for the application because the OpenCL kernel is specifically written for the CSR format. Moreover the CSR format is one of the most widely used formats in SpMV implementations and it gives an acceptable result [33].

The CSR formatted vectors are created by an independent application from OpenDwarf's which outputs a text file with the generated numbers. When it creates the CSR format it also makes sure that the nonzero numbers are distributed equally between the rows to ensure good load balancing. This file is used as input by the SpMV application.

A specific parameter for SpMV that is tested is the density of non-zero values in the matrix. The reason that this can have an effect on the performance is that a higher density equals more operations.

Our hypothesis is that a denser matrix will equal a faster execution speed on the CPU compared to on the GPU. It should be a linear relation since what happens with a denser matrix is that there will be more nonzero numbers on the same row. A density increase in SpMV affect the application much like a filter size increase in Image Convolution. It simply increases the amount of sequential operations performed by each kernel. More specifically it means that the relationship CPU_time(x)/GPU_time(x) is described by the function f(x) = k*x where x is the density and k is a constant. This reasoning is based on the simple fact that the CPU has a much higher clock speed and executes sequential instructions much faster [3].

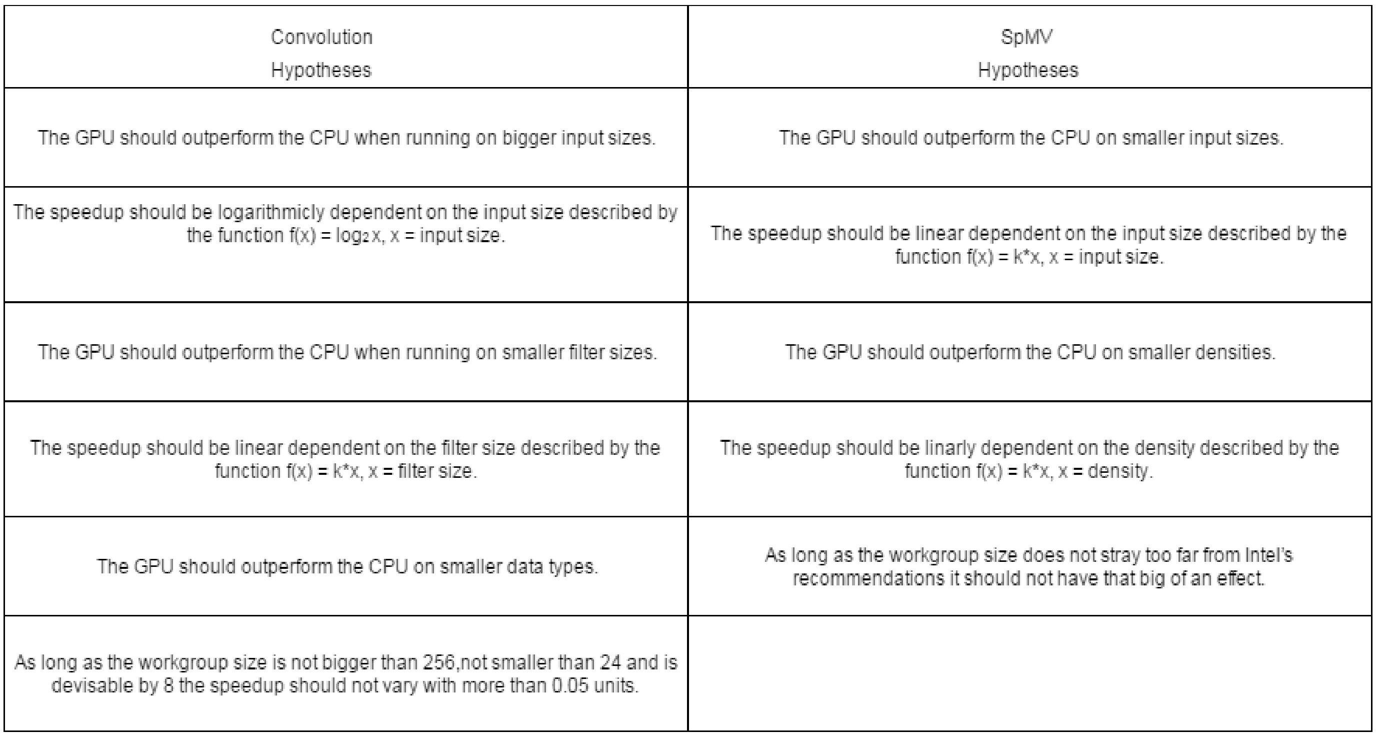
However based on S. Kim, I. Roy and V. Talwars [4] result the CPU should only outperform the GPU on the bigger density's and sizes.

The sizes which will be tested are 512x512, 1024x1024, 1536x1536 and 2048x2048, the densities of non-zero values being tested are 5%, 10% and 15%. These sizes and densities are used because they enables us to test our hypotheses that the density relation will be logarithmical and the input relation will be linear.

We were not able to test different data types on SpMV since a float type was needed and the Intel HD 4600 does not support the use of the type double (double precision) instead of float.

3.5 Hypotheses summary

Figure 9 shows all my hypotheses which are tested in the results.



Convolution Hypotheses	SpMV Hypotheses
The GPU should outperform the CPU when running on bigger input sizes.	The GPU should outperform the CPU on smaller input sizes.
The speedup should be logarithmicly dependent on the input size described by the function f(x) = log$_2$x, x = input size.	The speedup should be linear dependent on the input size described by the function f(x) = k*x, x = input size.
The GPU should outperform the CPU when running on smaller filter sizes.	The GPU should outperform the CPU on smaller densities.
The speedup should be linear dependent on the filter size described by the function f(x) = k*x, x = filter size.	The speedup should be linarly dependent on the density described by the function f(x) = k*x, x = density.
The GPU should outperform the CPU on smaller data types.	As long as the workgroup size does not stray too far from Intel's recommendations it should not have that big of an effect.
As long as the workgroup size is not bigger than 256,not smaller than 24 and is devisable by 8 the speedup should not vary with more than 0.05 units.	

Figure 9: Convolution and SpMV hypotheses

3.6 Analysis

The analysis consists of two parts, firstly check which implementation that is faster and secondly why it is faster.

The first part relies entirely on the execution time of the two implementations, which was measured with OpenCL's built in profiling tool. The OpenCL profiling functions measures the time it takes for a specified event to be completed. The events are bound to commands in the command queue, this means in practice that it is the commands that are timed. The commands we timed were the OpenCL kernel itself as well as the different data transferral commands, e.g. the reading and writing to the buffers sent between host and device.

The programs were run with the configurations specified in section "3.3 Problem Configurations". Each of the configuration for A Söderholm and J Sörman's code ran 5 times since there was not an especially big difference between runs. The AMD examples configurations were run 100 times since there was rather large differences between runs. The SpMV implementation configurations were run 10 times on the CPU and 5 times on the GPU since there was bigger variation the CPU. Out of the results from these runs a mean value for the data transferal, kernel and transferal + kernel time was calculated for each configuration.

The second part was conducted by using Intel Vtune Amplifier xe in order to locate the performance bottlenecks in the code relating to the different hardware features. Intel VTune amplifier is a performance measuring tool geared towards multicore applications for Intel CPUs and GPUs. VTune has a variety of analysis types, each focusing on a specific set of measurements.

For the CPU analysis we mainly used two analysis types:

- Advanced hotspots, which focuses on the most time consuming parts of the code and can be used to analyse the OpenCL kernels.
- Memory access, which focuses on how bound the application is by the memory bandwidth.

However the main part of the analysis was conducted on the GPU runs since it was on the GPU that the programs did not behave as expected. The analysis being used were GPU Hotspots, which is quite thorough and focuses on the most time consuming parts of the code run on the GPU, (i.e. the kernels). Furthermore, it shows hardware related metrics such as memory speed and concurrency of the GPU execution units. These three analysis types were chosen because they are the ones available in VTune that shows relevant data. There are for example a performance characterization analysis and a locks-and-waits analysis, however they are not meant for OpenCL programs. This meant that the data they produced was completely irrelevant, this was true for all of the other analysis types.

These analyses were run on the most interesting problem configurations for each application. The problem configurations analysed are specified in chapter 4 Results. The different results were compared in order to see which of the variables that had the biggest impact on the execution time. Why these were the ones which had the biggest impact was checked in the analysis results as well as in the literature.

4. Results

In this chapter the results of all the tests will be presented.

4.1 Convolution

The presentation of the results will follow the structure of the method. This means that first the results of the time measurements will be presented and then the relevant results from the analysis of that test. In this section the results of the two Image Convolution implementations are presented.

4.1.1 A Söderholm and J Sörman Convolution code results

We begin with the results of the original non-modified code of A Söderholm and J Sörman [2]. The application code itself is unchanged, however the time measuring code is changed to OpenCL's profiling tools as mentioned in section "3.2.3 Previous thesis Convolution kernel". This change in itself solved the question why the ratio between the CPU execution and GPU execution that they observed stayed constant. They simply used the wrong timers which did not measure the times accurately enough.

Figure 10 and 11 showcases our results for the image size and filter size tests:

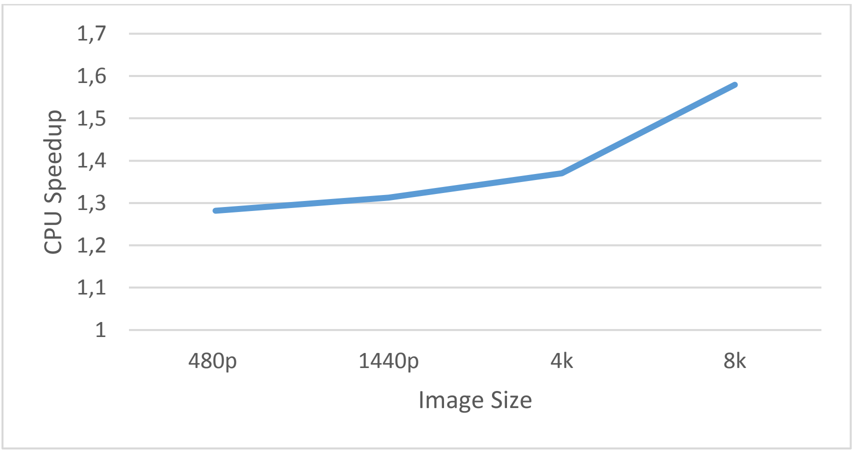


Figure 10: The CPU kernel speedup as a function of the image size with a 3x3 filter and 16x16 workgroup.

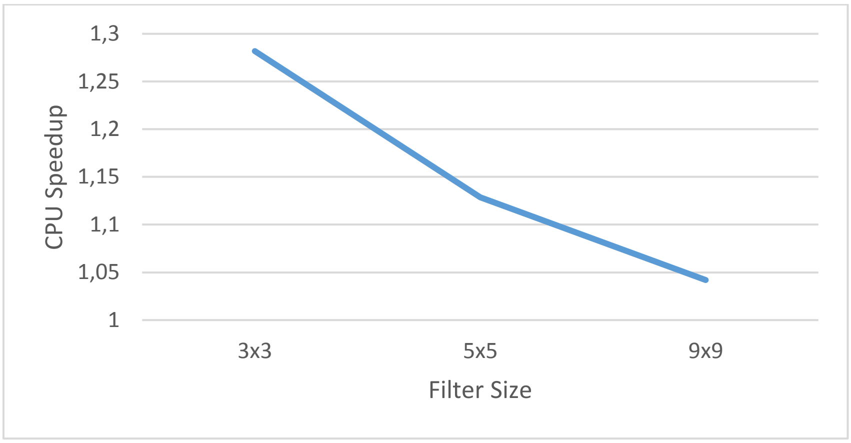


Figure 11: The CPU kernel speedup as a function of the filter size on a 480p image and 16x16 workgroup.

Just as in the previous thesis results the CPU is always faster, however you can clearly see in figure 10 and 11 that the results does in fact depend on both the filter size and the image size. However, the relation is the opposite of what is expected, instead of favouring the GPU on smaller filters and bigger images it favours the CPU.

If you take in to account the time it takes to send the data to the GPU the results just gets even better for the CPU. As shown in figure 12 and 13.

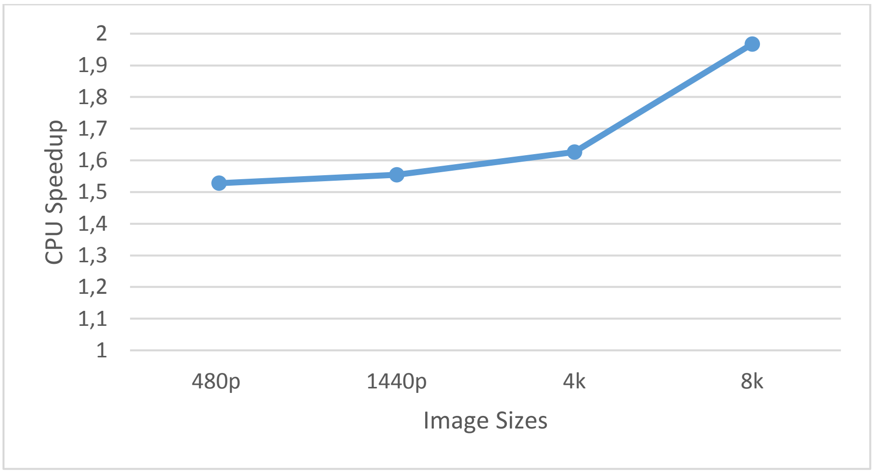


Figure 12: The CPU kernel speedup as a function of the image size with a 3x3 filter and 16x16 workgroup including data transfer time to the GPU.

