Master thesis under supervision of H. Corporaal and C. Nugteren

Electronic Systems Group
Faculty of Electrical Engineering
Eindhoven University of Technology

Evaluating the Performance and Portability of OpenCL

Jarno van der Sanden
j.j.f.v.d.sanden@student.tue.nl

August 11, 2011

ABSTRACT

Recent developments in processor architecture have settled a shift from sequential processing to parallel processing. This shift was not based on a breakthrough in processor design, but was actually an alternative design trajectory to avoid the limits that were reached on single core development. Along with the shift towards parallel architectures, a gap arose between sequential programmers and parallel architectures. Several efforts from industry have tried to bridge the gap, resulting in parallel programming frameworks such as CUDA, OpenMP and the Cell SDK. A more recent parallel programming standard is OpenCL, as offered by the Khronos Group. OpenCL uniquely distinguishes itself by offering the programmer a single flexible programming framework, which can be used to target multiple platforms from different vendors.

To what extent OpenCL is a suitable substitute for current programming standards is the main topic of interest in this thesis. It includes a detailed comparison and analysis of the performances of several image-processing algorithms implemented in both CUDA and OpenCL, and mapped onto an NVIDIA GPU. Despite the similarity of OpenCL and CUDA, performance differences up to 16% are observed. Furthermore, the suitability of OpenCL as a single standard for targeting multiple platforms is investigated by mapping and optimizing the image-processing algorithms to other architectures, including an AMD GPU and an Intel multi-core CPU. Cross-platform OpenCL mappings show that functional portability as well as performance portability cannot always be guaranteed. A method is proposed to improve the performance portability by developing a single OpenCL implementation that ports to multiple target devices, reaching at least 80% of the performances of the optimal implementations on the target devices.

ACKNOWLEDGEMENTS

This thesis is the result of my graduation project, carried out as the final chapter of the Master Embedded Systems at the Eindhoven University of Technology. It gave me the opportunity to work in an academic environment, inspired and motivated by professors and PhD students around me. I would like to thank Henk Corporaal, my graduation supervisor, for giving me the opportunity to fulfill my project in the Electronic Systems Group. Special thank goes out to Cedric Nugteren, who has carefully guided me through the project and encouraged me in my research, by always giving me new bright ideas in our weekly discussions. Furthermore, I would like to thank the people participating in the fortnightly PARsE (Parallel Architecture Research Eindhoven) Architecture meetings, attending at my progress presentations and giving me valuable feedback on the project thus far. The Architecture meetings also gave me the exclusive opportunity to learn about current research performed by people within the Electronic Systems Group, inspiring me along the way. Finally, I would like to thank Rudolf Mak, for participating in my graduation commission and giving me feedback on my preparation project, which I carried out prior to the graduation project.

CONTENTS

1 INTRODUCTION 1

1.1 PROBLEM STATEMENT 1
1.2 OUTLINE 2

2 THE OPENCL PROGRAMMING FRAMEWORK 3

2.1 PLATFORM MODEL 3
2.2 EXECUTION MODEL 3
2.3 MEMORY MODEL 4
2.4 OPENCL PROGRAM STRUCTURE 5

3 ARCHITECTURES 7

3.1 NVIDIA FERMI GPU 7
3.2 AMD EVERGREEN GPU 8
3.3 INTEL NEHALEM CPU 9
3.4 OPENCL MAPPINGS 10
3.5 PERFORMANCE MODELS 11

4 METHODOLOGY 13

4.1 BENCHMARK ALGORITHMS 13
4.1.1 Binarization 13
4.1.2 DCT 14
4.1.3 Convolution 14
4.1.4 Sum 14
4.1.5 Histogram 14
4.2 EXPERIMENTAL SETUP 15

5 RESULTS 17

5.1 FROM CUDA TO OPENCL 17
5.2 ARCHITECTURE SPECIFIC MAPPINGS 20
5.2.1 NVIDIA GPU 20
5.2.2 AMD GPU 25
5.2.3 Intel Multi-core CPU 26
5.2.4 An Absolute Comparison 28
5.3 PERFORMANCE PORTABILITY 30
5.4 GENERIC KERNEL DEVELOPMENT 33

6 RELATED WORK 38

7 CONCLUSION 39

7.1 FUTURE WORK 40

BIBLIOGRAPHY 41

APPENDIX A. PERFORMANCE EVALUATION OPENCL VS CUDA 44

1 INTRODUCTION

Recent advances in processor architecture have marked a milestone in the history of computing. Prior processor development was mainly focused on single thread performance (using techniques such as pipelining, branch prediction, instruction-level parallelism and out-of-order execution) to speed up the execution of a single threaded program. For many years the industry has successfully followed this trajectory, motivated by the advancements in chip fabrication which still adhere to Moore's law. More recently, this trajectory suddenly stopped by running into two walls, the power- and memory walls. The power-wall limits developers to further increase clock-speeds, since they would cause serious heat problems on a chip. The memory-wall refers to the growing disparity between the speed of a processor and the speed of off-chip memory. Limited by these walls, processor architecture development had to make some drastic changes to keep the industry alive. This resulted into the design of parallel compute architectures, which nowadays dominate the market.

The forced shift from sequential to parallel architectures has burdened programmers with a difficult challenge. To improve program execution, they have to rewrite their application in order to fully exploit the capabilities of a parallel architecture. Developing parallel applications is far more complex than sequential programming. Besides the functionality of an algorithm, a programmer must also deal with work division, communication and synchronization issues that are inherent to parallel program execution. Several programming models have been introduced to help programmers bridge the gap between the sequential and parallel programming worlds. These models include CUDA [**1**] and ATI Stream Technology [**2**], for targeting NVIDIA GPUs and AMD GPUs respectively, OpenMP [**3**] and more recently Intel ArBB [**4**] to target multicore CPUs and the Cell SDK [**5**] to target the Cell B.E. processor. The aforementioned programming models have the intention to expose the architectural specifics to the programmer and usually come with a dedicated compiler and driver structure that targets the specific architecture. It certainly supports a programmer in efficiently mapping an algorithm to a specific parallel architecture, but a problem occurs when it is decided to map the algorithm on another or even multiple architectures at once. Due to the architectural specifics - that are only accessible through a vendor dependent programming interface - a programmer must repeat the effort of mapping and optimizing an algorithm at each change of target platform.

The problem of portability required the industry to introduce a new programming framework. The result is OpenCL (Open Computing Language), the first open standard for parallel programming across CPUs, GPUs and other processor architectures, aimed at portable and efficient access for programmers to the power of heterogeneous processing platforms [**6**]. OpenCL is maintained by the Khronos Group and is a product of the co-operation of many industry leading companies.

1.1 PROBLEM STATEMENT

OpenCL appears to be a promising standard for developers of parallel compute applications, that do not want to tie their application to a specific platform, or want to target multiple platforms from a single source-code base. It offers the developer a solution to the problem of rewriting the application each time a different platform is targeted. For OpenCL to be accepted by developers,

they must be convinced into the uniqueness and usefulness of OpenCL. Two properties that will be of valuable interest for a developer to use OpenCL are:

1. The performance of an OpenCL program as compared to an architecture's native implementation of that same program.

2. The performance portability of an OpenCL program across different architectures.

The first property is obvious, if an OpenCL program performs poor as compared to an architecture's native program implementation, a developer - mainly focused at high performance execution on a single platform - would undoubtedly stick to the native programming standard. Ideally, the OpenCL program should reach at least equal performance as compared to the architecture's native implementation. The second property concerns performance portability, the degree in which the performance on one architecture ports to another architecture. Ideally, an OpenCL implementation that reaches optimal performance on a particular architecture, should also reach optimal performance when it is run on another architecture.

In this thesis the two properties mentioned above will be investigated by performing various experiments. The experiments include OpenCL- as well native implementations of algorithms targeted at different architectures. Also, a method will be introduced to develop a single, generic OpenCL implementation, which is not targeted at one architecture in particular.

1.2 OUTLINE

This thesis is structured as follows. First, chapter 2 will introduce the OpenCL framework by explaining the platform-, execution- and memory models according to the OpenCL specification. A brief explanation of an OpenCL program will clarify the structure and commands used in OpenCL programming. Continuing, chapter 3 discusses the three architectures that will be used as hardware platforms for the experiments. Only architectural details that are relevant for this work will be treated. Along with the architectures, roofline models will be introduced which are used later on for analysis and reasoning about performance results. The algorithms that are used for benchmarking as well as the experimental setup will be discussed in chapter 4. A detailed analysis of the results will be given in chapter 5, from which some intermediate conclusions regarding the suitability of OpenCL will be drawn. The chapter is finalized by providing a method used to improve the performance portability of an OpenCL program, along with the results after applying the method. The uniqueness of the research performed in this work and the proposed method will be given by comparing it to other work from literature that is closely related to this work. Finally, the conclusions from this thesis are summed up and some possibilities of future work will be given.

2 THE OPENCL PROGRAMMING FRAMEWORK

As today's computer systems often include highly parallel GPUs, CPUs and other types of processors, enabling software developers to take full advantage of these heterogeneous processing platforms becomes important. OpenCL attempts to fulfill this demand. The OpenCL specification is the result of the contributions of many people, representing a cross section of the desktop, hand-held, and embedded computer industry [**6**]. OpenCL includes a language, an Application Programming Interface (API), libraries and a runtime system to support software development for any OpenCL supported device in a system. The language is a subset of the C99 standard, with added extensions that enable the expression of parallelism in a program.

2.1 PLATFORM MODEL

The OpenCL platform model is depicted in Figure 2-1. A host (usually a CPU) is connected to one or more OpenCL *Compute Devices* (e.g. a GPU or a CPU). A *Compute Device* consists of one or more *Compute Units*, which are further divided into one or more *Processing Elements*, on which the actual processing takes place.

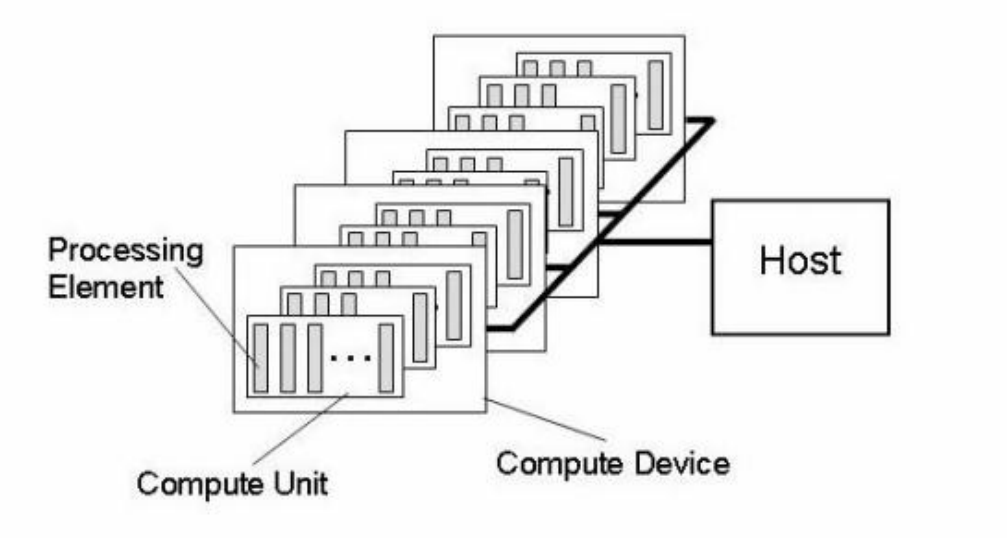


Figure 2-1: OpenCL platform model

2.2 EXECUTION MODEL

Execution of an OpenCL program can be separated in two parts: host code that runs on the host device and device code that runs on one or more *Compute Devices*. The host code defines the context of the device code and manages its execution. The device code exists of one or more kernels, in which the actual processing takes place. The execution of a kernel on a device is defined by an index space, called an *NDRange*. An *NDRange* is an N-dimensional index space, where N can be one, two or three. Figure 2-2 gives an example of a 2-dimensional index space. OpenCL uses a two level hierarchical model to subdivide the work-items, which is similar to the thread-hierarchy as used by the CUDA programming framework [6]. An *NDRange* consists of a one, two or three dimensional space of work-groups. A work-group on its turn consists of a one, two or three dimensional space of work-items. This model provides a natural way to handle groups of elements such as a vector, a matrix or a volume. A work-item does the actual processing and is mapped on a *Processing Element*.

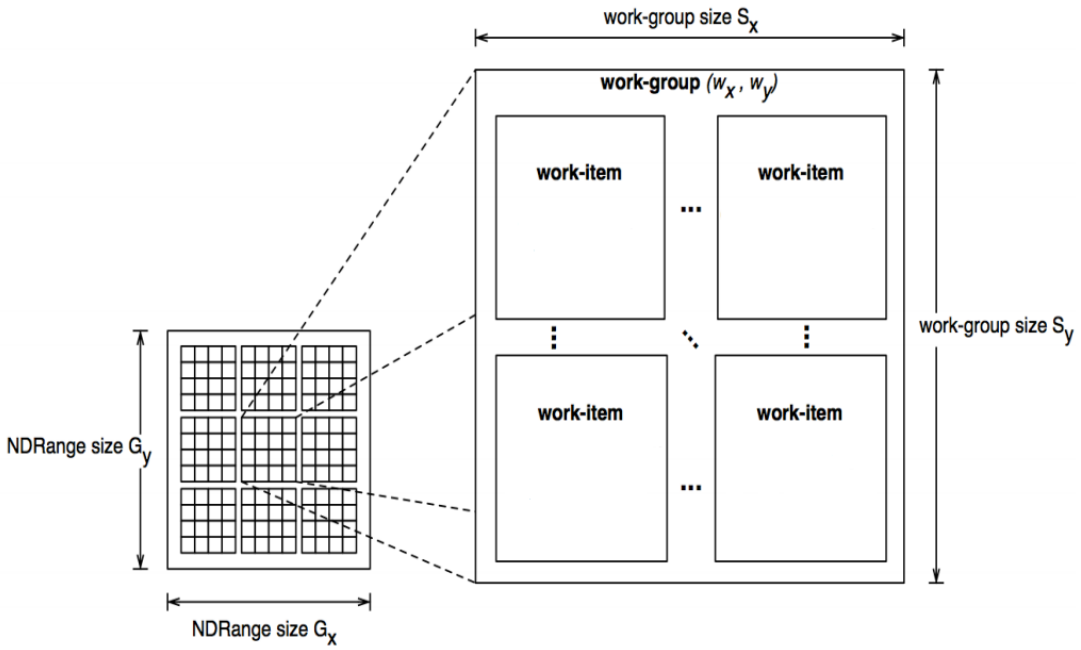


Figure 2-2: OpenCL execution model

2.3 MEMORY MODEL

Figure 2-3 indicates the memory model used inside a *Compute Device*. The execution model as discussed in chapter 2.2 is mapped onto this model. A work-group is mapped onto a *Compute Unit*, whereas a work-item runs on a PE (Processing Element). Work-items that execute a kernel have access to different memory regions. Global memory permits read/write access to all work-items of all work-groups. Global memory accesses might be cached, depending on the capabilities of the *Compute Device*. Constant memory is a region of the global memory that remains constant during the execution of a kernel. Furthermore, local memory is a memory region which is only accessible by the work-items inside the same work-group. Local memory can be used to share data between the work-items in a work-group. Depending on the capabilities of the device, local memory may be mapped onto dedicated memory regions of the device or, if not available, onto sections of global memory. Private memory is only accessible to the corresponding work-item, which will not be visible to other work-items.

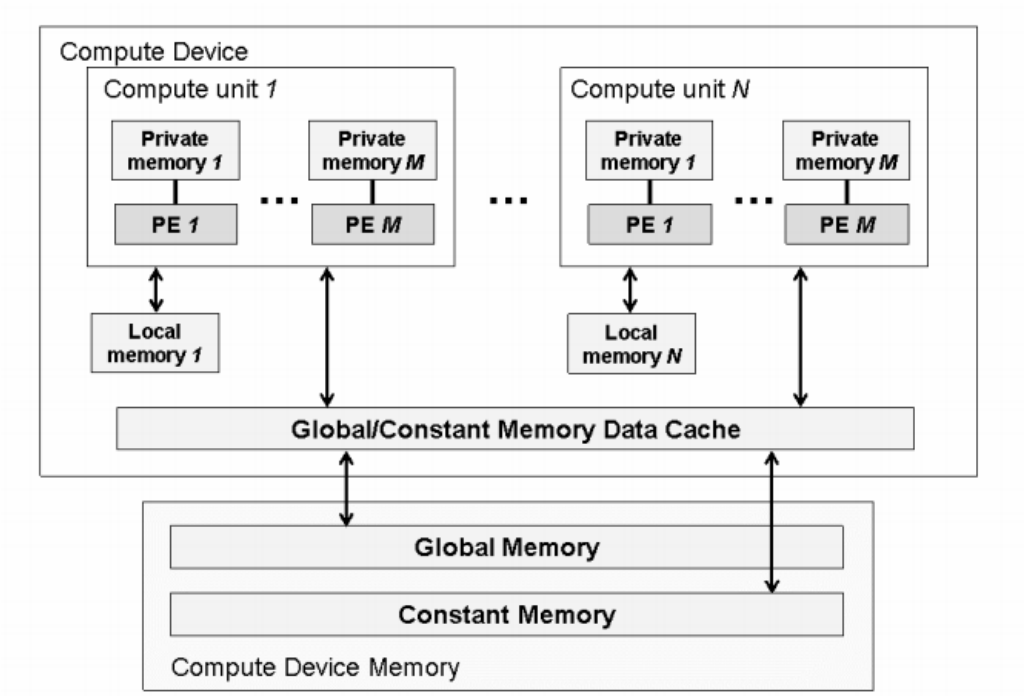


Figure 2-3: OpenCL memory model

2.4 OPENCL PROGRAM STRUCTURE

An abstract overview of the flow inside an OpenCL program is visualized in Figure 2-4. A clear separation is made between processes running on the host and processes running on the target device. The host code is usually performed on a CPU, whereas the device code could run on any OpenCL supported device. The device could also be a CPU, in which case both host- as device code will run on the same device.

It all starts on the host by using the OpenCL API to query the system for OpenCL supported devices, followed by selecting a device which will be used as a target device for running the OpenCL kernel upon. Then, a *context* is created which is used by the OpenCL runtime for managing objects such as memory, program and kernel objects and for executing kernels on the device specified in the *context*. Operations on these objects are performed using a *command-queue*, which must be created as part of the *context*. The next step is to read the OpenCL kernel code and compile it into a binary code file. At this point in the program the target device is known. The OpenCL Installable Client Driver (ICD) feature ensures the kernel is compiled for the chosen target device. In most implementations the OpenCL source code is first compiled into an intermediate representation which is device independent. This intermediate code is optimized as much as possible, before the final code for the selected device is generated by the device's code generator (as part of the device's OpenCL driver/runtime infrastructure). The data that is needed by the kernel must then be copied to the target device. Note that this is only necessary when the host and device don't have a shared memory space. For example, when both the host and the device code are running on a CPU (or a fusion of a CPU and a GPU on the same chip), the copying of data between the two is not necessary, since both have access to the same memory space. Now everything is set, the host signals the device that it can start executing the kernel. At this moment the host will stall and wait for the kernel to be

finished. When the kernel finishes executing on the device, the results can be copied back to the host.

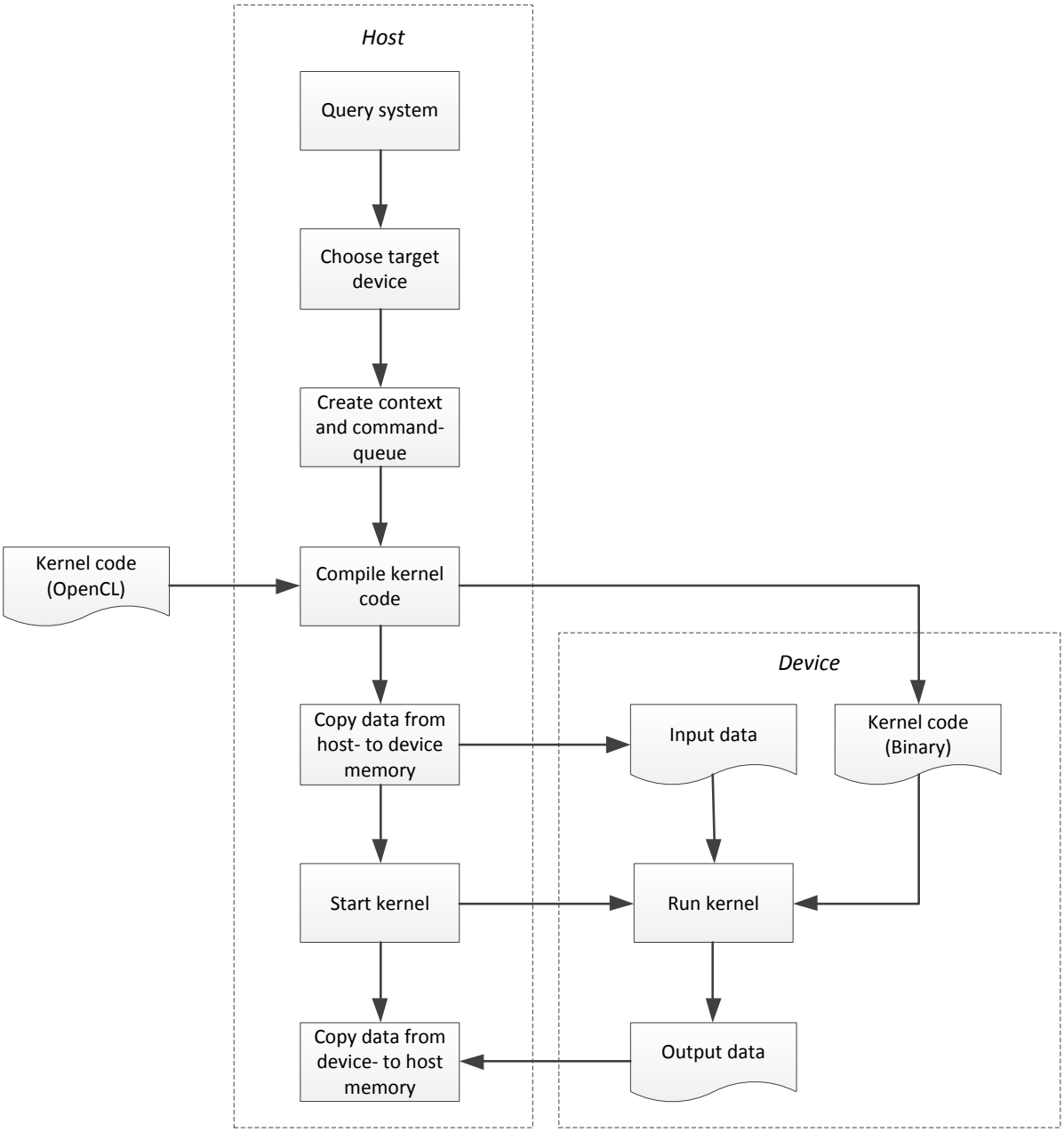


Figure 2-4: OpenCL program flow

3 ARCHITECTURES

This chapter will introduce and discuss the architectures that will be used for the experiments. Currently, OpenCL has been adopted by a growing number of large companies [**7**]. The devices used for the experiments are listed in Table 3-1. Three different vendors are represented, which are currently leading the marked in producing CPUs and GPUs.

Vendor	Model (architecture)	SDK (driver version)
NVIDIA	GTX 470 (Fermi)	NVIDIA CUDA SDK 4.0.1 (270.41.19)
AMD	HD 5850 (Evergreen)	ATI-Stream-v2.3 (CAL 1.4.1417)
Intel	Core i7-930 (Nehalem)	Intel OpenCL SDK 1.1 beta

Table 3-1. OpenCL target devices

3.1 NVIDIA FERMI GPU

The Fermi architecture is currently NVIDIA's latest GPU architecture [**8**]. It is the successor of the Tesla architecture, which was the first GPU that unified the vertex and pixel processors into a single architecture, enabling high-performance parallel computing on GPUs [**9**]. Along with the introduction of the Tesla architecture, NVIDIA introduced the CUDA programming environment [**1**], which exposes the parallel compute architecture of the GPU to programmers. This exposure enables a programmer to efficiently map an algorithm onto an NVIDIA GPU, exploiting its architectural specifics.

Figure 3-1 illustrates an abstract overview of the Fermi architecture. The device is built up of a number of *Streaming Multiprocessors* (SMs). Each SM resembles an SIMD (Single Instruction Multiple Data) processor, containing 32 cores and 4 *Special Function Units* (SFU). Each core has a fully pipelined integer *Arithmetic Logic Unit* (ALU) and *Floating Point Unit* (FPU). The SFUs are used for transcendental instructions, such as sine and cosine.