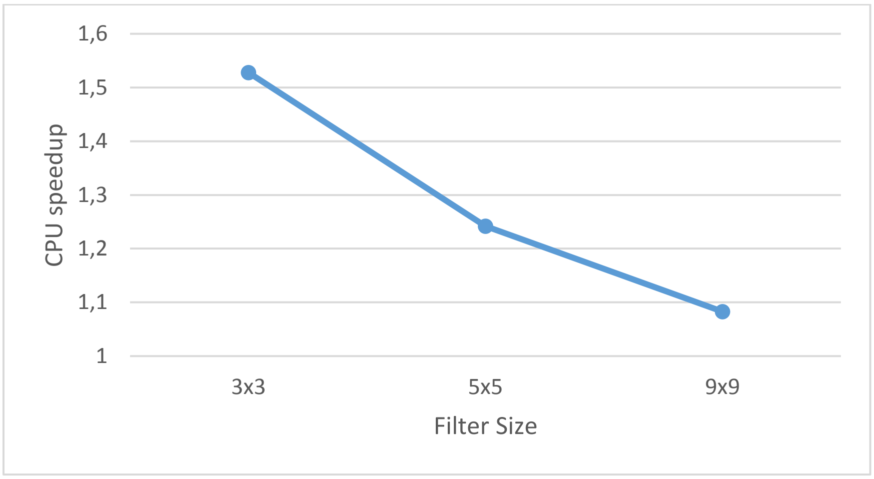


Figure 13: The CPU kernel speedup as a function of the filter size on a 480p image and 16x16 workgroup including data transfer time to the GPU.

Furthermore on both image size graphs (figure 10 and 12) we see that between 4k and 8k there is a much bigger difference than between any other of the sizes. In order to find out why that was the case the GPU hotspot analysis was run with a size of 4k and of 8k. The difference was that when the size was 4k only one kernel was needed to run the entire program, but on 8k 4 kernels which had to run sequentially was needed. Figure 14 and 15 shows the Vtunes graphics pipeline output on 4 input and 8k input respectively.

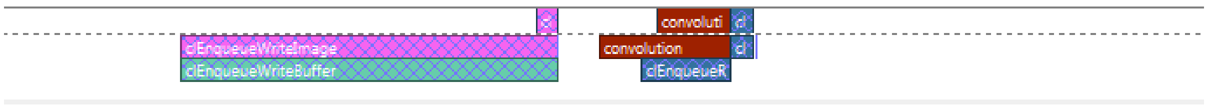


Figure 14: Intel VTune intelHD graphics pipeline showing only one convolution kernel on 4k.

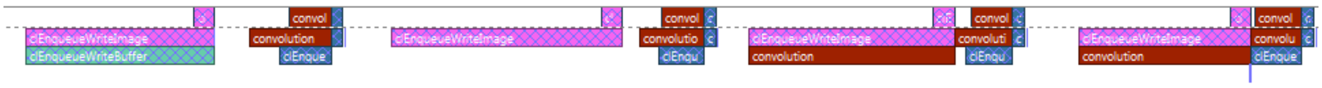
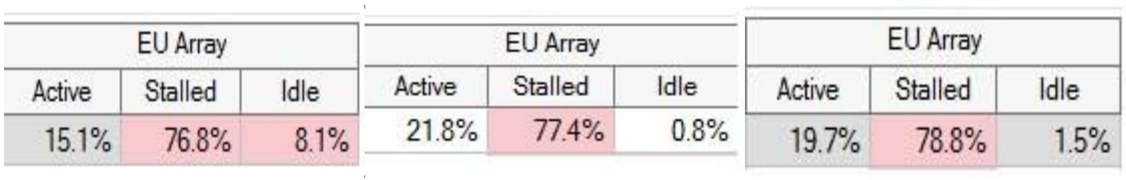


Figure 15: Intel VTune intelHD graphics pipeline showing four sequential convolution kernels on 8k.

Why this is the case is explained in the next chapter.

In order to examine why the application always ran faster on the CPU and the relations were the opposite of what was expected, the GPU hotspots analysis was run on a 480p image with the different filter sizes. The analysis showed that the execution on the GPU was significantly hampered by a high number of memory stalls. This meant that a significant amount of the execution time the GPU's execution units spent waiting to get access to specific memory locations. This can be explained by the fact that the different kernels will try to access the same memory locations in the image when computing the filter. However there were a very high percentage of stalls and the number of stalls were basically independent from the filter size (shown in figure 16) which should not be the case. Because the bigger the filter is the more memory locations are shared between the EUs, meaning that bigger filters should have more stalls than small filters.



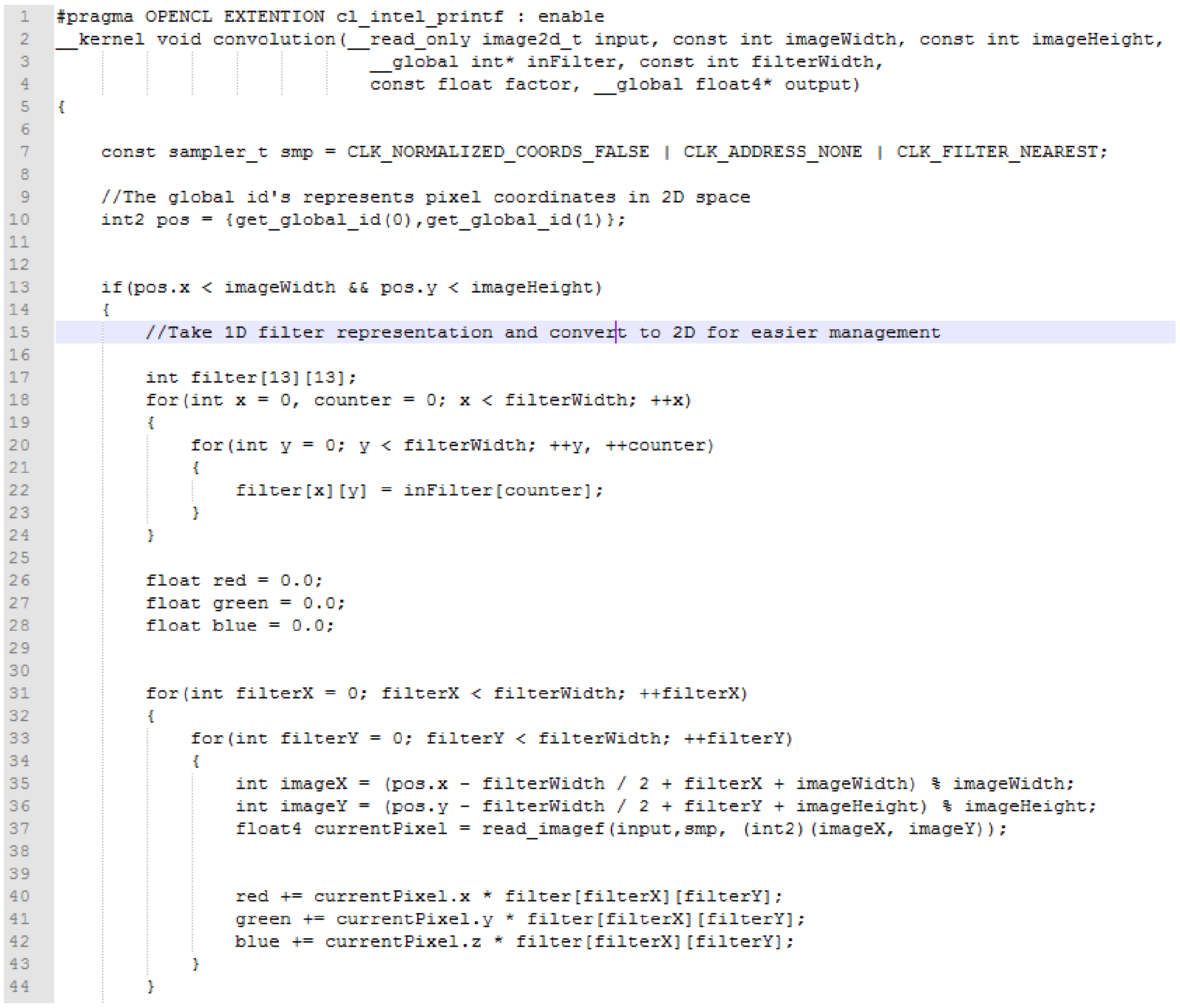
EU Array		
Active	Stalled	Idle
15.1%	76.8%	8.1%

EU Array		
Active	Stalled	Idle
21.8%	77.4%	0.8%

EU Array		
Active	Stalled	Idle
19.7%	78.8%	1.5%

Figure 16: Percentage of execution time which the GPU's EUs were stalled on a 480p image with the different filter sizes. From left to right: 3x3, 5x5 and 9x9.

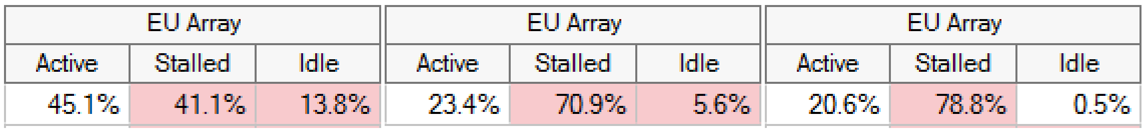
This caused a closer inspection of how they handled the filters in the kernel.



```
1  #pragma OPENCL EXTENTION cl_intel_printf : enable
2  __kernel void convolution(__read_only image2d_t input, const int imageWidth, const int imageHeight,
3                            __global int* inFilter, const int filterWidth,
4                            const float factor, __global float4* output)
5  {
6  
7      const sampler_t smp = CLK_NORMALIZED_COORDS_FALSE | CLK_ADDRESS_NONE | CLK_FILTER_NEAREST;
8  
9      //The global id's represents pixel coordinates in 2D space
10     int2 pos = {get_global_id(0),get_global_id(1)};
11 
12 
13     if(pos.x < imageWidth && pos.y < imageHeight)
14     {
15         //Take 1D filter representation and convert to 2D for easier management
16 
17         int filter[13][13];
18         for(int x = 0, counter = 0; x < filterWidth; ++x)
19         {
20             for(int y = 0; y < filterWidth; ++y, ++counter)
21             {
22                 filter[x][y] = inFilter[counter];
23             }
24         }
25 
26         float red = 0.0;
27         float green = 0.0;
28         float blue = 0.0;
29 
30 
31         for(int filterX = 0; filterX < filterWidth; ++filterX)
32         {
33             for(int filterY = 0; filterY < filterWidth; ++filterY)
34             {
35                 int imageX = (pos.x - filterWidth / 2 + filterX + imageWidth) % imageWidth;
36                 int imageY = (pos.y - filterWidth / 2 + filterY + imageHeight) % imageHeight;
37                 float4 currentPixel = read_imagef(input,smp, (int2)(imageX, imageY));
38 
39 
40                 red += currentPixel.x * filter[filterX][filterY];
41                 green += currentPixel.y * filter[filterX][filterY];
42                 blue += currentPixel.z * filter[filterX][filterY];
43             }
44         }
```

Figure 17: The beginning of the original kernel.

As you can see on line 3 in figure 17 they send their one dimensional filter (inFilter) to their kernel. Then as stated in the comment on line 15 they convert it to a 2D filter for easier management. However, when they created their 2D filter (line 17) they statically used the size 13x13 which was their largest filter size. When the exact size of the filters were used instead, (i.e. 3x3, 5x5 and 9x9) the percentage of stalls lowered dramatically for 3x3, a bit for 5x5, hardly anything for 9x9 (shown in figure 18) but the time measurement results were very different.



EU Array			EU Array			EU Array		
Active	Stalled	Idle	Active	Stalled	Idle	Active	Stalled	Idle
45.1%	41.1%	13.8%	23.4%	70.9%	5.6%	20.6%	78.8%	0.5%

Figure 18: Percentage of execution time which the GPUs EUs were stalled on a 480p image with the different filter sizes, after the code was modified. From left to right: 3x3, 5x5, and 9x9.

Figure 19 shows the new results for the image sizes.

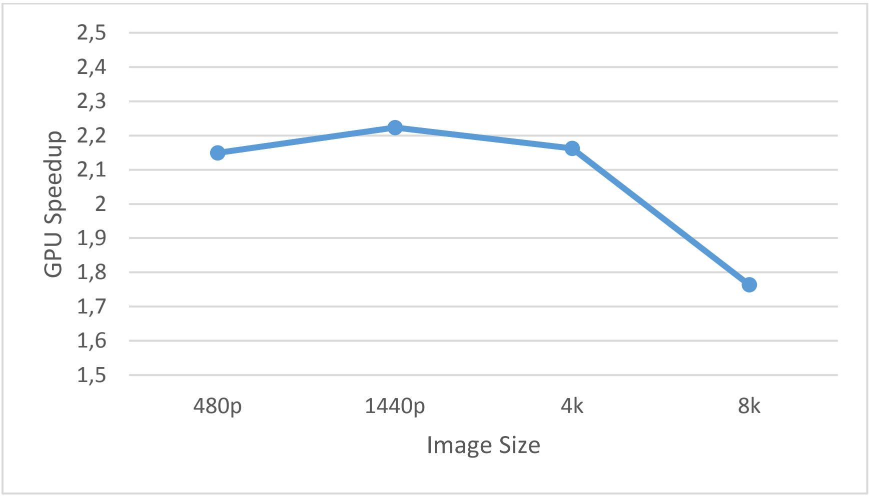


Figure 19: The GPU kernel speedup as a function of the image size with a 3x3 filter and 16x16 workgroup.

As you can see in figure 19 there is a small increase from 480p to 1440p and it is a very large step in size. Because of this 720p and 1080p was added in figure 20 in order to see if anything interesting happened in between.

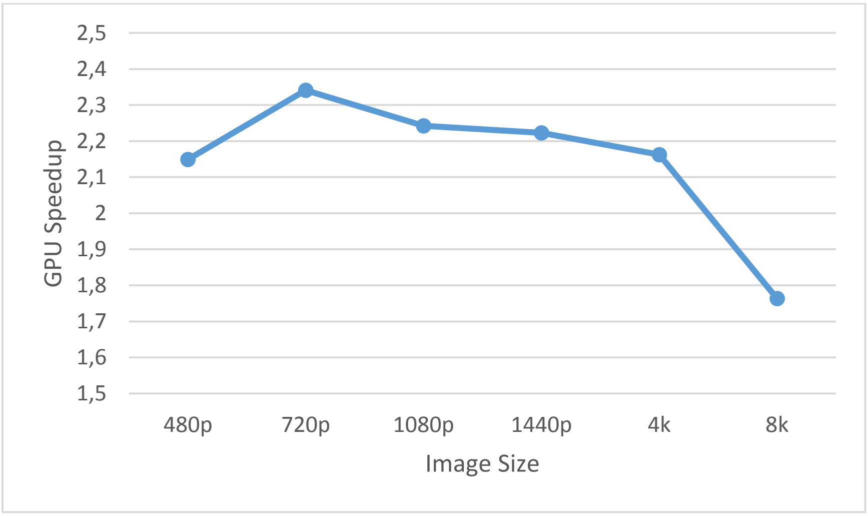
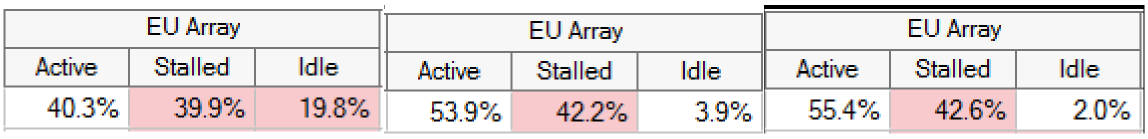


Figure 20: The GPU kernel speedup as a function of the image size with a 3x3 filter and 16x16 workgroup.

	480	720	1080	1440	4k	8k
CPU time	0,0055 s	0,0129 s	0,0281 s	0,0483 s	0,1056 s	0,4027 s
Time increase		2,31 times	2,17 times	1,71 times	2,18 times	3,81 times
GPU time	0,0026 s	0,0055 s	0,0125 s	0,0217 s	0,0488 s	0,2283 s
Time increase		2,12 times	2,27 times	1,73 times	2,24 times	4,67 times

Table 1: Kernel execution time and time increase on the different image sizes on the CPU and GPU using a 3x3 filter and a 16x16 workgroup.

As you now clearly can see in figure 20 the GPU is faster than the CPU, however the varying relation between the image size and the speedup still did not behave as expected. Table 1 shows clearly that the only size increase where the GPU is favoured is from 480p to 720p. Therefore, the GPU hotspot analysis was once again run with the sizes 480p, 720p and 1080p in order to see what differentiated the three sizes.



EU Array		
Active	Stalled	Idle
40.3%	39.9%	19.8%

EU Array		
Active	Stalled	Idle
53.9%	42.2%	3.9%

EU Array		
Active	Stalled	Idle
55.4%	42.6%	2.0%

Figure 21: Percentage of execution time which the GPUs EUs were stalled and idle with a 3x3 filter with different image sizes. From left to right: 480p, 720p, 1080p.

As you can see in figure 21, with a size of 480p the GPU is not fully utilized. 19% of the execution time, execution units are idle but if you increase the size to 720p more or less the entire GPU is utilized and at 1080p the utilisation is basically the same. This means that the performance gain is the largest on the GPU going from 480p to 720p, this will be explained further in the next chapter.

Figure 22 shows the results of the filter size tests.

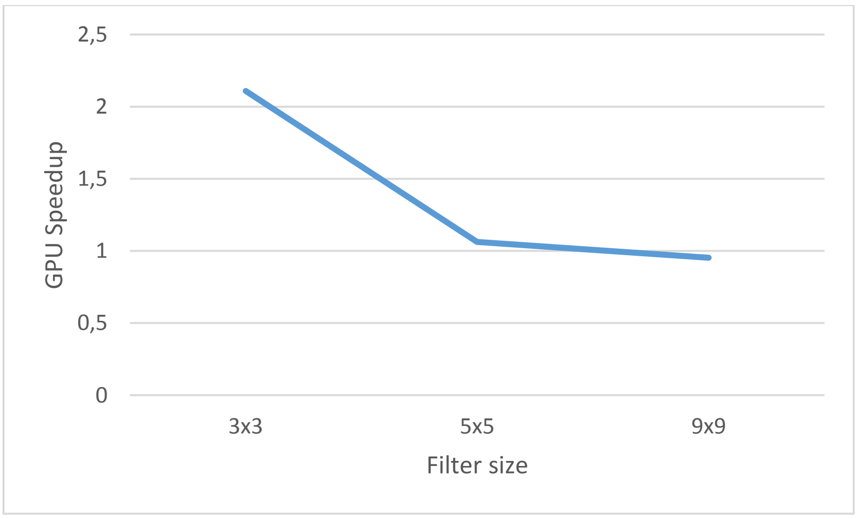


Figure 22: The CPU kernel speedup as a function of the filter size on a 480p image and 16x16 workgroup.

	3x3	5x5	9x9
CPU time	0,0055 s	0,0130 s	0,0372 s
Time increase		2,34 times	2,84 times
GPU time	0,0026	0,0115	0,0376
Time increase		4,42 times	3,27 times

Table 2: Kernel execution time and time increase when using different filter sizes on the CPU and GPU on a 480p image and 16x16 workgroup.

As expected the bigger filter sizes impact the GPU significantly more and on smaller sizes the GPU outperforms the CPU. Why the difference between 5x5 and 9x9 is so small is explained in the next chapter as well as the mathematical relation.

Figure 23 and 24 shows the same graphs as 20 and 22 but with the data transfer included and as expected some of the GPUs advantage is nullified.

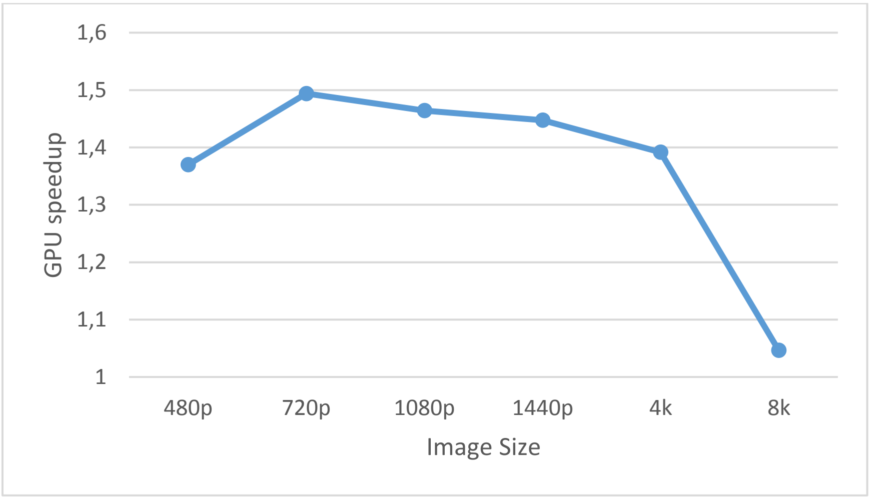


Figure 23: The GPU kernel speedup as a function of the image size with a 3x3 filter and 16x16 workgroup including data transfer time to the GPU.

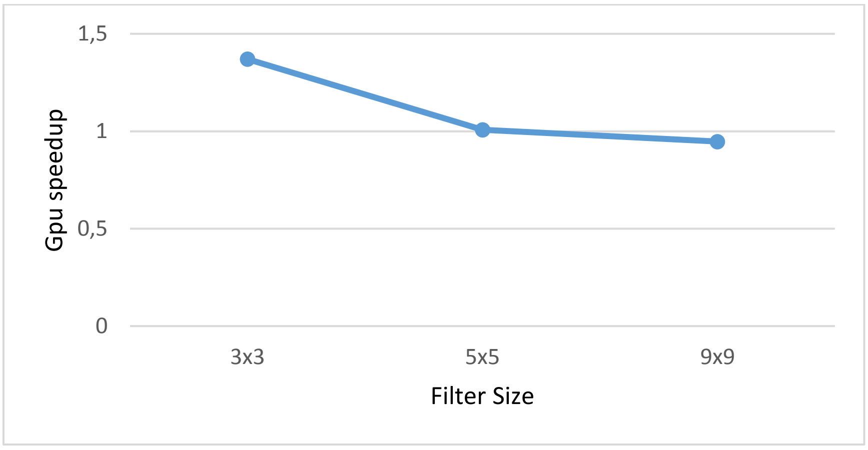


Figure 24: The GPU kernel speedup as a function of the filter size on a 480p image and 16x16 workgroup including data transfer time to the GPU.

	480p	720p	1080p	1440p	4k	8k
GPU time	0,0026 s	0,0055 s	0,0125 s	0,0217 s	0,0488 s	0,2283 s
Transfer time	0,0014 s	0,0031 s	0,0066 s	0,0116	0,0270	0,1566
Transfer/ GPU	0,56 %	0,56 %	0,53 %	0,53 %	0,55 %	0,68 %

Table 3: The GPU execution time and the transfer time as well as how big part of the time that is spent in transfer with a 3x3 filter and 16x16 workgroup.

As shown in table 3 the transfer time relation to the execution time stays more or less the same for different sizes.

A Söderholm and J Sörman [2] also made an “optimized” GPU version using the GPUs shared local memory in order to lessen the amount of accesses to global memory which is slower. However, their results showed that their optimization actually made the program run slower.

My results with their optimized version shown in figure 25 showed the same.

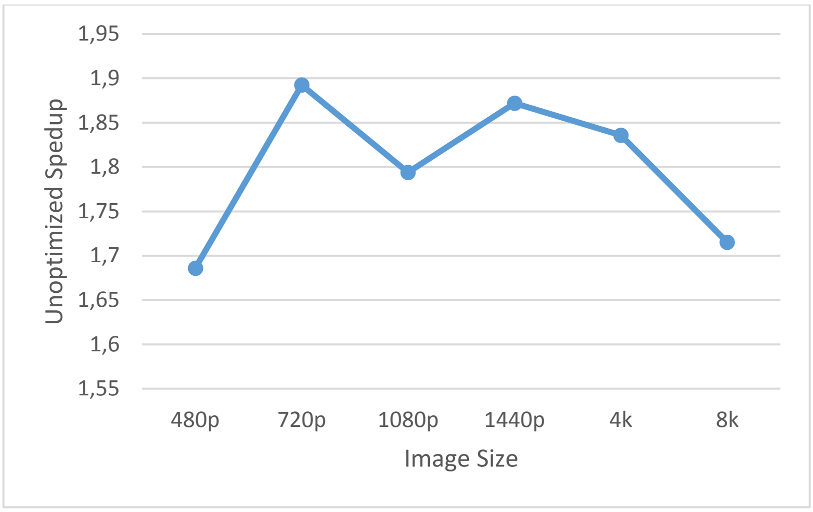
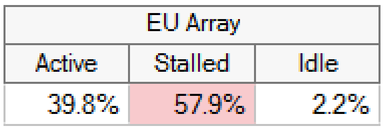


Figure 25: The GPU basic kernel speedup compared to the optimized GPU kernel as a function of the image sizes with a 3x3 filter and 16x16 workgroup size.

The reason for this is that the optimized version is stalled about 15% more of the execution time (showed in figure 26). Why this is the case is explained in the next chapter.



EU Array		
Active	Stalled	Idle
39.8%	57.9%	2.2%

Figure 26: Percentage of execution time which the GPUs EUs were stalled with a 3x3 filter and a 1080p image.

Finally, the results of the workgroup size and data types tests in figure 27 and 28 respectively.

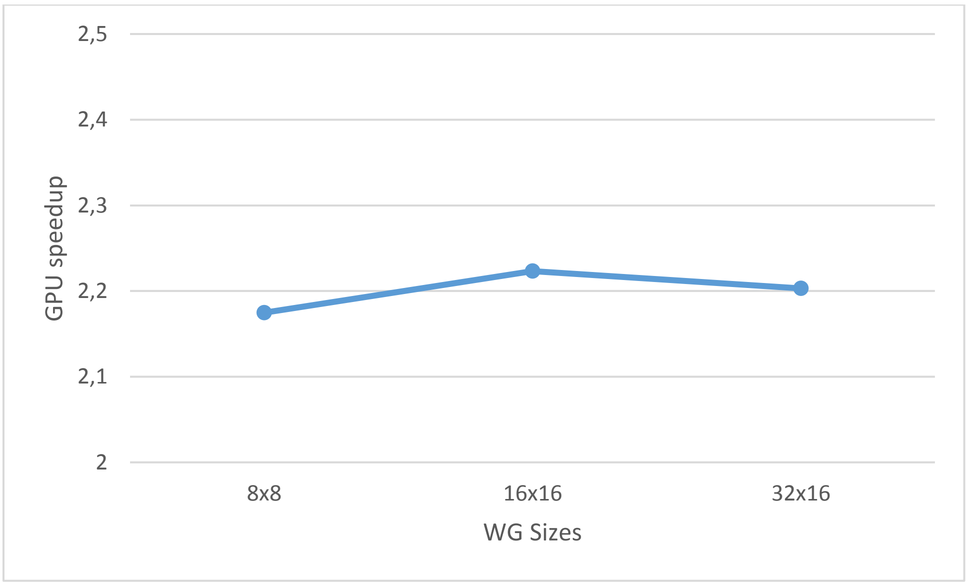


Figure 27: The GPU kernel speedup as a function of the workgroup size on a 1440p image and 3x3 filter.

When looking at figure 27, the scale should be noted, the speedup difference between the different workgroup sizes are very small.