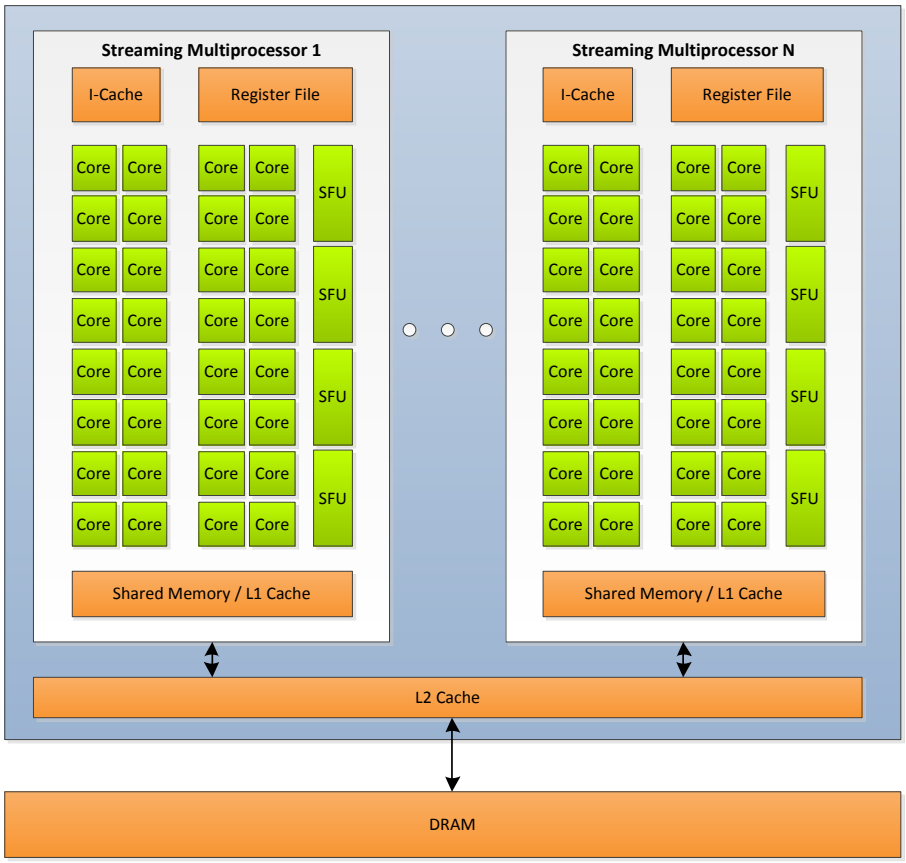


Figure 3-1: NVIDIA Fermi architecture

The memory hierarchy consists of several layers. Each layer has a distinguishable access property. Closer to the processing elements, the memory access latencies are smaller (along with the size of the memory). The DRAM is positioned off-chip and consequently has the largest access latency. Data from DRAM is cached in a L2 cache, which is shared by all SMs. Each SM has its own region of L1 cache combined with a *Shared Memory* region. The amount of L1 cache versus *Shared Memory* is configurable by the programmer. The main difference between the L1 cache and the *Shared Memory* is that the latter resembles a scratchpad memory, meaning that a programmer has to explicitly read data from and write data into the memory, while the contents of a cache are managed by hardware. The closest memory to the processing elements is the *Register File*, which has the fastest memory access time.

3.2 AMD EVERGREEN GPU

A simplified block diagram of the Evergreen GPU architecture is shown in Figure 3-2. On this level of abstraction, the device architecture is similar to the NVIDIA Fermi architecture. The device contains a number of *Compute Units*, which in turn each contain 16 *Stream Cores*. A *Stream Core* is arranged as a five-way (or four-way, depending on the GPU type) Very Long Instruction Word (VLIW) processor [**10**], each containing five (or four) PEs. One of the PEs inside the *Stream Core* has the ability to perform transcendental operations. The remaining PEs can execute single-precision floating point or integer operations.

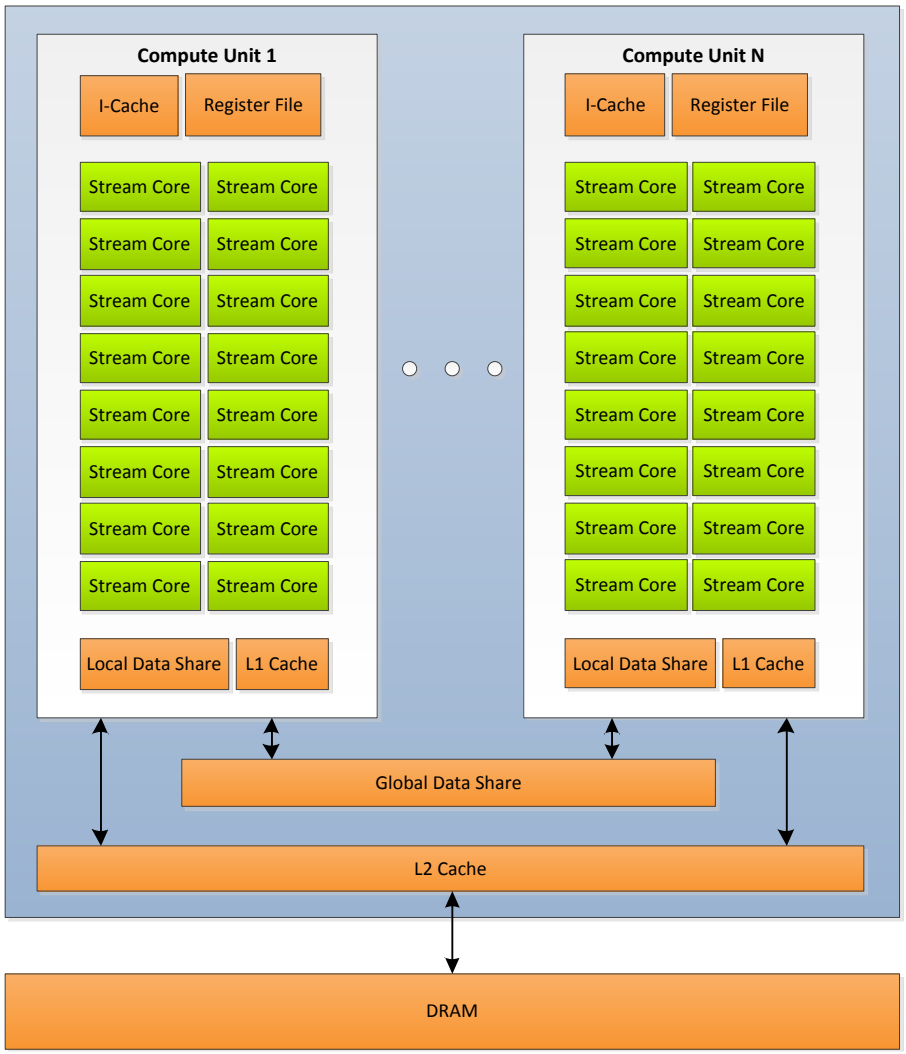


Figure 3-2: AMD Evergreen architecture

The memory hierarchy also compares to the hierarchy available on the NVIDIA Fermi. The main differences can be found in a separated L1 cache and *Local Data Share* (LDS), and the existence of a *Global Data Share* (GDS), which is not available on the Fermi architecture. The LDS is a scratchpad memory shared by the all *Stream Cores* inside the same *Compute Unit*. The GDS is also a scratchpad memory, but can be accessed by all *Stream Cores* from all *Compute Units* on the device.

3.3 INTEL NEHALEM CPU

Intel has a wide range of CPU architectures, primarily based on the x86 architecture, which was first launched in 1978 [**11**]. Over the past years, many additions and extensions have been added to the x86 instruction set, including branch prediction, superscalar- and out-of-order execution, vector instructions and simultaneous multithreading. Where the evolvement of CPU architectures used to be primarily focused on deep pipelining to increase clock frequencies, the last decade the design has moved to multi-core architectures, putting more than one processor core onto a single chip [**12**].

Figure 3-3 illustrates a top-level overview of the Intel Nehalem architecture. A single Nehalem device consists of up to eight cores, each one containing a full x86 processor. Compared to both GPUs as discussed above, the Intel CPU does not contain a local scratchpad memory. Instead,

each core has its own L1 and L2 caches, whereas a L3 cache is shared by all cores. *Hyper-Threading* has been reintroduced in the Nehalem architecture, enabling two threads to share the function units of a single core. A processor core is able to execute vector instructions on multiple data elements in parallel, with a vector width of 128 bits.

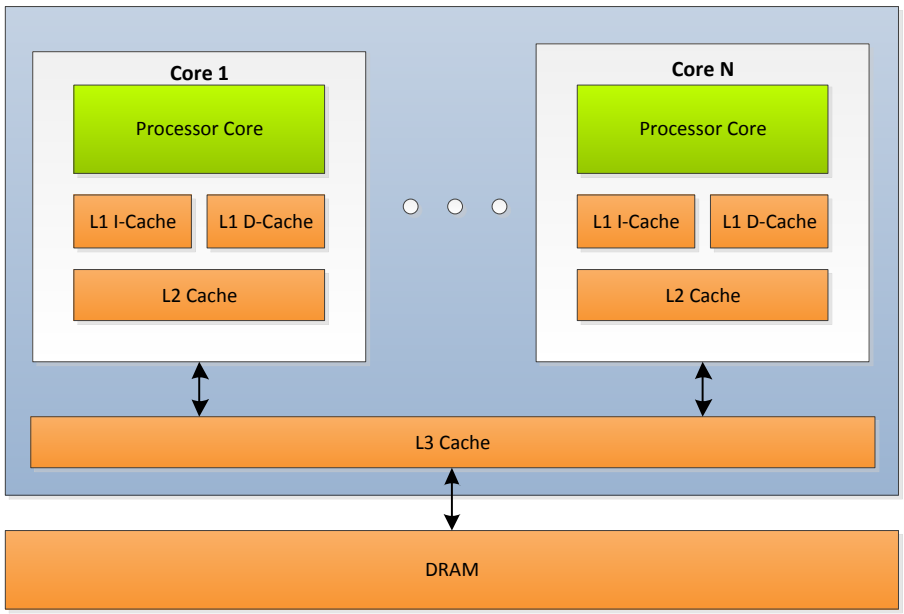


Figure 3-3: Intel Nehalem architecture

3.4 OPENCL MAPPINGS

For a programmer to efficiently write OpenCL code that runs on one of the architectures, it is important to know how the OpenCL model maps onto the architecture. Table 3-2 summarizes for each device how it maps on each OpenCL architectural structure.

OpenCL	AMD Evergreen GPU	NVIDIA Fermi GPU	Intel Nehalem CPU
Compute Device	AMD GPU	NVIDIA GPU	Intel CPU
Compute Unit	Compute Unit	Streaming Multiprocessor	Thread
Processing Element	Stream Core (5-way VLIW)	CUDA Core	Processor Core
Global Memory	DRAM (off-chip), cached	DRAM (off-chip), cached	DRAM (off-chip), cached
Constant Memory	DRAM (off-chip), cached	DRAM (off-chip), cached	DRAM (off-chip), cached
Local Memory	Local Data Share (on-chip)	Shared memory (on-chip)	DRAM (off-chip), cached
Private Memory	Registers	Registers	Registers
Global/Constant Memory Data Cache	Constant read-only-caches: L1 (per CU) and L2 cache	Constant read-only-caches: L1 (per SM) and L2 cache	Global read/write caches: L1/L2 (per core) and L3 cache

Table 3-2: OpenCL mappings

On the Intel CPU, *Hyper-Threading* enables the mapping of two OpenCL *Compute Units* (Intel threads) onto one OpenCL *Processing Element* (Intel Core). As mentioned before, the Intel Nehalem

architecture has no dedicated scratchpad memory region that can be used to map OpenCL local memory upon. For Intel to comply with the OpenCL specification, it was decided to map the local memory onto regions of the global memory (off-chip DRAM memory). This choice discards the benefit of using the local memory as a fast low latency memory for frequently accessed data. Instead, on the Intel CPU a programmer should rely on the available caches to cache the frequently accessed data from global memory.

The *Global Data Share* that is available in the AMD Evergreen architecture (see Figure 3-2), is currently not supported by the OpenCL implementation. There is no OpenCL equivalent for this type of memory, which means that it cannot be used within an OpenCL implementation.

3.5 PERFORMANCE MODELS

When optimizing an algorithm for performance it is important to know the architectural details of the development platform. Besides that, awareness of the performance limits that come with the device should be kept in mind. Here, a roofline model is introduced which will be used later on in this work for bottleneck analyses and reasoning about performances. The roofline model, as described in [**13**], provides a performance oriented programmer insight in the bounds and bottlenecks of an algorithm-device combination. Basically, the roofline model can be split in two sections, one where the algorithm is bound by the device's bandwidth and one where the algorithm is bound the device's computational ability. But, more complex bounds are possible.