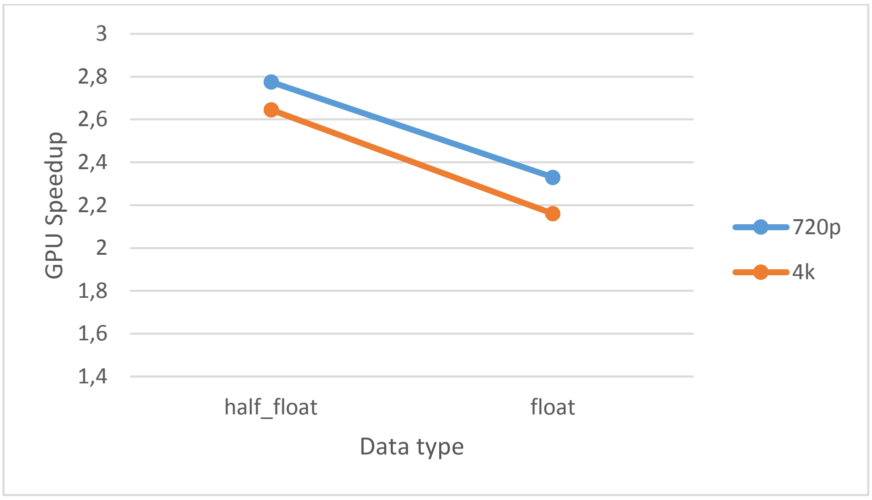


Figure 28: The GPU kernel speedup as a function of the data type on a 720p and a 480p image and 3x3 filter.

	Half float	float
CPU time	0,0135 s	0,0129 s
Time increase		0,95 times
GPU time	0,0048 s	0,0055 s
Time increase		1,13 times

Table 4: Kernel execution time and time increase on data types using a 3x3 filter on a 720p image and a 16x16 workgroup.

As expected, with a smaller data type the GPU performs better than the CPU and the ratio stays more or less the same for different image sizes. In fact with a bigger data type the CPU executes it faster as seen in table 4.

As specified in section “3.3 Problem Configurations” the intention was also to test the execution time on one entirely white image and one entirely black image in order to test the extreme cases. However, for some reason the program crashed when the black and the white image were used as input. Therefore the extreme cases are only tested on the AMD example.

4.1.2 AMD Convolution results

The AMD convolution example was used as a standard implementation to compare the previous convolution kernel to. In this section the results from the tests performed on the AMD example are presented. Worth to mention is that there are no data transfer times included in any of the AMD tests (se figure 8), being figure 29, 30, 31 or table 5, 6, 7 since the GPU uses the CPUs memory. Figure 29 and table 5 shows the results of the image size tests.

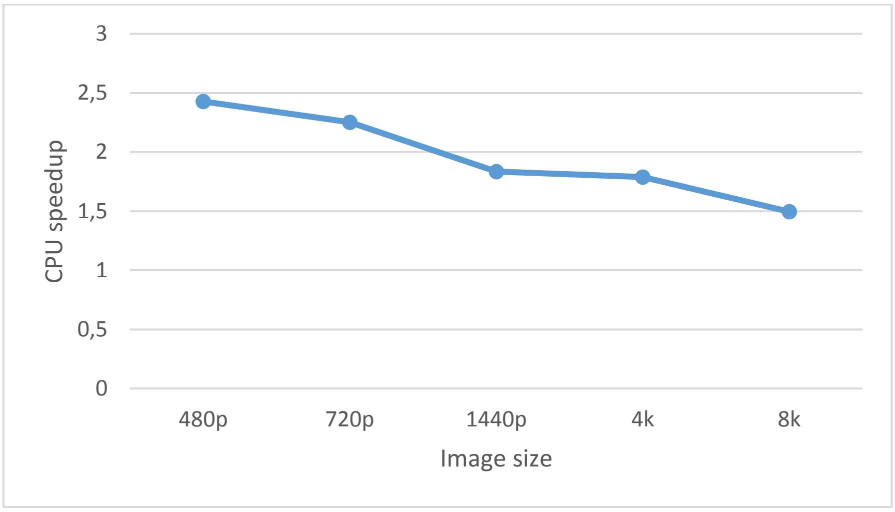


Figure 29: The CPU kernel speedup as a function of the image size with a 3x3 filter and 16x16 workgroup.

	480p	720p	1440p	4k	8k
CPU time	0,00026 s	0,00056 s	0,00270 s	0,00615 s	0,02685 s
Time increase		2,10 times	4,77 times	2,27 times	4,36 times
GPU time	0,00065 s	0,00127 s	0,00495 s	0,01100 s	0,04017 s
Time increase		1,95 times	3,88 times	2,21 times	3,65 times

Table 5: Kernel execution time and time increase on the different image sizes on the CPU and GPU using a 3x3 filter and a 16x16 workgroup.

The big difference from the same tests with A Söderholm and J Sörman's [2] code is that the CPU actually is faster. However, the relation between the speedup and the image sizes behaves as expected. In this case (figure 29) the 8k size fits on the GPU since it is only an array with ints instead of an image. Therefor it does not favour the CPU the same way as it did with A Söderholm and J Sörman's [2] code. And here too we can see that the larger image sizes impact the CPUs execution time more. Note that 1080p is not included in this test, therefore the time increase is so big from 720p to 1440p.

We continue with the filter size tests in figure 30 and table 6.

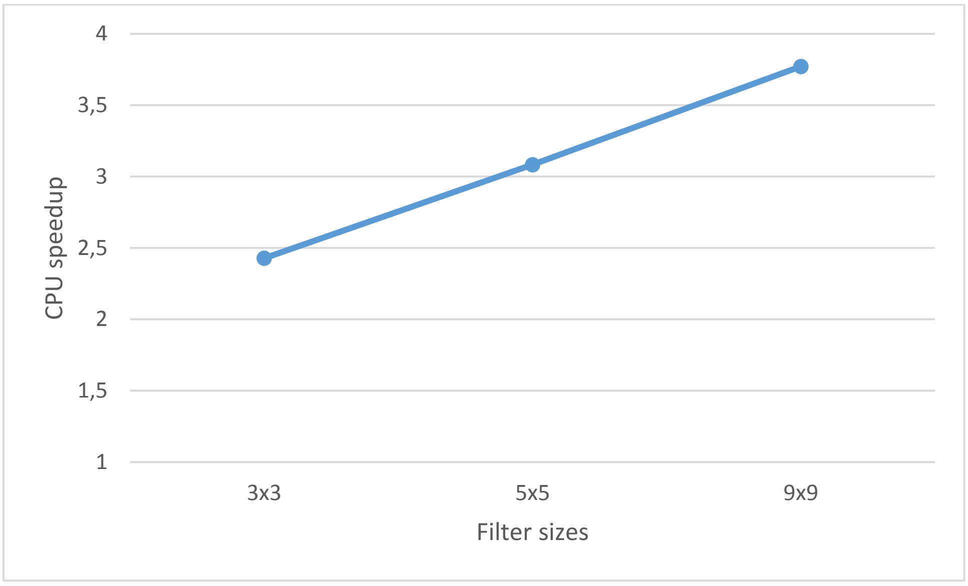


Figure 30: The GPU kernel speedup as a function of the filter size on a 480p image and 16x16 workgroup.

	3x3	5x5	9x9
CPU time	0,00026 s	0,00045 s	0,00110 s
Time increase		1,7 times	2,4 times
GPU time	0,00065 s	0,00141 s	0,00415 s
Time increase		2,16 times	2,94 times

Table 6: Kernel execution time and time increase when using different filter sizes on the CPU and GPU on a 480p image and 16x16 workgroup.

As you can see in figure 30 the CPU is faster than the GPU on all the different filter sizes but it follows the expected relation, the bigger the filter the better for the CPU and vice versa.

And on the next side in figure 31 you can see that the execution time is basically not impacted at all by the workgroup size.

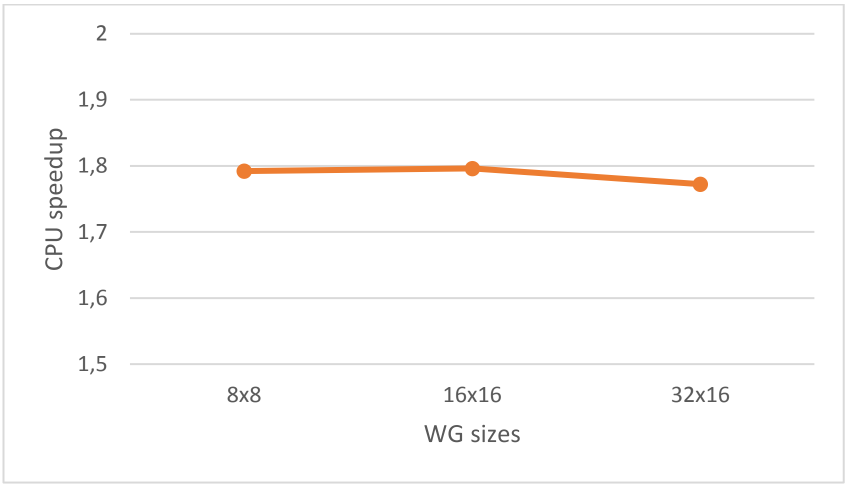
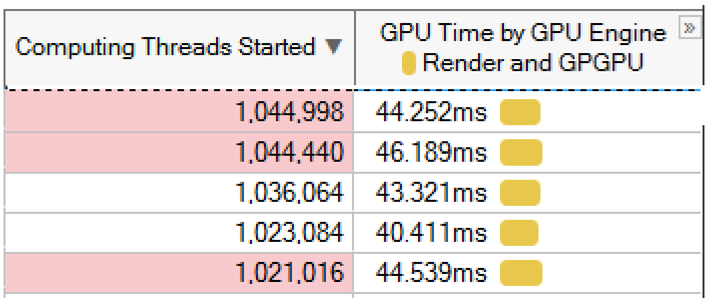


Figure 31: The GPU kernel speedup as a function of the workgroup size on a 1440p image and 3x3 filter.

The big difference (as mentioned before) from the other implementation is that the CPU runs the application faster. In order to examine why this is the case, the GPU hotspots analysis was used on the code to test all the different problem sizes. The only real problem that the analysis found was that the GPU sometimes created too many threads resulting in some overhead. However this overhead is not big enough to be the sole reason for the program executing faster on the CPU. As can be seen in figure 32 the difference in execution time is not that big when too many threads are started compared to when not too many threads are started.



Computing Threads Started ▼	GPU Time by GPU Engine Render and GPGPU
1,044,998	44.252ms
1,044,440	46.189ms
1,036,064	43.321ms
1,023,084	40.411ms
1,021,016	44.539ms

Figure 32: The amount of threads created on five different runs with an image size of 8k and their execution times, the ones marked in red is when too many are created.

Finally we also tested if extreme inputs had any impact on the performance, i.e. if the array only consisted of zeros or only consisted of the largest value possible generated by the random function (32 767 in this case).

	Random	Min	Max
CPU	0,00045	0,00044	0,00044
GPU	0,00135	0,00138	0,00138

Table 7: The execution times in seconds for filling the array with only min number, only max number and random numbers for the GPU and CPU. With a size of 720p, 3x3 filter and 16x16 workgroup size.

As you can see in table 7 there is hardly any difference at all, and the small difference there is, most likely comes down to the fact that the execution time is not deterministic.

4.2 SpMV

In this section the result of the SpMV implementation is presented. We begin with the Density tests in figure 33 and table 8.

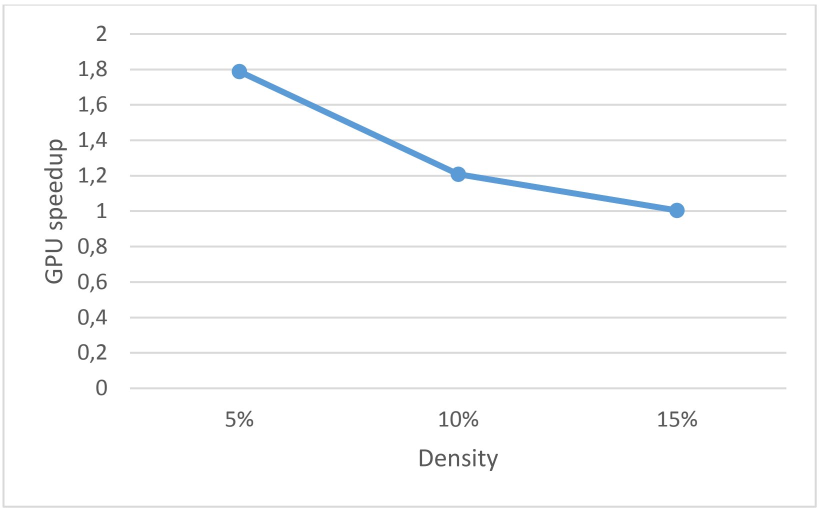


Figure 33: The GPU kernel speedup as a function of the nonzero density with an input size of 512 and a 16x16 workgroup.

	5%	10%	15%
CPU time	0,03534s	0,04107 s	0,04790 s
Time increase		1,16 times	1,15 times
GPU time	0,01976 s	0,03398 s	0,04728 s
Time increase		1,71 times	1,39 times

Table 8: The CPU and GPU kernel execution times and the time increase between the different densities.

As expected on lower densities the GPU outperforms the CPU the relation is discussed in the next chapter.

The input size tests is displayed in figure 34 and table 9.

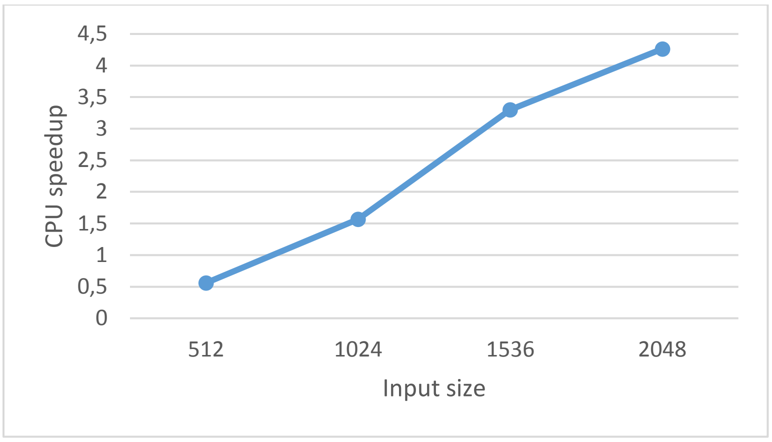


Figure 34: The CPU kernel speedup as a function of the input size with a nonzero density of 5% and a 16x16 workgroup.

	512	1024	1536	2048
CPU time	0,03534s	0,0479 s	0,080095 s	0,011052 s
Time increase		1,35 times	1,67 times	1,37 times
GPU time	0,01976 s	0,07497 s	0,264816 s	0,47120 s
Time increase		3,79 times	3,53 times	1,77

Table 9: The CPU and GPU kernel execution times and the time increase between the different densities.

Figure 34 and table 9 shows that as expected with smaller sizes the GPU is faster and on the bigger sizes the CPU is faster. The relation is discussed in the next chapter.

Both these graphs and tables were without the data transfer time on the GPU, with the transfer time the GPU basically does not stand a chance. This is shown in figure 35, 36 and table 10, 11.

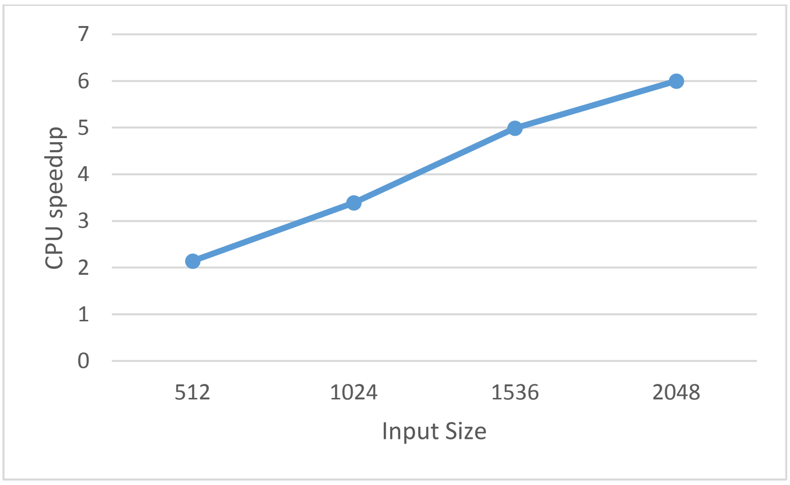


Figure 35: The CPU kernel speedup as a function of the input size with a nonzero density of 5% and a 16x16 workgroup with the data transfer time.

	512	1024	1536	2048
GPU time	0,01976 s	0,07497 s	0,264816 s	0,47120 s
Transfer time	0,05593 s	0,08724 s	0,13478 s	0,19208 s
Transfer/ GPU	283%	116 %	50%	40%

Table 10: The GPU execution and transfer time with a nonzero density of 5% and a 16x16 workgroup.

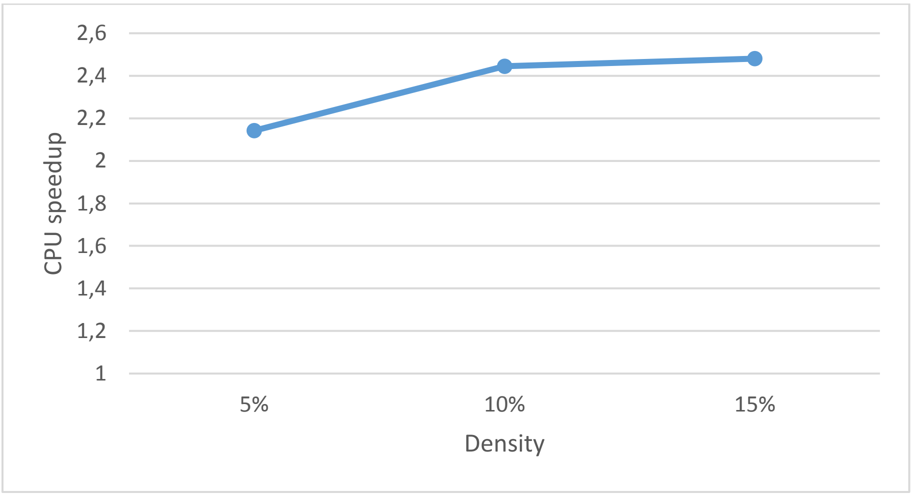


Figure 36: The CPU kernel speedup as a function of the nonzero density with an input size of 512 and a 16x16 workgroup with the data transfer time.

	5%	10%	15%
GPU time	0,01976 s	0,03398 s	0,04728 s
Transfer time	0,05593 s	0,06644 s	0,07049 s
Transfer/ GPU	283%	195%	149%

Table 11: The GPU execution and transfer time with an input size of 512 and a 16x16 workgroup.

As you can see in figure 35 and 36 the transfer time had a much bigger impact on SpMV than on Image Convolution on the smaller sizes. The bigger the size the longer it will take for the GPU to process the data, meaning that the transfer times matters less. The relation between the execution time and transfer time follows the same downward trend when you look at the densities, but it is slower since it is a smaller total work increase for the GPU.

Finally, in figure 37 the workgroup size tests are shown.

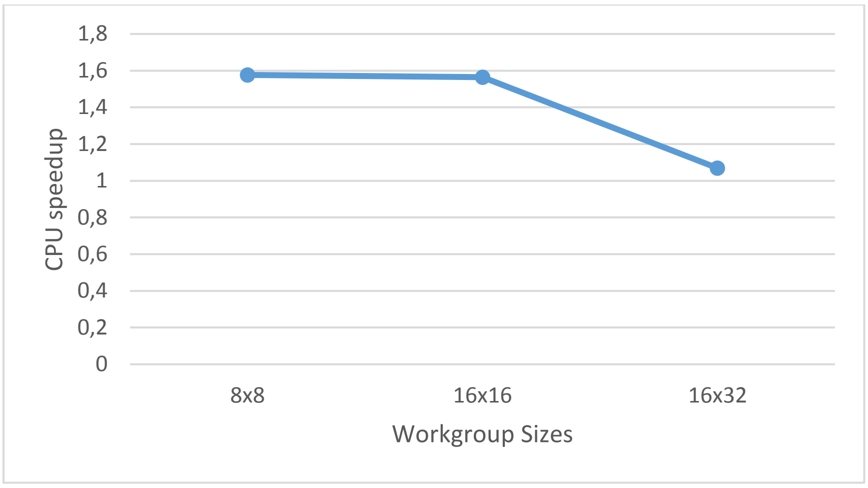


Figure 37: The CPU kernel speedup as a function of the workgroup size with a nonzero density of 5% and an input size of 1024.

	8x8	16x16	16x32
CPU time	0,04799s	0,04790 s	0,07176 s
Time increase		1,002 times	1,49 times
GPU time	0,07536 s	0,07497 s	0,07673 s
Time increase		0,99 times	1,02 times

Table 12: The CPU and GPU kernel execution times and the time increase between the different workgroup sizes.

This result is quite different from the workgroup tests on the two convolution implementations. 8x8 and 16x16 stays more or less the same, however as you can see in table 12 at 16x32 the CPU execution time increases by 50%. This will be discussed in the next chapter.

4.3 Results summary

The following table (table 13) summarises which configurations ran faster on which architecture.

Configurations running faster on the GPU	**Configurations running faster on the CPU**
A Söderholm and J Sörman Convolution	**A Söderholm and J Sörman Convolution**
All input sizes with and without data transfer when a 3x3 Filter was used.	All the configurations of the original code
Using a 3x3 or 5x5 filter with and without data transfer.	Using a 9x9 filter with and without data transfer.
Using the Half_float and Float data types with a 3x3 filter.	
All workgroup sizes with a 3x3 filter.	
AMD Convolution	**AMD Convolution**
	All input sizes, filter sizes and workgroup sizes.
SpMV	**SpMV**
All densities on the input size 512 without data transfer.	5% and larger densities on the input sizes 1024, 1536 and 2048 without data transfer.
	5% and larger densities and all input sizes with data transfer.
	All workgroup sizes on the input size 1024 with a 5% density.

Table 13: Results summary

5. Discussion

5.1 Convolution results

Here the most interesting results presented in 4.1 Convolution is explained and discussed.

5.1.1 4k -> 8k difference

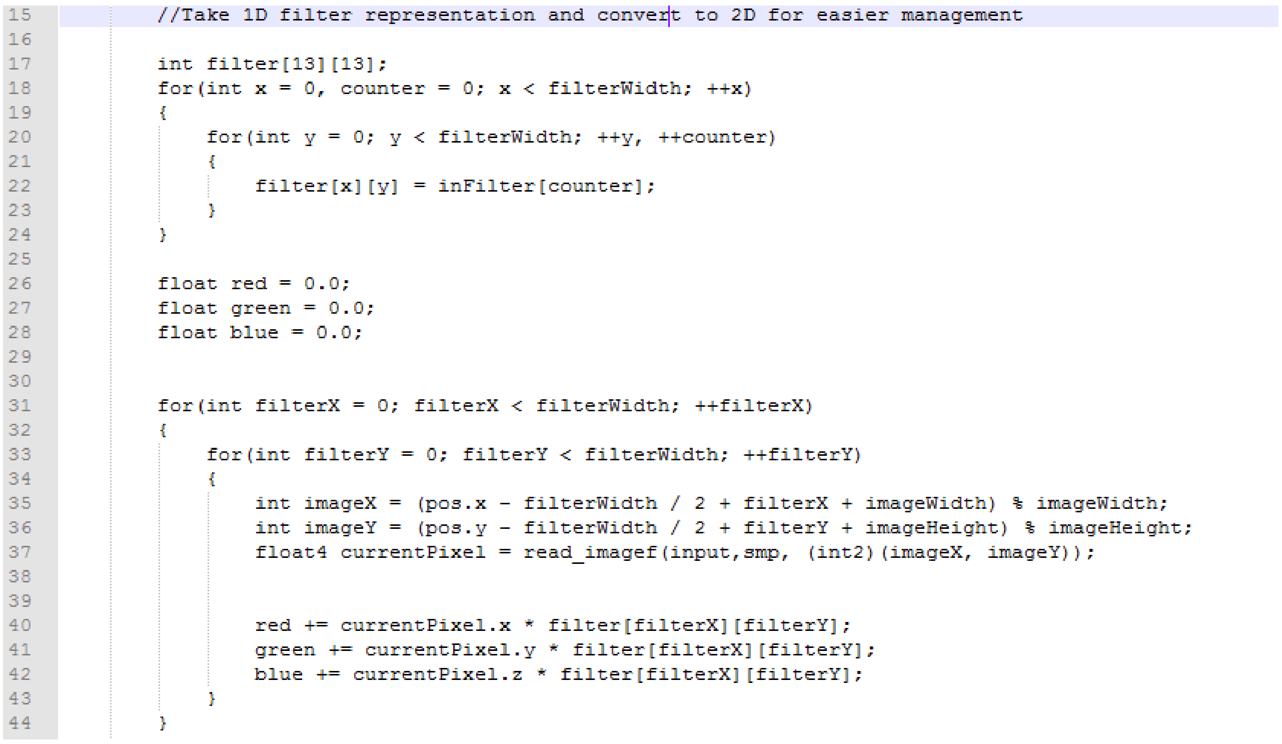
As shown and explained in section "4.1.1 Previous Convolution Results" the big performance difference between a 4k image and an 8k image, is caused by the fact that at 8k the GPU runs 4 sequential kernels instead of just one. But why does it run 4 kernels and how does it behave on the CPU?

It runs four sequential kernels because of the GPUs memory limitations. A 4k image the GPU can manage to process in parallel however as A Söderholm and J Sörman discovered, the 8k image is too big for the GPU. This meant they had to split the picture in to four parts, giving one part to each kernel. Resulting in four kernels, theses four kernels will be processed in sequential order on the GPU. However, the CPUs memory is big enough for the 8k image which means that it can process the four kernels in parallel. Since one kernel on its own runs faster on the CPU, it is logical that four kernels running in parallel on the CPU will be a lot faster than four running sequentially on the GPU. Hence the dramatic speedup difference for 8k images.

5.1.2 GPU EU stalling

As shown in figure 12 when running the original code on the GPU, the GPU execution units were stalled just under 80% of the execution time. This was partially solved by using the correct filter sizes, but why did the filter size cause stalling? Moreover, even after the filter sizes were modified, 3x3 filters had about 40% stalling, 5x5 70% and 9x9 still had 80%. What causes this stalling and why did the fix work so much better for 3x3 than on any of the other filter sizes?

First what cause the original stalling, let’s have a look at the code again.



```
15          //Take 1D filter representation and convert to 2D for easier management
16
17          int filter[13][13];
18          for(int x = 0, counter = 0; x < filterWidth; ++x)
19          {
20              for(int y = 0; y < filterWidth; ++y, ++counter)
21              {
22                  filter[x][y] = inFilter[counter];
23              }
24          }
25
26          float red = 0.0;
27          float green = 0.0;
28          float blue = 0.0;
29
30
31          for(int filterX = 0; filterX < filterWidth; ++filterX)
32          {
33              for(int filterY = 0; filterY < filterWidth; ++filterY)
34              {
35                  int imageX = (pos.x - filterWidth / 2 + filterX + imageWidth) % imageWidth;
36                  int imageY = (pos.y - filterWidth / 2 + filterY + imageHeight) % imageHeight;
37                  float4 currentPixel = read_imagef(input,smp, (int2)(imageX, imageY));
38
39
40                  red += currentPixel.x * filter[filterX][filterY];
41                  green += currentPixel.y * filter[filterX][filterY];
42                  blue += currentPixel.z * filter[filterX][filterY];
43              }
44          }
```

Figure 38: Array creation and convolution computation.