Table 3-3 summarizes the device specifics used to create the roofline models. Performance and bandwidth numbers of a device are usually expressed as the theoretical peak that can be reached when all the device's resources are fully utilized. For example, when considering the theoretical peak compute performance of a GPU, it is assumed that only *Fused Multiply Add* (FMA) instructions are used. Most GPUs are able to perform an FMA instruction in one clock cycle. In practice though, the FMA instruction will be rarely used, resulting in an unreachable performance roofline. For this reason, besides the theoretical roofline an additional ceiling will be introduced, which assumes no FMA instructions are used on the GPUs. On the CPU the ceiling is chosen to have no vector instruction set (SSE) usage. This enables the analysis of the amount of vector instructions an algorithm uses, when it falls between the theoretical roof (full vector set utilization) and the additional ceiling (no vector instructions). Furthermore, an additional bandwidth ceiling is introduced which is obtained by performing streaming bandwidth tests on each device.

Device	Processing Cores	Core Frequency (MHz)	Performance (GFLOP/s)		Bandwidth (GB/s)	
			Peak Single Precision	*Without FMA (GPU) / SSE (CPU)*	*DRAM Bandwidth*	*Stream Bandwidth*
NVIDIA GTX 470	448	1215	1088	544	134	117
AMD HD5850	1440	725	2088	1044	128	126
Intel Core i7-930	4	2800	44	11	25	11

Table 3-3: Architectural performance specifics

Figure 3-4, 3-5 and 3-6 show the roofline models, which are created by applying the following formula:

$$Attainable\ Performance\ = min\begin{cases} Peak\ Performance \\ Peak\ Bandwidth\ \ x\ Arithmetic\ Intensity \end{cases}$$

Note that in the roofline graphs both scales are logarithmic. What can be observed is that for both GPUs in practice it is possible to nearly reach the theoretical peak bandwidth, while on a CPU it is hard to reach the theoretical peak.

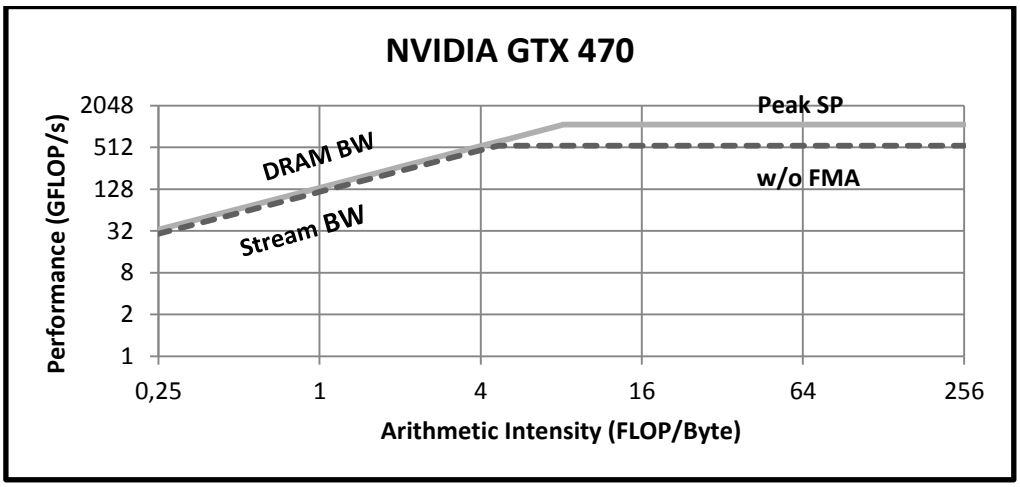


Figure 3-4: Performance roofline model NVIDIA GPU

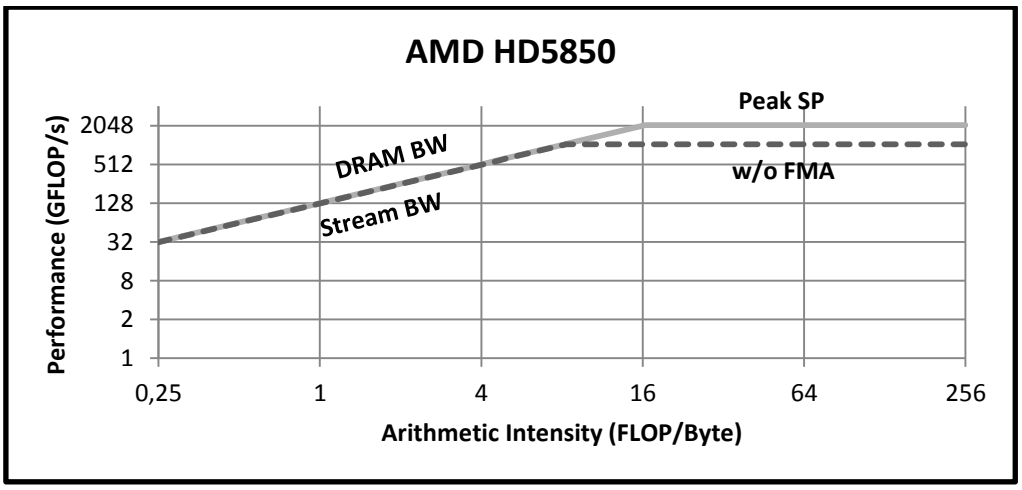


Figure 3-5: Performance roofline model AMD GPU

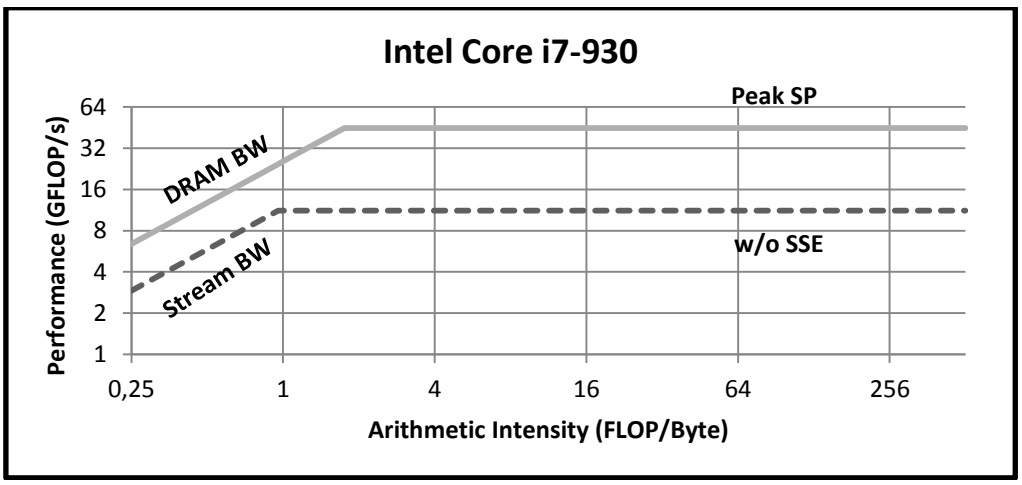


Figure 3-6: Performance roofline model Intel CPU

4 METHODOLOGY

To be able to measure and compare performances, suitable benchmark applications are needed. For this purpose, several algorithms are chosen which are widely applied in industry. The algorithms are taken from the domain of image-processing, which finds its roots in applications as computer-vision and medical-imaging. Since these algorithms have a high potential of parallel execution, they are very suitable to be efficiently mapped upon highly parallelized architectures.

4.1 BENCHMARK ALGORITHMS

Within the Electronic Systems Group of the Electrical Engineering department of the TU/e, a number of image-processing algorithms have already been successfully implemented in the CUDA language and optimized for the NVIDIA GPU architecture [**14**] [**15**]. Furthermore, an algorithm classification has been constructed according to the data access patterns of an algorithm [**16**]. Each class has a particular mapping on the architecture, enabling potential architectural performance improvements. For example, if there is data reuse in an algorithm, the scratchpad memories on a GPU can be used to speed up memory accesses of frequently accessed data.

The algorithms used in this work are chosen to have a wide variety in arithmetic intensities. This enables the use of a wide range on the architectural roofline models as discussed in chapter 3.5. Table 4-1 summarizes the algorithms along with the algorithm classification and the performance bound. The classification represents the input/output properties of the algorithm, i.e.

$$\text{input size} \mid \text{input pattern (property)} \longrightarrow \text{output size} \mid \text{output pattern (property)}$$

where AxB is the size of the input/output image. To see the impact of different workloads varying input sizes are fed to the algorithms. Note that the performance bound is not directly related to a classification. For example, an algorithm from the classification Element to Element might also be bound by a device's compute performance.

Algorithm	Classification	Performance Bound
Binarization	AxB|Element ⟶ AxB|Element	Bandwidth
DCT	AxB|Tile(8x8) ⟶ AxB|Tile(8x8)	Computation
Convolution	AxB|Neigh(5x5)⟶AxB|Element	Computation
Sum	AxB|Element⟶1|Shared	Bandwidth/reduction
Histogram	AxB|Element⟶256|Shared	Bandwidth/conflict

Table 4-1: Algorithm characteristics

4.1.1 BINARIZATION

Binarization is an algorithm that converts an image of up to 256 gray levels into an image of two levels, i.e. black and white. Whether a pixel value from the input will become black or white in the output image, depends on a threshold value. Binarization is frequently used as a pre-processor for other algorithms, which only accept a black and white image as input.

The arithmetic intensity of Binarization is low (only a few operations are needed per byte memory access). Therefore, on most devices it will be bound by memory bandwidth. The amount of exploitable parallelism is high; each pixel can be processed independently of all others. But as

Binarization is typically limited by the device's memory bandwidth, more parallelism (in terms of the number of threads) will at some point not give a performance improvement anymore.

4.1.2 DCT

DCT is an abbreviation for Discrete Cosine Transform, which is an algorithm used in various applications, mostly with the purpose of data compression. There are many variants of the DCT algorithm [**17**]. The version used in this work is the 2-dimensional type II DCT with a fixed size of 8 by 8, as commonly used in JPEG compression. For each block of 8 by 8 pixels from the input image, it calculates the DCT for the whole block and stores it at the corresponding block in the output image.

The implementation of DCT has a computational complexity of $O(n^2)$, where n is the data input size. On most devices this means that DCT will be bound by the computational performance of the device, since it has a high arithmetic intensity.

4.1.3 CONVOLUTION

Convolution is a widely applied algorithm in the domain of image processing and can be used as a standalone algorithm. The basic idea is for an output pixel to be the average of a mathematical operation on the neighboring pixels in the input image. Because the size and the values of the filter are adjustable, the algorithm can perform many different functions such as low/high pass filtering or image blurring. The function used in this work is a 2-dimensional blurring filter, which averages the value of a pixel with its neighboring pixels in the input image, using a filter size of 5 by 5.

The arithmetic intensity of a convolution algorithm is dependent on different factors. An important factor is the size of the filter. For a small filter size the arithmetic intensity is low, resulting in a bandwidth limited performance. For a large filter size the arithmetic intensity of the algorithm increases towards the computational performance bound. The filter size used in this work is 5 by 5, resulting in a computational performance bound on most devices.

4.1.4 SUM

Sum is a reduction algorithm which takes as input an array of values, and outputs a single value which is the sum of all values in the input image. Since the arithmetic intensity is low (one addition per input element), one would expect the algorithm to be bandwidth bound. This is true for the sequential implementation of the algorithm, but for a parallel implementation the sum is calculated over several reduction steps, where each step requires a synchronization mechanism for correct functionality. A synchronization step might introduce significant overheads. For this reason the performance bound of this algorithm will be bandwidth/reduction bound.

4.1.5 HISTOGRAM

An image histogram shows the distribution of pixel values in an image. It counts for each of the 256 gray-scale values the occurrence in the image. The algorithm only involves the calculation of the histogram, of which the result can be used for analytical reasons or as part of a more sophisticated function such as histogram equalization.

Histogram has a low arithmetic intensity: one incremental addition is needed for each input element. A sequential implementation would in most cases be limited by the available bandwidth. However, a problem occurs when parallelizing the algorithm. It will cause multiple threads to update

the values of the same histogram. When multiple threads must increment the same histogram value, there will be a memory conflict, which needs to be resolved by a special mechanism, introducing additional overheads. A solution would be to let each thread have its own histogram, but this requires an increased amount of memory when running many threads in parallel.

4.2 EXPERIMENTAL SETUP

The main goal of this research project is to investigate OpenCL as a programming standard for parallel processing architectures. As mentioned before in chapter 1.1, two properties will be of most interest:

1. The performance of an OpenCL program as compared to an architecture's native implementation of that same program.

2. The performance portability of an OpenCL program across different architectures.

The first property concerns the qualification of OpenCL to replace an architecture's native programming framework. The second property addresses the advantage OpenCL possesses over other parallel programming interfaces, i.e. cross-vendor support. In this case, OpenCL will be evaluated as a single code-base to target multiple devices from different vendors. Two important qualification metrics will be used, functional portability and performance portability. Functional portability means that a single OpenCL implementation performs functionally equivalent on different devices. Performance portability concerns the degree in which the performance of an OpenCL implementation on one device differs from the performance of that same implementation on another device. Ideally, one prefers to have one implementation which offers full functionality and maximal performance on all supported architectures.