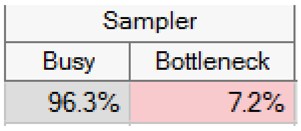
The first assumption was that the main loops on line 31 and 33 in figure 38 used the size of the newly created 2D array *filter* as the end value. Thereby always using the same filter size which would explain why the stalling percentage did not change. However, they in fact use the variable *filterWidth*, which is the actual filter width i.e. 3, 5 or 9, this is supplied from the host code and sent to the kernel. This meant that the filter sizes did change which implicates that this was not the problem.

The next theory was that it was a memory problem because a 13x13 int array takes up a lot of memory.

As stated in section “2.4.1 Intel HD graphics 4600 GPU”, the GPUs Execution units have 28 kilo bytes of local storage. One int is 4 bytes long and in a 13x13 array there are 169 ints equalling 676 bytes and the other structures in the kernel takes up 60 bytes which equals 767 bytes per work-item. Furthermore each EU has 7 hardware threads which in turn runs 32 work-items in parallel which means, at full capacity there are 224 work items on one EU and 224 * 767 = 171,808 kilo bytes which is a lot more than 28 kilo bytes. This means that the rest of the data instead is put in the GPU cache, which all of the EUs will have to access more or less at the same time. This results in a very big percentage of stalls.

If you use the correct filter sizes, work-items with a 3x3 filter will take up 21 kilo byte of storage, 5x5 35 kilo byte and 9x9 86 kilobyte. This is why the fix worked so much better for 3x3 because then only the EUs local storage is needed, while the other sizes need to use the GPU cache.

However, even though 3x3 does not use too much memory it still has 40% stalls. This is caused by line 37 in the code above. That is when the kernel accesses the image stored in the GPUs sampler. Since you access the neighbouring pixels when calculating the filter the kernels handling neighbouring pixels will use the same pixels (i.e. access the same memory addresses). Furthermore, as you can see in the code they do not use any special data access pattern instead they just go from left to right when calculating the filter for each pixel. The amount of stalling is worsened because when accessing the image they use the address clamping mode built in to OpenCL. Address clamping means that if a pixel outside the boundary is selected instead the nearest pixel inside the image is selected. This happens on all the boundary pixels and results in even more shared memory locations between the threads, which in turn equals more stalling. This assumption is supported by the GPU hotspot analysis (figure 39) which shows that the sampler is bottlenecked when a 3x3 filter is used. When the sampler is bottlenecked it has more requests than it can handle, i.e. its input queue is full. Or rather, it cannot handle the requests fast enough since it stalls on accessing the same memory locations at the same time.



Sampler	
Busy	Bottleneck
96.3%	7.2%

Figure 39: The percentage of execution time that the sampler is bottlenecked on a 1440p image with a 3x3 filter.

5.1.3 Image and filter size relation to execution time

As stated by B. R. Payne et al. [6] the GPU speedup should be logarithmically dependent on the image size. However their tests are performed on a much more powerful, dedicated GPU. Therefore it is not surprising that we do not get the exact same result.

As described in the results it in fact is only beneficial for the GPU to have a bigger image size up to 720p from there on out it only gets better for the CPU. Since at 720p it is already fully utilized. The decrease is cause by two implementation details favouring the CPU.

1. The GPU still stalls quite heavily.
2. It is a big convolution kernel, in which they except perform the main computation, does some not entirely necessary work (create an entirely new array and clamps the RGB values).

The biggest part is that the GPU still stalls 40% of the execution time while the CPU hardly stalls anything at all (on the big sizes 4k-8k it sometimes stall around 10%). Secondly the extra computations is making it more computationally bound which favours the CPU's fast cores. These two characteristics coupled with the fact that the GPU is not very fast nullifies the performance gained by the GPU on bigger sizes.

In order to examine the specific relation of the two parameters and the execution time regression analysis was performed over the input data. The hypotheses are the following:

1. Speedup should be logarithmically dependent on the input size.
2. Speedup should be linearly dependent on the filter size.

Input size relation

The regression analysis showed the following.

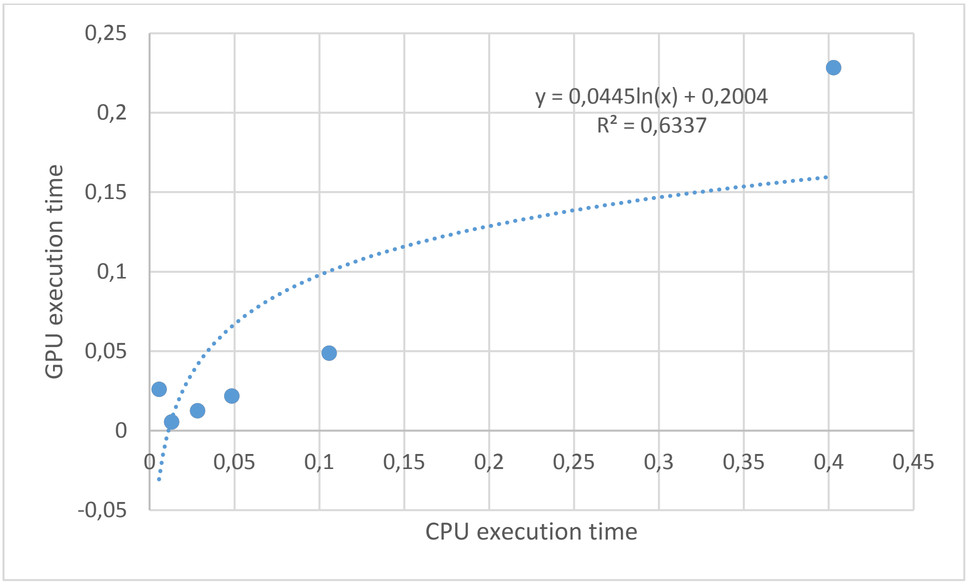


Figure 40: Regression analysis over the input sizes and execution time with a logarithmic trend line.

Figure 40 shows whether the relation between the CPU execution speed and the GPU execution speed over a number of input sizes can be fitted to a logarithmic trend line. If it can it means that the relation is logarithmic. As you can see the data points does not fit that well to the trend line. Moreover the R^2 value which estimates how well the data fits to the trend line (going from 0-100%) has a value of 63%. Since this was quite a low value I also calculated Pearsons correlation coefficient which produces a value between -1 and 1. This value indicates whether there is a negative correlation (-1), no correlation (0) or a positive correlation (1) between the two data sets (GPU time and CPU time). The correlation coefficient over the image sizes are 0.99 which means that there is a strong positive correlation. Therefore I can confidently say that the speedup is dependent on image size however it does not seem to be logarithmically dependent because of the low R^2 value.

Filter size relation

The regression analysis showed the following:

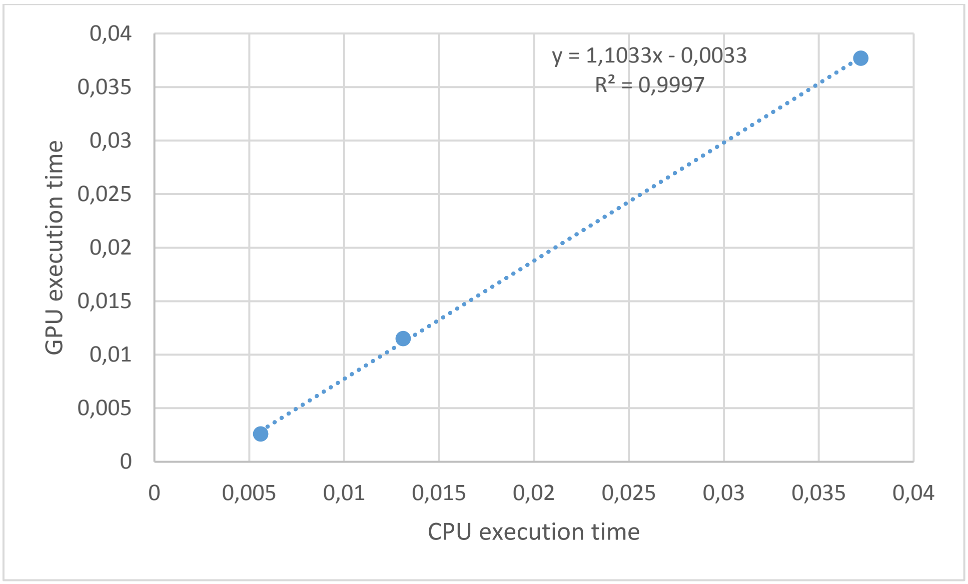
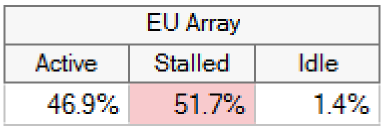


Figure 41: Regression analysis over filter sizes and execution time with a linear trend line.

As you can see in figure 41 the data fits the trend line very well, the R^2 value is at 99% and the correlation coefficient is 0.99. Therefore I can say that the hypothesis was accurate and the speedup is linearly dependent on the filter size following the function y = 1.1033x – 0.033.

5.1.4 Why the optimized version causes more stalling

As showed in the results the optimized version causes more stalls. What they do in the optimized kernel is transferring the image from the global memory to the faster shared memory. Each workgroup shares one two dimensional array with the size 32x32 = 1024 containing 1024 pixels of the image. The work-groups have the size 16x16=256 and each kernel transfers 4 pixels to the array, thereby filling up the entire array 256*4=1024. After that all the actual filter operations can be done from the faster shared local memory instead of from the slower global. This lowers the amount of global memory accesses. However, the writing from global memory to local memory causes the sampler to be bottlenecked once again, not as much but it equals some extra stalling (figure 42). When writing from the global memory to the local memory they overwrite some of the local memory positions which also equals some stalls. Those two together causes about 6% of execution time spent in stalls, the rest of the stalls is caused by their data access pattern which is worse for local memory than global.



EU Array		
Active	Stalled	Idle
46.9%	51.7%	1.4%

Figure 42: Percentage of execution time which the GPUs EUs were stalled with a 3x3 filter and a 1080p image without writing to the local memory from the sampler.

It is worse for local memory because of the way they handle each workgroups edge pixels, if they want a pixel outside of the local 32x32 array e.g. -1,-1 they instead take 31,31. They simply use the corresponding pixel on the other side of the array. The main difference from the basic implementation is that those pixels would use their actual neighbour pixels. Those neighbouring pixels would be handled by work-items in another workgroup which it is possible, does not run at the same time. The work-items in the optimized version on the other hand only uses pixels from its own workgroup, and all work-items in the same workgroup runs concurrently. Basically there are more stalls since work-items in the same work-group share more memory locations. And there is a higher chance that the work-items in the same workgroup will run in parallel than the work-items in different work-groups. Furthermore, because there are more stalls the performance gain from using the faster local memory is nullified. Instead the optimized version loses time on the extra writing from global memory to local memory.

5.1.5 AMD example results

The biggest difference between A Söderholm and J Sörman's code and the AMD example is that the AMD example always runs faster on the CPU. This is because it is written for CPU and the setup and data access is written in a way to favour the CPU. For example within the kernel they access the data row wise which favours the CPU while the GPU wants to access it column wise. They also create too many threads sometimes as shown in the results which causes extra overhead.

Regarding the different parameters however, the AMD example more or less confirms the hypotheses which are the following:

1. The bigger the input the better it is for the GPU.
2. The smaller the filter the better it is for the GPU.
3. As long as the workgroup size is not bigger than 256, not smaller than 24 and is devisable by 8 the speedup should not vary with more than 0.05 units.

5.1.6 Parameter impact

The two parameters that had the biggest impact on the execution times of the two architectures was the input size and the filter size. In order to compare these two parameters accurately I compared the number of operations that the image size and filter sizes added. Since increasing the image size and the filter size means adding on extra operations in two different ways to the computation. If you convert the different parameter tests to the number of operations you see that a good comparisons point is between a 480p image with a 9x9 filter and a 1440p image with a 3x3 filter since both of them converts into 33 million operations. If you look at the execution times of these two tests in table 1, 2, 5 and 6 you will see that the larger image time takes a longer time to execute in both the previous thesis convolution as well as in the AMD example. This means that the image size actually has a larger impact on the execution time of both architectures.

Moreover you can also see on the AMD example that the larger image sizes impacts the CPU more than the GPU. And that the filter size impacts the GPU more than the CPU.

The data type did not impact the architectures that much individually since the CPU lost some time on the smaller data type while the GPU gained some. In relation to each other the difference was on par with the impact of the input size.

The workgroup size did hardly impact the execution time at all. Even though 16x32 is four times larger than Intel recommends.

5.2 SpMV results

The most surprising part of the SpMV results were that the GPU performed as bad as it did compared to the CPU since S. Kim, I. Roy and V. Talwar [4] got much better results. This was because of the usage of the CSR format and the fact that each kernel on the bigger sizes gets a large chunk of the problem. This means that when using CSR on a GPU you would either want a matrix with short rows and long columns or one with a very small density. This would ensure each kernel only got a handful of numbers each instead of the 25-300 numbers that they got in our tests.

In order to confirm this, a test was performed with a 2048x2048 matrix with a 0.5% density meaning that each work-item got about 10 nonzero numbers each.

	CPU	GPU
Execution time	0,04845 s	0,03404 s
Speed up	0,70 times	1,42 times

Table 14: The CPU and GPU SpMV execution times without data transfer on a 2048 input and 0.5% density. And the speed up on the GPU compared to the CPU and vice versa.

As you can see in table 14, with a low enough density the GPU outperforms the CPU.

However if the matrix has a high density a implementation better suited for a GPU would be for example to split each row between a number of threads/work-items. In order to ensure that each work-item does not get to many operations, thereby utilizing the parallelism better. This kind of implementation is used in the SHOC benchmark suit which S. Kim, I. Roy and V. Talwar [4] used. Consequently as stated in section “2.9 Related works” their results were that the GPU outperformed the CPU when running SpMV with a 16 megabyte input. In comparison, our largest test 2048 with 15% density was 8 megabyte and ran faster on the CPU. However they do not specify how high density of nonzero numbers they use.