To be able to reason about the properties, experiments will be conducted of which a setup can be found in Figure 4-1. A subdivision in three levels can be made. The top level (algorithm level) comprises the functional description of each algorithm. Following is the implementation level, which involves the implementation of each algorithm using a particular language framework along with device specific optimizations. The implemented algorithms will then be mapped on the hardware level, consisting of the actual hardware architectures.

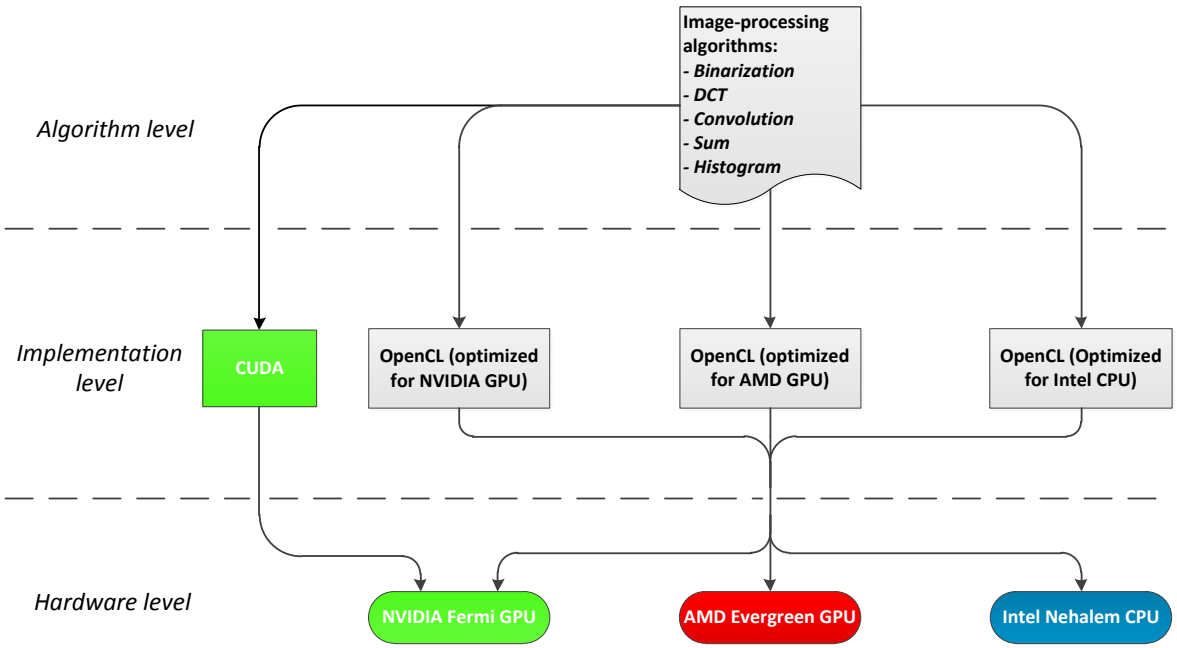


Figure 4-1: Research plan overview

Starting from the algorithms as mentioned in chapter 4.1, implementations in CUDA as well as OpenCL will be mapped upon the NVIDIA GPU. The execution times of the kernels running on the device will be used as a metric to compare the performances. From there on, the OpenCL implementations of the algorithms will be optimized for the remaining architectures: AMD GPU and Intel CPU. The roofline models (as discussed in chapter 3.5) will be used to analyze the performance of each algorithm-device mapping. To analyze the functional- and performance portability of an OpenCL implementation, each single optimized kernel version will be mapped upon all other devices.

5 RESULTS

This chapter will present and discuss the results obtained from performing the experiments as described in chapter 4.2.

5.1 FROM CUDA TO OPENCL

CUDA [**1**] is the current standard to program NVIDIA GPUs. Programmers use C for CUDA to write specific kernels that can run on NVIDIA GPUs. From a functional point of view, the CUDA language and the OpenCL language are similar. The main differences can be found in the API (used in the host code) and language specific keywords (used in the device code). Table 5-1 matches the keywords from CUDA and OpenCL which have the same functional meaning. OpenCL offers some additional keywords for which there is no corresponding keyword in CUDA. In CUDA, such a keyword can be replaced by using a combination of other keywords. For example, to obtain the global ID of a work-item (which gives the ID of the work-item with respect to all work-items in the whole NDRange), in CUDA this can be obtained by multiplying *blockIdx* with *blockDim* and adding *threadIdx*.

CUDA	OpenCL	Meaning
__global__	__kernel	Kernel declaration
__shared__	__local	Local memory declaration
__device__	__global	Global memory declaration
threadIdx	get_local_id()	Get ID of work-item inside work-group
blockIdx	get_group_id()	Get ID of work-group inside NDRange
gridDim	get_num_groups()	Get number of work-groups inside NDRange
blockDim	get_local_size()	Get number of work-items inside work-group
__syncthreads()	Barrier()	Synchronization point
Calculate manually	get_global_id()	Get ID of work-item inside NDRange
Calculate manually	get_global_size()	Get number of work-items inside NDRange

Table 5-1: Different keywords in CUDA and OpenCL

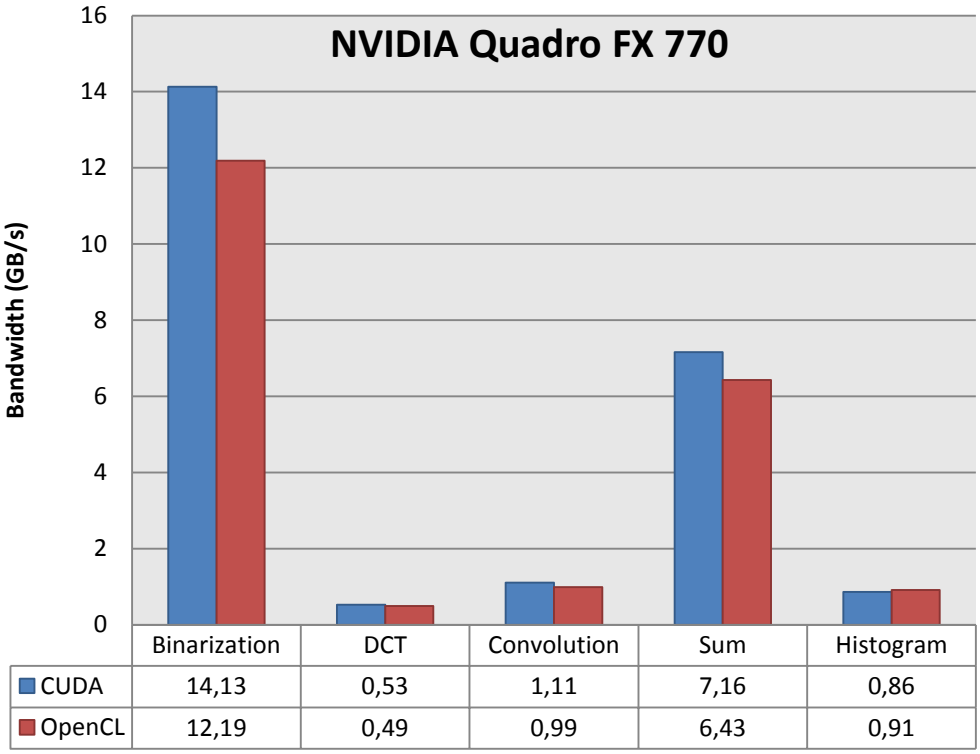
On the kernel level, CUDA and OpenCL are functionally equal. In other words, what can be done in CUDA can also be done in OpenCL and vice versa. This enables a fair comparison between CUDA and OpenCL on an NVIDIA GPU, since both languages are able to equally map an algorithm onto the NVIDIA GPU. Figure 5-1 gives the performance results when running the same implementations of the algorithms in CUDA and OpenCL on an NVIDIA GPU (NVIDIA Quadro FX 770). The NVIDIA GPU used for this experiment is based on an older architecture, NVIDIA Tesla [**9**]. The performances are expressed in bandwidths of the kernels (higher is better), which are obtained by dividing the data transfer size by the execution time of the kernel. In most cases, the CUDA implementations perform better than the OpenCL implementations (up to 16%), but the overall difference is small.

From [**18**] we know that the performance differences are caused by a difference in kernel launch time and by using different compilers for CUDA code and OpenCL code (the corresponding chapter with detailed explanations can be found in Appendix A). The kernel launch time refers to the time that expires between calling the kernel on the host and the start of executing the kernel on the device. OpenCL has a slightly higher kernel launch overhead, which can have a significant impact on the performance when the kernel execution time is small. An increased kernel execution time will result in a smaller impact of the kernel launch time on the overall performance. Besides the

difference in kernel launch times between CUDA and OpenCL, the use of different compilers for CUDA code and OpenCL code explains the remaining performance difference. Both CUDA code as OpenCL code are compiled into PTX code, which is NVIDIA's high level assembly code to target their GPUs [**19**]. A CUDA implementation and an OpenCL implementation of the same algorithm (which only differ in keywords as mentioned in Table 5-1) result in different PTX assembly codes. Further analysis of the PTX codes shows that the PTX code produced by the OpenCL compiler could be improved, resulting in better performances.

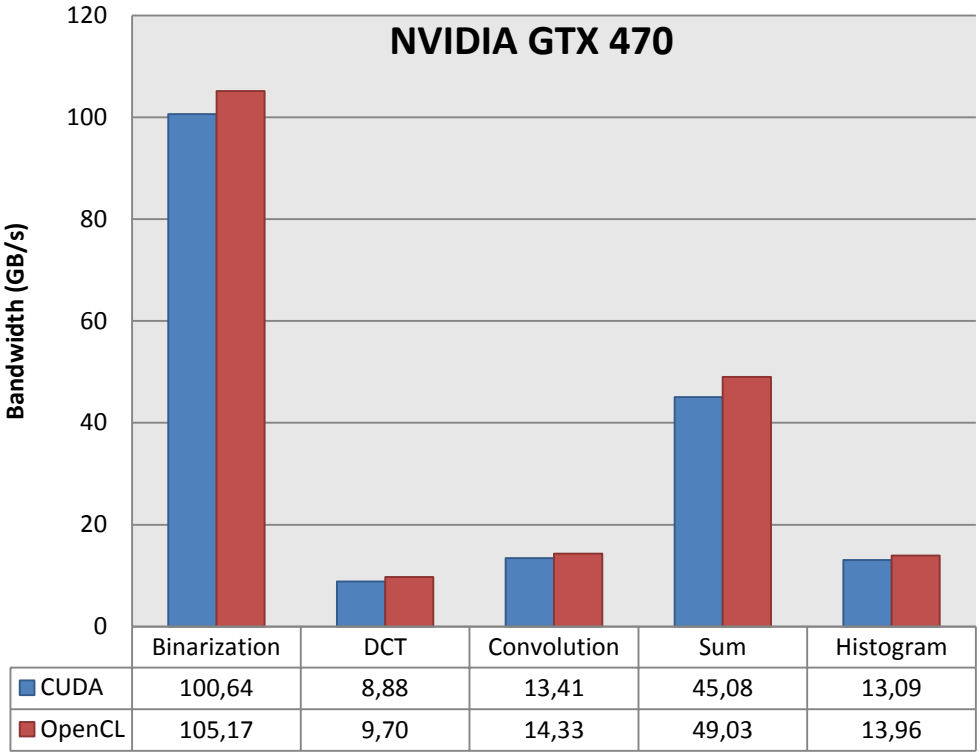
When mapping the same CUDA and OpenCL implementations on a more recent NVIDIA GPU (NVIDIA GTX 470) based on the Fermi architecture [**8**], the performance differences decrease in favor of OpenCL (see Figure 5-2). This is not only caused by targeting another device, but also by using a newer driver and SDK version.

Concluding, the CUDA and OpenCL programming languages show much similarity. The observed performance differences can be largely dedicated to the immatureness of OpenCL, competing against CUDA which has been on the market for a longer while. Taking into account that the OpenCL implementation is improving over SDK versions, it can be considered a decent alternative of CUDA in targeting NVIDIA devices.