5.2.1 Transfer times

That the transfer times had such a big impact is not that surprising because instead of just sending one big buffer with all the data as in Image Convolution, in SpMV you need to transfer each of the arrays used to compute the results. This equals significantly more overtime since you need to start and stop the individual data transfers of each array. This means that this type of algorithm would be ideal for using the CPU’s memory instead of sending it to the GPU. As mentioned before, this could equal a bit slower kernel execution, however it would probably not be as big of a time sink as the data transfer currently is. This is supported by the fact that, as mentioned in section “3.2.5 Data transfer”, when we tested to transfer the data in the AMD example there was not any significant speedup. There was however a significant speedup when the data was transferred in A Söderholm and J Sörman’s code. The reason for this had probably to do with their kernels heavy use of the sampler memory which is located on the GPU.

5.2.2 Workgroup sizes

The workgroup sizes had practically no impact at all on the GPU, however on the CPU the 16x32 size slows down the execution time by 1.5 times. The fact that it was slowed down at 16x32 is not that weird since it is four times as big as Intel recommends. However why it impacted the CPU and not the GPU is another question. The most likely reason is that it in this case affects the load balancing more on the CPU. Each CPU hardware thread handles one workgroup each, the Intel I7-4790 has 8 hardware threads. At a size of 64 there are 32 workgroups, at 256 there are 8 workgroups, both of which utilizes the entire CPU. While at 512 there are only four workgroups utilizing only half of the CPU. Since the CPU is fully utilized at both 32 and 8 workgroups we do not see any specific difference scaling down from 32 to 8. However why we do not see a change on the GPU from 8x8 (32 workgroups) to 16x16 (8 workgroups) is a bit strange. Since at 32 workgroups the GPU is fully utilized while at 8 it is not. It could be because the kernels handle too big parts of the problem nullifying the gain and loss of more/less parallelization.

5.2.3 Density and input size relation

My hypotheses regarding how the density and input size related to the speedup were the following: The speedup should be linearly dependent on the input size and the speedup should be logarithmically dependent on the density.

Ones again regression analysis was performed in order to examine these relations.

Input size relation.

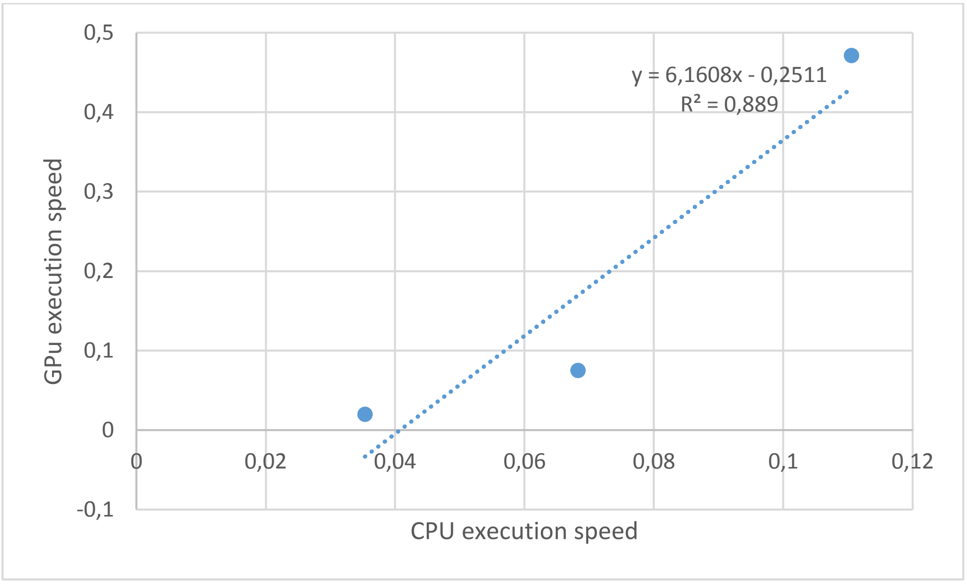


Figure 43: Regression analysis over the input sizes and execution time with a linear trend line.

In figure 43 the linear trend line fits quite well and the correlation coefficient is 0.94. Therefore I can say that the speedup is linearly dependent on the input size following the function y = 6.1608x – 0.2511.

Density size relation.

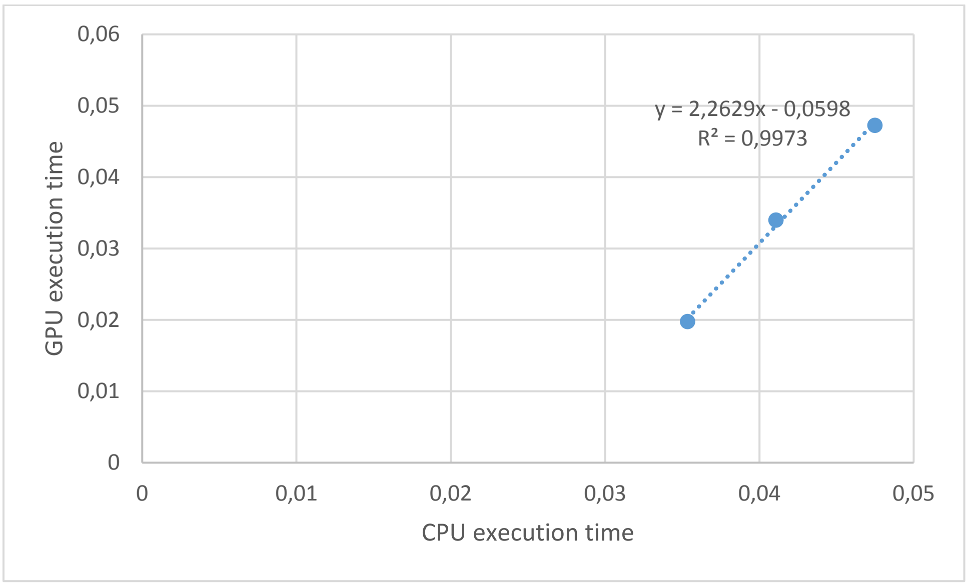


Figure 44: Regression analysis over the density sizes and execution time with a linear trend line.

In figure 44 you can see the linear trend line fitting more or less perfectly to the data, the R^2 value is at 99% and the correlation coefficient is 0,99. Therefore I can confidently say that the hypothesis is accurate and the speedup depends linearly on the density described by y = 2.26 *x – 0.0598.

5.2.4 Parameter Impact

At first glance my tables seem to show that the input size has a much bigger negative impact on the GPU than the density. This is true since a doubling of the input size means that the number of operations are quadrupled while a doubling of the density means that the number of operations are doubled. However if you want a more equal comparison you would have to compare a large size with a small density to a smaller size with a high density, in order to ensure that the number of operations is the same.

Therefore a complementary test was performed. The test was between an input of 4096 with 2.5% density and 2048 with 10 % density. This means that both configurations have the same number of operations to perform. The results of the complementary test are shown in table 15.

	2048, 10%	4096, 2,5%
Execution time	1,18 s	0,90 s

Table 15: Execution time of kernel on GPU with input size 2048 and density 10% as well as input size 4096 and density 2.5%.

And as you can see in table 15, the bigger input size with a smaller density outperforms the smaller input size with a higher density. This is because the higher density adds extra operations to each work-item while a larger input spreads out the extra operations over more work-items. This makes perfect sense since the GPU is built for many small work-items.

The CPU on the other hand is impacted more or less the same amount by the density and the input size. This also makes sense because both of them equals more sequential operations. As explained in the previous section the Workgroup size did not have any impact on the GPU but on the CPU 16x32 did have a larger impact than both the input size and the density. Since 16x32 is too large for the CPU it is not that weird either, as long as Intel's recommended sizes are used it did not have any real impact.

5.3 General Results

How does these results compare to other studies and what do they actually mean?

As showed in the results, Image Convolution executes faster on the GPU for small enough filters i.e. when it is bandwidth bounded. For bigger filters when the algorithm is computationally bounded, the CPU executes it faster. This aligns well with both the results of B. R. Payne et al. [6] and E. Ching et al. [8]. B. R. Payne et al. [6] show that their dedicated GPU can outperform the CPU on the smaller filter sizes and E. Ching et al. [8] that bandwidth bounded applications are favoured by a GPU.

Moreover V. W. Lee et al. [3] state that on the dedicated GTX 280 GPU (which is faster than the Intel HD 4600 GP) they achieved a 2.8 times speedup (data transfer omitted) when running Image Convolution, compared to running it on the i7-960 CPU. We achieved a 2.35 times speedup at best, with a suboptimal implementation. This coupled with the fact that our integrated GPU's data transfer most likely is faster, running Image Convolution on an integrated GPU could actually be preferable. As long as the entire image fits in the GPU memory.

The SpMV results differed from the related studies as mentioned in section "5.2 SPMV results" which is caused by OpenDwarf's usage of the CSR format in SpMV. And as stated by V. W. Lee et al. [3]:

Algorithms where each thread computes a large set of the problem benefits from architectures which have a higher single threaded performance.

The CPU has a much higher single threaded performance, which is why this standard implementation of SpMV is better suited for CPUs, unless the density is small enough.
When the density was low enough we could see a speedup on the GPU of 1.5-2x which is not so far behind the 2-2.5x speedup presented by V. W. Lee et al. [3] for SpMV.
The main difference being that the dedicated GPU get those kind of results in general while our result was a best case scenario. This is most likely since V. W. Lee et al. [3] used optimized versions for their architectures while we used a standard implementation.

C. Gregg and K. Hazlewood [18] tests the data transfer between a CPU and four different dedicated NVIDIA GPUs: the Tesla C2050, the GTX480, the 9800 GT and the 330 M.

Their result for Image Convolution on a problem size of 1536x1536 which is a bit larger than our 1920x1080p image is shown in table 16.

	Tesla C2050	GTX 480	9800 GT	330 M
Transfer time	10 ms	16 ms	15 ms	17 ms

Table 16: C. Gregg and K, Hazlewoods [18] GPU-CPU transfer times on a 1536x1536 image.

Compared to our result on a 1920x1080 image where the transfer time was 6 milliseconds it is obvious that the transfer times are much faster on our integrated GPU. However, the execution times are faster on all of the dedicated GPUs they test, but unfortunately they did not specify the filter size. Therefore we cannot have a fair comparison between the execution times.

They did test SpMV too, however they used another format than CSR and a bigger input, therefore we cannot compare our transfer results to theirs. This is because the amount of data sent is format dependent.

5.4 Method

The method consisted of three parts, benchmark application selection, benchmark application implementation and analysis. In this section we will discuss them one by one starting with the benchmark application selection.

Benchmark application selection

The biggest problem during benchmark application selection was the availability of relevant benchmarks. For example the results would have a higher validity if I would have been able to find a benchmark suit used in related studies which included an Image Convolution application. Instead the AMD example was used, while the AMD example in itself is not badly coded it has however not been proven by researchers to be suitable for benchmarking. Moreover since they state that it is written for CPUs it could be that there are some optimizations for a CPU which I simply missed. It simply is not that well suited for benchmarking.

This problem was somewhat negated when SpMV was chosen because checking whether the application was included in benchmarking suites was a part of the selection process.

However as mentioned before the specific SpMV implementation used differs from the implementations used in the related studies. This caused somewhat unexpected results which made the analysis of SpMV take an unnecessary amount of time. Moreover, it makes it difficult to compare our results to other studies results. This could have been avoided if the OpenDwarf's implementation were compared to the implementation used earlier in the project.

When choosing a benchmark application I searched for standard implementations which were not optimized for either architecture. However it would probably have produce more relevant results if I instead used optimized benchmarks for both the CPU and the GPU. Because in the real world the applications will most likely be optimized for the relevant architecture. On the other hand if the benchmarks were equally optimized the relation should stay more or less the same.

Benchmark application implementation

The majority of OpenCL benchmark suits are written for Linux while this thesis has been performed on a windows computer. Because of this the original plan was to perform the test in a virtual Linux environment. I spent about a week to set up Intel's OpenCL distribution on a Linux virtual machine only to realize that the GPU cannot be accessed through a virtual machine. If I had checked that before a lot of time could have been saved.

Instead, as described in section "3.2.1 Implementation procedure" the Linux benchmark applications were converted from Linux to Windows. Since the main part of the time measuring and analysis have been focused on the kernel which did not need any changes, the modifications should not hurt the validity in a meaningful way. However since I have not been able to test the original version, or found results on the same hardware, I cannot be 100% sure that my modifications did not have any impact on the performance.

The only way to ensure reliability is to use my code since as mentioned I cannot 100% guarantee that the modifications did not impact the performance. Moreover if the same tests were to be performed on a Linux machine with the original OpenDwarf's code and the results are different from mine. I cannot say if that has to do with my modifications or the change of operating system.

In order to ensure replicability you need to use the same hardware, software and the same code. The hardware and software are easy to replicate since they are detailed in the report and the code will be available on github, therefore replicability should not be a problem.

Analysis

There was two problems during the analysis, the number of times to run the tests and the chosen input sizes.

On the GPU the test times were in general very stable, the individual runs differentiated normally between 0.1-03 milliseconds, that is why almost all the GPU test were run only 5 times. This was also true for the CPU when running the previous Image Convolution. However the CPU could vary noticeably between executions up to 10-20 milliseconds running SpMV. That is why that one was run 10 times instead of 5, the results would have a higher validity if more tests were run. However one of the tests were run 50 times and the mean was approximately the same as the mean of 10 runs. Therefore I deemed 10 runs to be enough. The reason for running the AMD example 100 times per test was that it does multiple kernel runs and calculates the mean execution time automatically. The other ones I had to run manually and calculate the mean times in excel.