	Binarization	DCT	Convolution	Sum	Histogram
CUDA	14,13	0,53	1,11	7,16	0,86
OpenCL	12,19	0,49	0,99	6,43	0,91

Figure 5-1: Benchmark results NVIDIA Quadro FX 770



	Binarization	DCT	Convolution	Sum	Histogram
CUDA	100,64	8,88	13,41	45,08	13,09
OpenCL	105,17	9,70	14,33	49,03	13,96

Figure 5-2: Benchmark results NVIDIA GTX 470

5.2 ARCHITECTURE SPECIFIC MAPPINGS

This chapter will discuss the OpenCL mappings of the algorithms onto the different architectures. For each architecture as discussed in chapter 3, optimization techniques as applied to the algorithms will be explained. The performance numbers are plotted onto the device's roofline models (as introduced in chapter 3.5).

5.2.1 NVIDIA GPU

The NVIDIA GPU is a many-core device capable of running thousands of threads simultaneously. In order to reach maximum performance, a number of optimization steps are necessary to perform.

Maximize parallel execution

Probably the most important step into efficient GPU execution is to maximize the utilization of all resources that are available on the device. First of all, it is necessary to occupy all *Compute Units*. Since an OpenCL workgroup maps onto a single *Compute Unit*, this can be accomplished by launching at least as many workgroups as available *Compute Units* on the GPU. Depending on the number of resources available, multiple workgroups can share a single *Compute Unit*. For reasons of latency hiding (switching between active warps to hide memory- and pipeline latencies) and future scalability it will be advantageous to launch more workgroups than available *Compute Units*. Furthermore, it is important to utilize the resources available inside each *Compute Unit*. For a GPU, a *Compute Unit* is usually built up as an SIMD engine, capable of performing one instruction on multiple data elements. The width of the SIMD unit determines by how many concurrent threads an instruction will be performed. For the NVIDIA GPU, one instruction is performed by 32 threads at once. Such a bundle of 32 threads is called a *warp*. To fully utilize a single *Compute Unit* it is important to have the number of threads inside a workgroup at least the size of a *warp*. If this rule is not followed not all resources of the *Compute Unit* will be used which will not lead to peak performance. Also, in this case it is recommended to launch multiple *warps* per workgroup for the device to be able to hide latencies (by switching between active *warps*). On the other hand, care has to be taken not to launch too many threads per workgroup, since they might consume too many of the *Compute Unit*'s resources which could lead to register spilling.

Memory access coalescing

Besides the utilization of the *Compute Units* on the device, another important optimization step is to maximize the utilization of the memory bandwidth that is offered by the device. On a GPU, global memory loads and stores by multiple threads are, if possible, coalesced by the device into one memory transaction. However, this can only be achieved when certain requirements are met. One condition in which memory coalescing will take place is when sequential threads in a *warp* access corresponding sequential addresses in global memory. Global memory access coalescing can lead to significant performance improvements. In a non-coalesced access pattern, memory accesses will be spread over time which can lead to rapidly increasing performance penalties.

Local memory usage

To speed up memory accesses, GPUs offer a significantly faster on-chip memory. This memory region, which in OpenCL terms is called local memory, is a scratchpad type of memory which means that the programmer has to explicitly write data into and read data from this memory. In fact, on the NVIDIA GPU local memory latency is 100 times lower than global memory latency [**20**]. For this reason, transferring frequently accessed data temporarily into the local memory could give a significant performance improvement. In contrast to caches, where exploiting data locality is handled completely by hardware, a disadvantage of local memory is that it requires additional effort from the programmer to use it.

Loop unrolling

Loop unrolling increases the speed of a program by reducing instructions that are needed to control the loop. A *#pragma* directive must be placed right before the loop, which hints the compiler to enable unrolling the loop.

The optimization techniques discussed above are applied to the algorithms in order to improve the performances on the NVIDIA GPU. The algorithms Sum and Histogram both use an additional optimization strategy, which will be shortly explained further on.

Sum

The mapping of the algorithm Sum onto the NVIDIA GPU uses a highly optimized parallel reduction structure, as described in detail in [**21**]. The implementation uses several stages to reduce the partial sums into a final sum, as depicted in Figure 5-3. At each stage a number of workgroups reduces part of the input array into a single value. It exploits local memory by keeping the partial sums and stores the final sum into global memory. To continue the reduction, all partial results need to be ready and communicated between the workgroups. Unfortunately, the NVIDIA GPU does not support a global synchronization mechanism to do this. For this reason, the launch of a new kernel acts as a synchronization barrier between the different stages. At some point in the reduction tree, the number of workgroups participating in a reduction step will be too small to fully occupy all the *Compute Units* of the GPU (the final stage will only launch one workgroup). But as most of the time is spend in the first couple of stages, this will only have a minor effect on the overall performance.

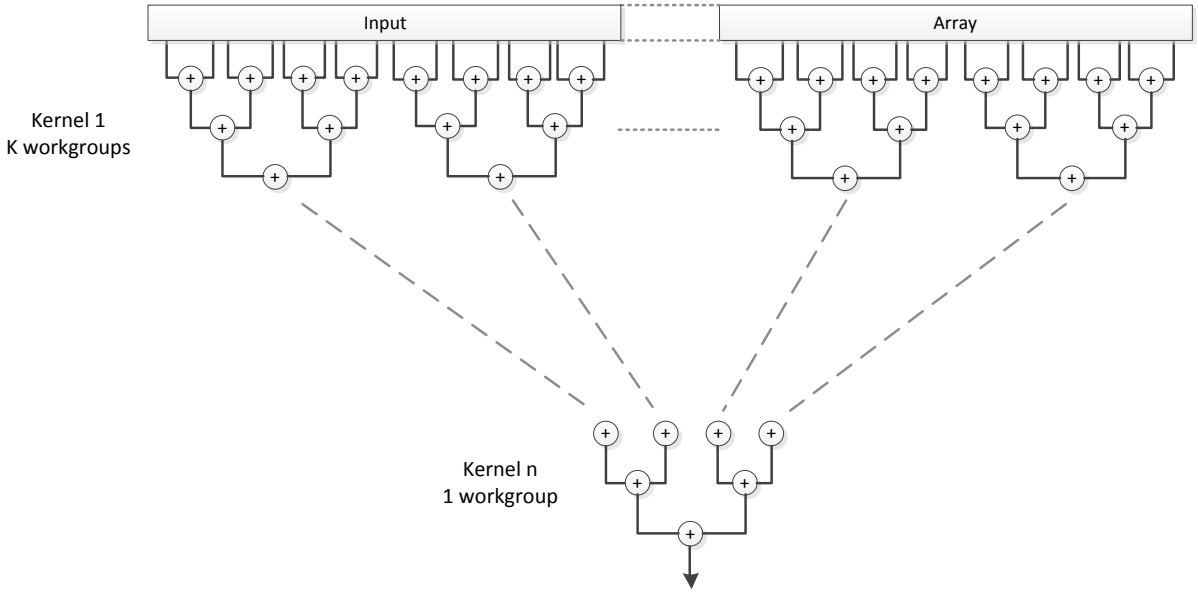


Figure 5-3: Sum parallel reduction

Histogram

To efficiently map the histogram algorithm onto the GPU, the implementation as described in [**22**] is used, for which an abstract overview is depicted in Figure 5-4. An amount of workgroups is launched to read all data from the input array. Each workgroup consists of a number of *warps*, where each *warp* has its own private histogram in local memory space. When a workgroup finishes reading all its data from the input image, the partial histograms as produced by the different *warps* are reduced into a single per-workgroup histogram, which is stored in global memory. For the lack of a global synchronization mechanism, another kernel is needed to synchronize all workgroups and make sure all per-workgroup histograms are stored in global memory. Then a second kernel is launched to sum the per-workgroup histograms into a final histogram.

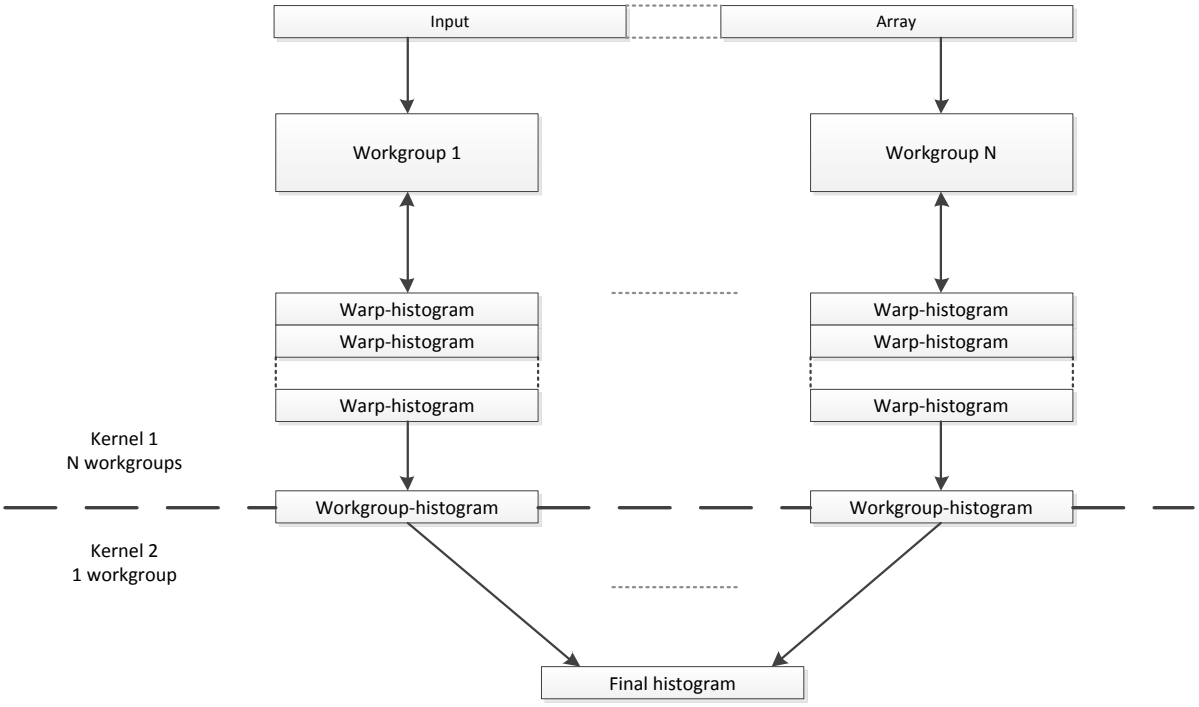


Figure 5-4: Histogram calculation

Table 5-2 summarizes the performance results that were obtained from running the in OpenCL optimized algorithms onto the NVIDIA GPU. The performance utilization indicates the percentage of achievable peak performance that is utilized, which is further visualized in Figure 5-5. Since the theoretical roof is hard to achieve, the additional ceiling is referred to as the achievable peak performance. The arithmetic intensity of an algorithm (FLOPs/Byte) decides where it will be on the horizontal axis of the roofline graph. In the OpenCL source code it is difficult to reason how many FLOPs a single line of code will produce. Therefore, the assembly language output files are used to determine the arithmetic intensities of the algorithms, which should give more accurate numbers.

Algorithm	Processing time (ms)	Performance (GFLOP/s)	Bandwidth (GB/s)	Performance utilization
Binarization	1,150	233,4	116,7	99,72%
DCT	12,513	650,3	10,7	119,48%
Convolution	9,131	316,0	14,7	58,06%
Sum	0,745	320,7	90,1	76,97%
Histogram	4,711	35,9	14,2	12,17%

Table 5-2: Performance results NVIDIA GTX 470

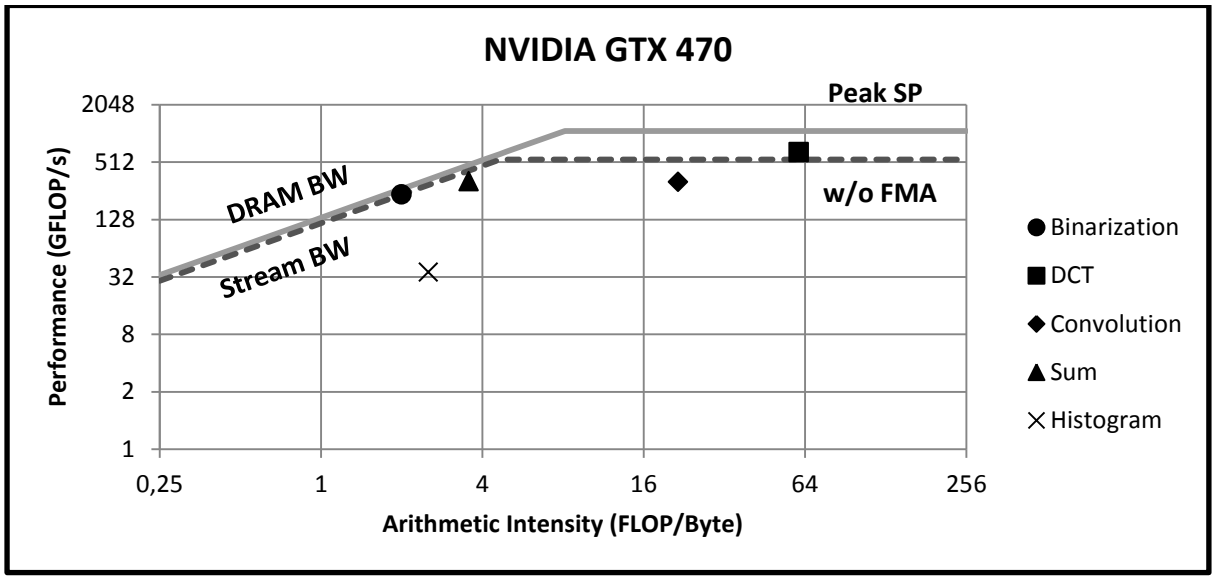


Figure 5-5: Performances on the NVIDIA GPU

- DCT performs better than the peak achievable performance (120%), as its performance is positioned between the theoretical roof and the additional ceiling. The reason for this high performance can be found in the analysis of the assembly file. It contains a considerable amount of FMA instructions, which are not taken into account in the additional ceiling.

- Convolution performs at 58% of the peak achievable performance. The main bottleneck of the Convolution implementation is the way that a block from the input image is pre-loaded into local memory. For calculating the convolution of a pixel, the neighboring pixels are needed. At the borders of a block, this means that additional edges are needed to calculate the convolution of border pixels. The loading of these edges from global memory into local memory are not coalesced, which results in a performance penalty.

- Sum has a quite low arithmetic intensity, thus it is expected to be bound by the peak bandwidth of the device. The reason that the stream bandwidth is not reached can be found in the structure of the implementation as was depicted in Figure 5-3. The implementation uses multiple kernel launches to synchronize between workgroups, which introduces some additional overheads.

- Since it has a low arithmetic intensity, Histogram is also expected to reach the stream bandwidth of the device. The bottleneck of the Histogram implementation on the NVIDIA GPU can be found in creating the partial histograms in local memory (see Figure 5-4). Because a partial histogram is constructed by a *warp* of threads concurrently, the possibility exists that multiple threads need to update the same value in the histogram, resulting in memory collisions. To avoid these collisions, the histogram values are updated by using atomic instructions, which guarantee that only one thread at a time updates a histogram value.

5.2.2 AMD GPU

The architecture of the AMD GPU is comparable to the architecture of the NVIDIA GPU. Similar optimization techniques are used to optimize the algorithms for the AMD GPU, such as maximizing parallel execution, memory access coalescing, local memory usage and loop unrolling. A difference as compared to the NVIDIA GPU is that, from a programmer point of view, the width of an SIMD unit on the AMD GPU is 64, meaning that 64 threads perform the same instruction concurrently. AMD calls such a group of 64 threads a *wavefront.* To fully utilize the SIMD units, it is therefore important to have the workgroup size be a multiple of the *wavefront* size. Another optimization that does not apply to the NVIDIA GPU is VLIW packing, which is described below.

VLIW packing

Each *Stream Core* on the AMD GPU is programmed with a five-wide (or four-wide, depending on the GPU type) VLIW instruction. Efficient use of GPU hardware requires that the kernel contains enough parallelism to fill all VLIW slots. This can be achieved in two ways, by loop unrolling or by using explicit vector data types. Loop unrolling exposes the underlying parallelism to the compiler, which allows the compiler to pack the instructions into the slots of the VLIW. Using vector data types also exposes the parallelism to the compiler. For example, when performing a computation on data types of *float4*, the compiler can translate this into four *float* instructions which can be executed in parallel by a single VLIW instruction.

The performance results of running the AMD GPU optimized OpenCL algorithms on the AMD GPU are given in Table 5-3 and plotted on the roofline model as shown in Figure 5-6. Below, a few remarks are given explaining some of the results.

- Compared to the DCT implementation on the NVIDIA GPU (which runs at 120% of the peak performance) the performance on the AMD GPU does not seem optimal. A reason for this can be found in analyzing the assembly code of the DCT algorithm targeted at the AMD GPU. Unlike the NVIDIA GPU, no FMA (Fused Multiply Add) instructions are used. Another cause is the performance of the cosine function on the AMD GPU, which will be further discussed in chapter 5.2.4.

- As for the Convolution implementation on the NVIDIA GPU, the AMD GPU implementation also suffers from non-coalesced loads at the border of a block.

- Histogram uses the same implementation structure as used on the NVIDIA GPU (see Figure 5-4). On the AMD GPU it runs at 61% of the peak achievable performance, which is higher as compared to the Histogram implementation on the NVIDIA GPU (12% of the peak). As for the NVIDIA GPU, Histogram on the AMD GPU also suffers from local memory collisions, caused by sharing a partial histogram by multiple threads. Though, the NVIDIA GPU suffers more from the local memory collisions, as will be further explained in chapter 5.2.4.

Algorithm	Processing time (ms)	Performance (GFLOP/s)	Bandwidth (GB/s)	Performance utilization
Binarization	1,108	258,0	121,1	95,77%
DCT	30,295	695,4	4,4	66,61%
Convolution	11,282	865,1	11,9	82,87%
Sum	0,645	581,6	104,0	82,26%
Histogram	0,870	526,1	77,1	60,98%

Table 5-3: Performance results AMD HD5850

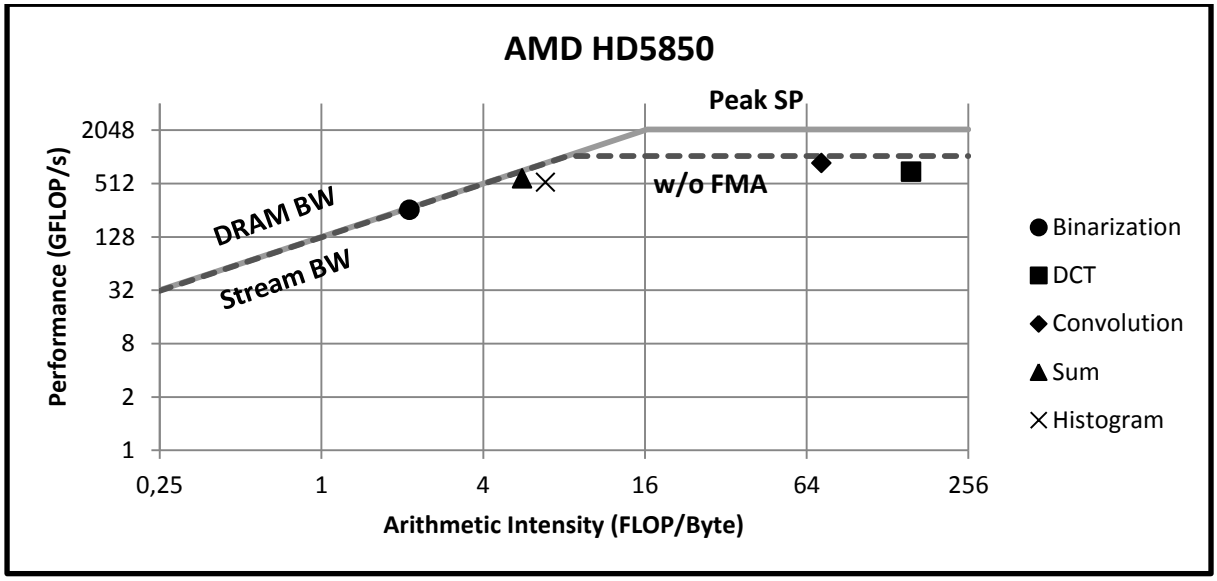


Figure 5-6: Performances on the AMD GPU

5.2.3 INTEL MULTI-CORE CPU

CPU architectures differ a lot from GPU architectures. The available parallelism on a CPU is limited to the width of the vector units inside a core multiplied by a relatively small number of cores. Concepts used on a GPU, such latency hiding by switching between threads and a local scratchpad memory for fast memory accesses do not apply to a CPU. Instead, CPU design was mainly driven by single thread performance and only until a few years more parallelism is being added to the design of the CPU. For OpenCL to reach hardware efficient implementations on a CPU, a different mapping as compared to the GPU implementations is required. Some important optimization techniques will be explained here.

Maximize parallel execution

Since a CPU core is mainly designed for single thread performance, it is best to launch one thread per *Compute Unit* (if hyper-threading is supported, one CPU core will be visible as two *Compute Units* in OpenCL). Launching more threads than available *Compute Units* will result in thread switching, which only introduces additional overheads. Besides the utilization of the *Compute Units*, the vector units inside each *Compute Unit* should be exploited for maximal performance. Several techniques can be used to help the compiler target the vector units. Besides loop unrolling, using explicit vector code in OpenCL will in most cases directly map onto the vector units. To enable this, it is important to have the width of the vector data types be equal to the width of the vector units. On the Intel Core i7-930 for example, the vector width is 128-bits. Therefore, data types such as *uint4* and *float4* should be used to target them.

Exploiting cache hierarchy

A central problem in CPU design has been the growing gap between the latency of a computation and the latency of a memory access. Over the years, performing computations on a CPU became rapidly faster, while memory access times only made small improvements. The cache hierarchy inside a CPU tries to solve this problem, by offering a layered memory system, where each layer that is closer to the processor has a lower memory access latency. For the cache hierarchy to improve the performance of a program, the program should contain spatial and/or temporal locality. The programmer can help exploit the cache hierarchy by using one dimensional arrays and by accessing data arrays in a sequential way.

Other optimization recommendations for the Intel CPU are to avoid local memory use and barrier synchronization instructions, as offered by OpenCL. Since there is no low-latency scratchpad memory available on a CPU, the local memory is mapped onto a region of the global memory. For this reason, using local memory will only introduce unnecessary communication overhead. Issuing a barrier instruction on the Intel CPU causes a lightweight context switch, caused by switching between threads. In case there is only one thread per workgroup, a barrier instruction would be superfluous and could always be omitted.

The performances of running the Intel CPU optimized algorithms on the Intel CPU are summarized in Table 5-4. To see the relative performance as compared to the peak achievable performance, the algorithm's performances are plotted on the roofline model of the Intel CPU, as shown in Figure 5-7.

Algorithm	Processing time (ms)	Performance (GFLOP/s)	Bandwidth (GB/s)	Performance utilization
Binarization	11,578	11,6	11,6	99,36%
DCT	1496,360	23,4	0,1	209,30%
Convolution	156,678	37,1	0,9	331,64%
Sum	4,216	7,0	15,9	135,93%
Histogram	5,683	17,7	11,8	105,43%