It would have been better to set up a similar system as in the AMD example in the SpMV code, this would have given a higher validity to the tests. The reason it was not done was because I deemed that the test results validity was high enough and that it was more valuable to spend the time on the analysis instead. The validity was deemed to be high enough since when calculating the mean from the specified number of runs over multiple sets of runs, the calculated means were within 0.1 milliseconds from each other.

The reason that the inputs 512, 1024, 1536 and 2048 was chosen and not the much bigger sizes which the related studies used was that the GPU's cache was not big enough. Instead the sizes were meant to be of about the same magnitude as the convolution input sizes and be able to show a linear relation, which they are. However the amount of work-items are significantly lower since each work-item gets one entire row. Since the amount of work-items are low it means that the thread creation overhead will matter more and could possibly skew the results somewhat. Note also that bigger sizes would not produce entirely different results. Since the amount of operations per work-item also increases as the input increase the CPU GPU relation stays basically the same. However with bigger sizes you can see more clearly what happens since then the thread overhead does not matter as much.

Sources

When it comes to the sources used in this thesis, depending on what they are used for I have had different standards for them.

All the related studies I wanted to be published and peer reviewed sources since they are the ones that the thesis is based on and compared to. The sources detailing performance measuring I have not had as high standards for since some of them I have had to use e.g. the ones about OpenDwarf's. Even though they are not the most well cited and there could be some bias (since the authors are the developers) they are the only published sources I found.

The sources about the hardware and parallel computing has had the lowest standards since much of it is more or less established facts. E.g. what it means that an algorithm is data parallel and how a SIMD architecture executes. Moreover, in the case of the hardware, Intel themselves does have the most in-depth explanation of their own hardware even though one can expect it to be somewhat biased. Therefore I have tried to only to cite the specific architectural specifications from the Intel sources.

5.5 The work in a wider context

This thesis shows that a cheaper GPU with less power consumption can be used to speed up Image Convolution and SpMV almost as much as more powerful, dedicated GPUs. This information enables both companies and individuals to choose the cheaper alternative if they want to use it for applications of this sort. While at the same time having a smaller budget and retaining a comparable speed to dedicated GPUs. The Intel HD 4600 is not always as fast as dedicated GPUs but it does consume less power. This makes it a prime candidate for companies or individuals whose main focus is to lower their power consumption. Thereby lowering their electricity costs and favouring the environment. Note however that this thesis has handled GPU performance, to optimize for power consumption is an entirely different matter. While we in this work always have striven to have as many of the GPU cores active at the same time as possible. When programing for a low power consumption the number of cores is much more delicate. Since the more cores that are active the more power is consumed and some applications does not gain any performance by using all cores. In those cases you want to find the optimal number of cores to use in order to maximise the speed at the same time as minimizing the power consumption [34].

This thesis presents results from different versions of the same Image Convolution application one of which is considerably slower than the other. Moreover, as stated earlier, even the faster of the two versions still has performance problems, specifically on bigger filters. This enables misinterpretation of the results, both from a careless reader who only looks at the graph and a reader with vested interests. A reader with vested interests can use the graphs to showcase worse performance on the hardware without specifying that it is caused by the implementation. This could be somewhat alleviated if figure 9, 10, 11 and 12 were not shown which shows the original implementations results. However, the graphs could still be taken out of context since the implementation is suboptimal and the results would lose some of its validity if not the earlier results were shown. Moreover the validity of the conclusions of why the previous code run worse would be significantly hampered.

6. Conclusions

The purpose of this thesis was to provide more information about GPGPU computing in OpenCL on the Intel HD 4600 GPU. Moreover what caused the unexpected results of the previous thesis? This should be achieved by examining the code from the thesis “GPU-acceleration of image rendering and sorting algorithms with the OpenCL framework” as well as testing a SpMV benchmark application and a standard Convolution implementation.

In regards to this several general lessons can be drawn from this work which are valuable to consider when writing OpenCL applications for an integrated GPU.

1. When choosing an integrated GPU be aware of its memory limitations, if the problem size is too big it will hamper the performance significantly. Since the GPU will have to run multiple kernels sequentially as in the case of the 8k images.
2. When writing a kernel you need to be aware of how much private memory the EUs has access to since if you use to much you will instead store private data without any reason in the much slower cache. Moreover it can also cause stalling as it did in A Söderholm and J Sörman’s code.
3. When writing applications such as Convolution where the same data is used by several EUs, data access patterns can have a significant impact on the performance. This is even more important when the shared memory is used. Since more data will be shared between work-items which run in parallel.
4. Lastly and one of the most important things when constructing a kernel. Do the least amount of work in each one as possible when optimizing for the GPU, in order to gain as much as possible from the massive parallel architecture.

These lessons are the reasons why the previous convolution code performed so badly on the GPU. I.e. the kernels used to much private memory, the data access pattern was not optimized and the kernel performed unnecessary extra operations not needed for the computation. Moreover when timing their application they did not use the intended OpenCL timers instead they used a built in C++ timer which did not give the correct execution times.

Furthermore the research questions were the following:

- Which are the most important factors determining the execution time of Image Convolution and SpMV on the Intel HD graphics 4600 GPU compared to the factors impacting the execution time on the Intel i7-4790 CPU?
- Do you observe any speedup when running Image Convolution or SpMV on the Intel HD graphics 4600 GPU compared to when running them on the Intel i7-4790 CPU, furthermore what causes this result?

The answer to the first question is that the image size had the biggest impact on the execution time of Image Convolution on both of the architectures. For SpMV the density of nonzero numbers had the biggest impact on the execution time on the GPU. While on the CPU both the density and input size impacted the execution time the most.

Note however that if the workgroup size was especially poorly chosen so that not the entire architecture was utilized this could impact the performance more than the other parameters.

However as long as you stay within Intel's recommendations it does not seem to have that much of an impact at all.

The answer to the second question is, yes for both applications a speedup was observed when running them on the GPU. A Söderholm and J Sörman's [2] code ran consistently faster on the GPU as long as the filter was not bigger than 3x3. This is simply because Image Convolution is a very suitable problem for data parallelization. The SpMV implementation did run faster on the CPU as long as the individual work-items did not get to big part of the problem each. I.e. either on small sizes or on big sizes with a very small density since the CSR format (which the implementation used) allocates one entire row to each work-item. This meant that most of the time SpMV ran faster on the CPU. With a more suitable format where the work-items never gets allocated to big a part of the problem it would run faster on the GPU most of the time instead of only for ideal input and density sizes.

A more general answer to this question would be: yes this GPU can be used to successfully GPU accelerate applications. However which applications it can accelerate is very dependent on the specific implementation.

This thesis has contributed some guidelines to consider when programming for GPUs in OpenCL. Furthermore it has shown that the Intel HD 4600 can be successfully used to GPU accelerate applications within the GPGPU computing area. More specifically it showed that two applications when bandwidth bounded (Conv with small filters and SpMV) can be accelerated on the Intel HD 4600. While when computationally bounded (Conv with big filters) they ran faster on the CPU. However this is only two applications on one GPU, in order to get a general answer to which types of applications that are suitable to GPU accelerate on Integrated GPUs, you would need to test several more applications and GPUs.

Future works within the same area would be to test more applications on the Intel HD 4600, in order to achieve some sort of general classification over what kind of algorithms that are suitable for integrated GPUs. A good starting point could be to test the Berkley dwarfs on the GPU and the CPU. Then you could set up a classification for this specific GPU and draw conclusions about similarly powerful integrated GPUs.

7. Bibliography

[1] J. L. Manferdelli, N. K. Govindaraju and C. Crall, "Challenges and Opportunities in Many-Core Computing," *Proceedings of the IEEE,* vol. 96, no. 5, pp. 808-815, 2008.

[2] A. Söderholm and J. Sörman , "GPU-accelleration of image rendering and sorting algorithms with the OpenCL framework," Linköping University, Linköping, 2016.

[3] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal and P. Dubey, "Debunking the 100X GPU vs. CPU Myth:," in *international symposium on Computer architecture*, San Jose, 2010.

[4] S. Kim, I. Roy and V. Talwar, "Evaluating Integrated Graphics Processors for Data Center Workloads," in *Proceedings of the Workshop on Power-Aware Computing and Systems*, Farmington, 2013.

[5] S. Azmat, L. Wills and S. Wills, "Accelerating adaptive background modeling on low-power integrated GPUs," in *41st International Conference on Parallel Processing Workshops (ICPPW)*, 2012.

[6] B. R. Payne, S. O. Belkasim, S. G. Owen, M. C. Weeks and Y. Zhu, "Accelerated 2d image processing on GPUs," in *Computational Science – ICCS 5th International Conference, Proceedings, Part II*, Atlanta, 2005.

[7] S. Y. Kim, J. Bottleson, P. Bindu, S. C. Sakhare and J. S. Spisak, "Power Efficient MapReduce Workload Acceleration Using Integrated-GPU," in *IEEE First International Conference on Big Data Computing Service and Applications (BigDataService)*, 2015.

[8] E. Ching, N. Egi, M. Mortazavi, V. Cheung and G. Shi, "Unleashing the Hidden Power of Integrated-GPUs for Database Co-Processing," in *GI-Jahrestagung*, Stutgart, 2014.

[9] C. Kessler and J. Keller, "Models for Parallel Computing: Review and Perspectives," *PARS Mitteilung,* vol. 24, pp. 13-29, 2007.

[10] L. Howes, "The OpenCL Specification," Khronos OpenCL Working Group, 2015.

[11] R. Duncan, "A Survev of Parallel Computer Architectures," *Computer,* vol. 23, no. 2, pp. 5-16, 1990.

[12] T. Jain and T. Agrawal, "The Haswell Microarchitecture - 4th Generation," *International Journal of Computer Science and Information Technologies,* vol. 4, no. 3, pp. 477-480, 2013.

[13] N. Kurd,, T. P. Thomas, M. Neidengard, and R. Rajwar, "Haswell: A Family of IA 22 nm Processors," *IEEE JOURNAL OF SOLID-STATE CIRCUITS,* vol. 50, no. 1, pp. 49-58, 2015.

[14] http://ark.intel.com/products/80806/Intel-Core-i7-4790-Processor-8M-Cache-up-to-4_00-GHz, "Intel® Core™ i7-4790 Processor," 2017 January 26.

[15] http://www.intel.se/content/www/se/sv/architecture-and-technology/turbo-boost/turbo-boost-technology.html, "Intel® Turbo Boost Technology 2.0," 2017 January 26.

[16] J. D. Owens, M. Houston,, D. Luebke, S. Green, J. E. Stone and J. C. Phillips, "GPU Computing," *Proceedings of the IEEE,* vol. 96, no. 5, pp. 879-899, 2008.

[17] M. Garland and D. B. Kirk, "Understanding Throughput-Oriented Architectures," *communications of the acm,* vol. 53, no. 11, pp. 58-66, 2010.

[18] C. Gregg and K. Hazelwood, "Where is the data? Why you cannot debate CPU vs. GPU performance without the answer," in *International Symposium on Performance Analysis of Systems and Software*, Austin, 2011.

[19] M. Daga, A. M. Aji and W.-c. Feng, "On the Efficacy of a Fused CPU+GPU Processor (or APU) for parallel Computing," in *Symposium on Application Accelerators in High-Performance Computing (SAAHPC)*, Blacksburg, 2011.

[20] S. Junkins, "The Compute Architecture of intel Processor Graphics Gen 7.5," Intel, white paper, 2014.

[21] J. Shen, J. Fang, H. Sips and A. L. Varbanescu, "Performance Traps in OpenCL for CPUs," in *Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2013.

[22] G. E. Blelloch, "Programming Parallel Algorithms," *communications of the acm,* vol. 39, no. 3, pp. 85-97, 1996.

[23] T. R. W. Bräunl, S. Feyrer and M. Reinhardt, Parallel Image Processing, Springer Science & Business Media, 2013.

[24] K. Moreland and E. Angel, "The FFT on a GPU," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, San Diego, 2003.

[25] N. Bell and M. Garland, "Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, Portland, 2009.

[26] I. Reguly and M. Giles, "Efficient sparse matrix-vector multiplication on cache-based GPUs," in *Innovative Parallel Computing (InPar)*, 2012.

[27] T. Harada and L. Howes, "Introduction to GPU Radix Sort," http://www.heterogeneouscompute.org/wordpress/wp-content/uploads/2011/06/RadixSort.pdf, 2011.

[28] J. Gómez-Luna, J. M. González-Linares, J. I. Benavides and N. Guil, "An optimized approach to histogram computation on GPU," *Machine Vision and Applications,* vol. 24, no. 5, pp. 899-908, 2013.

[29] F. Alecu, "PERFORMANCE ANALYSIS OF PARALLEL ALGORITHMS," *Journal of applied quantative methods,* vol. 2, no. 1, pp. 129-134, 2007.

[30] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams and K. A. Yelick, "The Landscape of Parallel Computing Research: A View from Berkeley," EECS Department, University of California, Berkeley, California, 2006.

[31] K. Krommydas, W.-c. Feng, C. D. Antonopoulos and N. Bellas, "OpenDwarf's: Characterization of Dwarf-Based Benchmarks on Fixed and Reconfigurable Architectures," *Journal of Signal Processing Systems,* vol. 85, no. 3, pp. 373-392, 2016.

[32] B. A. Kitchenham, S. L. Pfleeger, D. C. Hoaglin, K. E. Emam and J. Rosenberg, "Preliminary Guidelines for Empirical Research," *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING,,* vol. 28, no. 8, pp. 721-733, 2002.

[33] S. William, L. Oliker, R. Vuduc, J. Shalf, K. Yelick and J. Demmel, "Optimization of Sparse Matrix-Vector Multiplication," *Parallel Computing,* vol. 35, no. 3, p. 178–194, 2009.

[34] S. Hong and H. Kim, "An Integrated GPU Power and Performance Model," in *Proceedings of the 37th annual international symposium on Computer architecture*, Saint-Malo, 2010.