Table 5-4: Performance results Intel Core i7-930

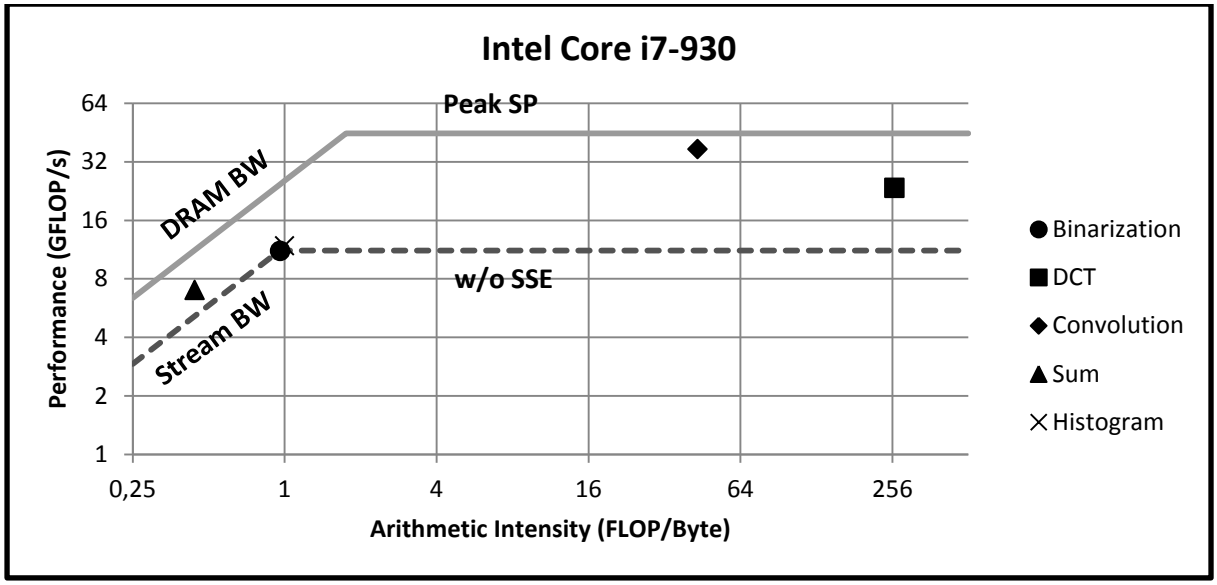


Figure 5-7: Performances on the Intel CPU

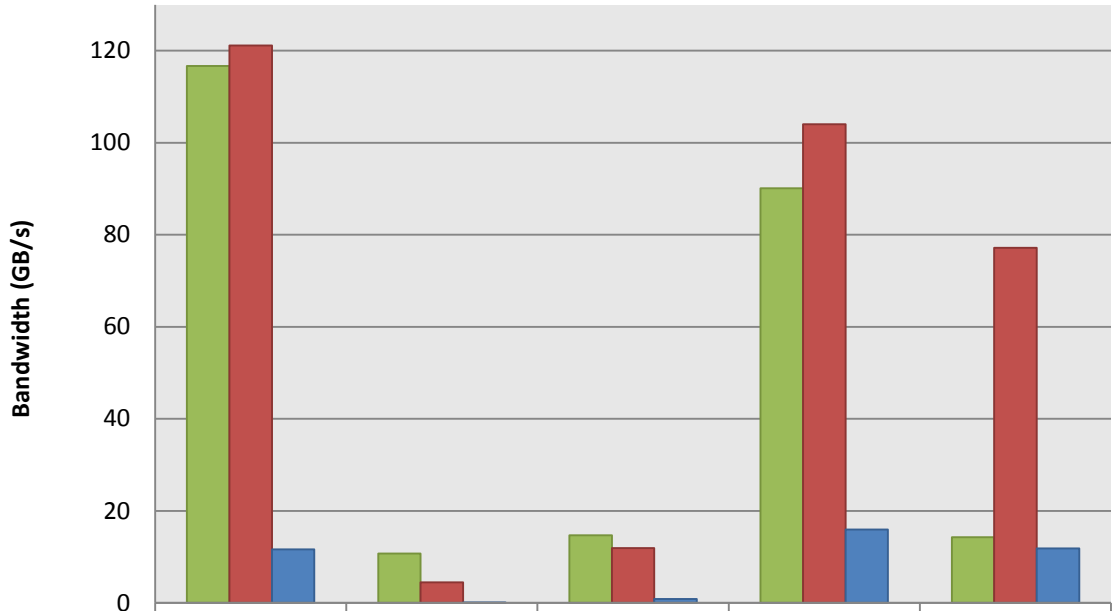
- DCT and Convolution are both between the theoretical roof and the additional ceiling, performing at 209% and 332% of the peak achievable performance, respectively. As both algorithms have a high arithmetic intensity, they are bound by the available compute performance of the device. The reason that the performances are higher than the additional ceiling is because both algorithms use a considerable amount of vector instructions (SSE), which target the devices vector units.

- Surprisingly, the Sum algorithm performs better than the stream bandwidth (136%), even though it is expected to be bound by the bandwidth of the device. This can be clarified by explaining how the stream bandwidth was obtained. The streaming bandwidth test measures the time needed for copying an amount of data from global- to global memory. It involves reading and writing of data. On the other hand, the Sum algorithm mainly reads data from global memory (it only writes a few values). This shows that the read and write bandwidths between a CPU and global memory are unbalanced.

- Histogram also performs slightly better than the practical peak (105%). Here the same reasoning can be used, that it mostly reads data from global memory and only writes a few data elements.

5.2.4 AN ABSOLUTE COMPARISON

Now that the algorithms are optimized for each architecture, a comparison between the architectures can be made by plotting their performances on the same graph. This is done in Figure 5-8. The performances are expressed in bandwidths of the algorithm, obtained by dividing the data transfer size of the algorithm by the processing time of the kernel. What can be observed is that there is a significant performance gap between the CPU and both GPUs. This is mainly caused by a specification mismatch, as a CPU cannot match the high bandwidth and GFLOP numbers as offered by a GPU.



	Binarization	DCT	Convolution	Sum	Histogram
NVIDIA GTX 470	116,7	10,7	14,7	90,1	14,2
AMD HD5850	121,1	4,4	11,9	104,0	77,1
Intel Core i7-930	11,6	0,1	0,9	15,9	11,8

Figure 5-8: Absolute comparison of performances

When comparing the performances on the NVIDIA GPU and the AMD GPU, especially for the DCT and the Histogram algorithms a large performance difference can be observed. Here is a short explanation for these differences.

- As mentioned earlier, the bottleneck in the Histogram calculation is the conflicts that arise in local memory, when multiple threads need to update the same histogram value. A small experiment shows that the NVIDIA GPU has more trouble in dealing with these local memory conflicts as compared to the AMD GPU. For this purpose, two different input arrays are fed to the Histogram implementations on both GPUs, one with random data and one where all elements of the array have the same value (e.g. a white image). With the random data array the chance of a local memory collision is smallest, since there is an even distributivity of values in the histogram. On the other hand, the white image will produce the maximum number of local memory conflicts, since all threads will have to update the same histogram value. Table 5-5 gives the performances. The loss in performance from a minimal to a maximal number of local memory collisions on the AMD GPU is a factor of 1,8, while on the NVIDIA GPU it is a factor of 22,7.

Input (size)	Processing time AMD GPU (ms)	Processing time NVIDIA GPU (ms)
Random data (1024 x 1024)	0,092	0,106
White image (1024 x 1024)	0,165	2,410

Table 5-5: Comparing different inputs for Histogram

- The performance difference of the DCT algorithm on the NVIDIA GPU and the AMD GPU is mainly caused by the latency of the cosine function, which is used a lot in DCT calculation.

Table 5-6 gives the execution times of running an amount of cosine functions on the AMD GPU and the NVIDIA GPU, respectively. The NVIDIA GPU is able to perform the same amount of cosines approximately 5-6 times faster than the AMD GPU.

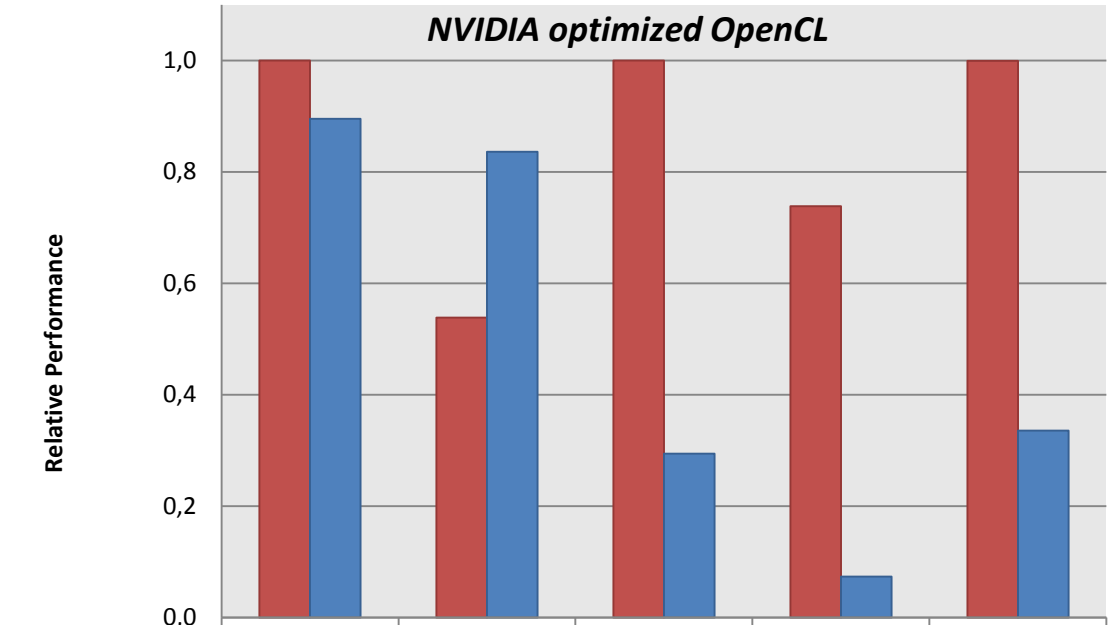
# Cosines	Processing time AMD GPU (ms)	Processing time NVIDIA GPU (ms)
10	0,011	0,009
100	0,071	0,021
1000	0,645	0,144
10000	8,029	1,370
100000	75,798	13,638
1000000	723,515	138,937

Table 5-6: Comparing Cosine performance

5.3 PERFORMANCE PORTABILITY

This chapter will discuss the performance portability of the optimized OpenCL implementations. For this purpose, each implementation - which was optimized for one of the architectures - will be mapped on the other two architectures. Ideally, an OpenCL implementation should reach optimal performance on all target devices. To what extent performance portability is reached will be quantified by comparing the ported performance to the performance of the optimal implementation on the same device, which will here be referred to as the relative performance. For example, if an NVIDIA optimized implementation of an algorithm is executed on the AMD GPU and it runs at half the speed of the AMD optimized implementation of that same algorithm on the AMD GPU, the relative performance would be 0,5.

Figure 5-9 shows what happens if the algorithms - that were optimized for the NVIDIA GPU - are mapped on the other two architectures: the AMD GPU and the Intel CPU.



	Binarization	DCT	Convolution	Sum	Histogram
AMD HD5850	1,000	0,538	1,000	0,738	0,999
Intel Core i7-930	0,895	0,836	0,294	0,074	0,336

Figure 5-9: NVIDIA GPU optimized OpenCL on other architectures

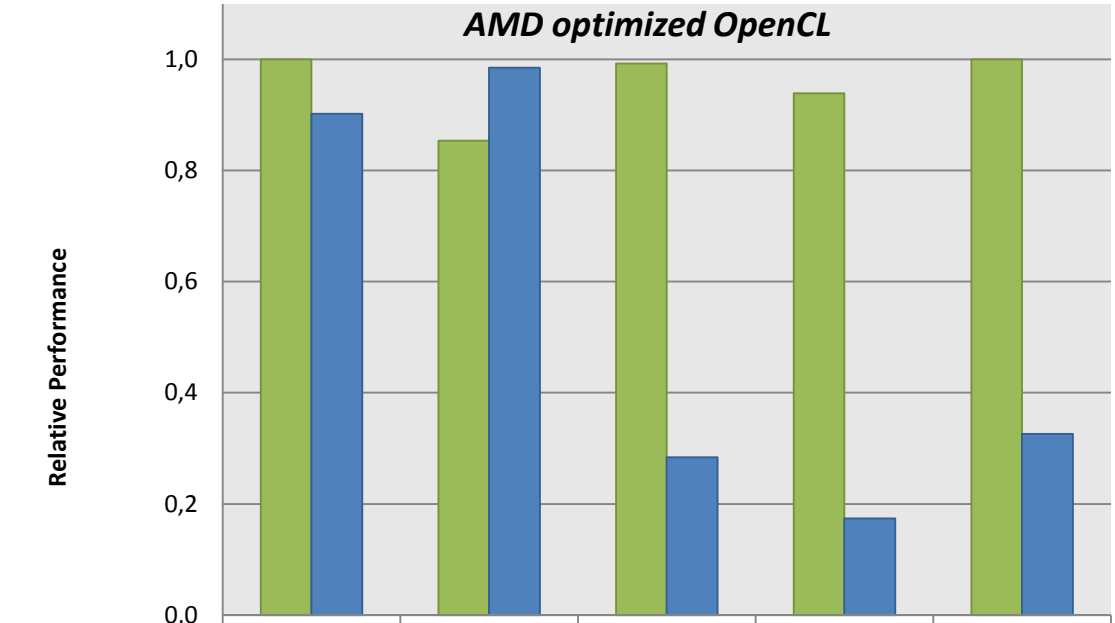
AMD GPU

For Binarization, Convolution and Sum optimal performance portability is reached. This is due to the similarity of the implementations and optimizations applied on both GPUs. Unlike the NVIDIA GPU optimized implementation of DCT, the AMD optimized implementation is vectorized. For this reason, the relative performance of DCT is just over a half. In the Sum implementation as optimized for NVIDIA, the last *warp* is manually unrolled with the assumption that each instruction is executed by 32 threads concurrently. On the AMD GPU however, each instruction is executed by a *wavefront*, which equals 64 threads. This results in a performance degradation on the AMD GPU.

Intel CPU

The relative performances of the NVIDIA GPU optimized implementations on the Intel CPU do not all reach full performance portability. The main reason for this is that in the NVIDIA GPU optimized implementations many threads are launched for reasons of latency hiding, which on the Intel CPU only results in heavy thread switching overheads. Also, the use of local memory and synchronization instructions in the NVIDIA implementations does not map well on the Intel CPU.

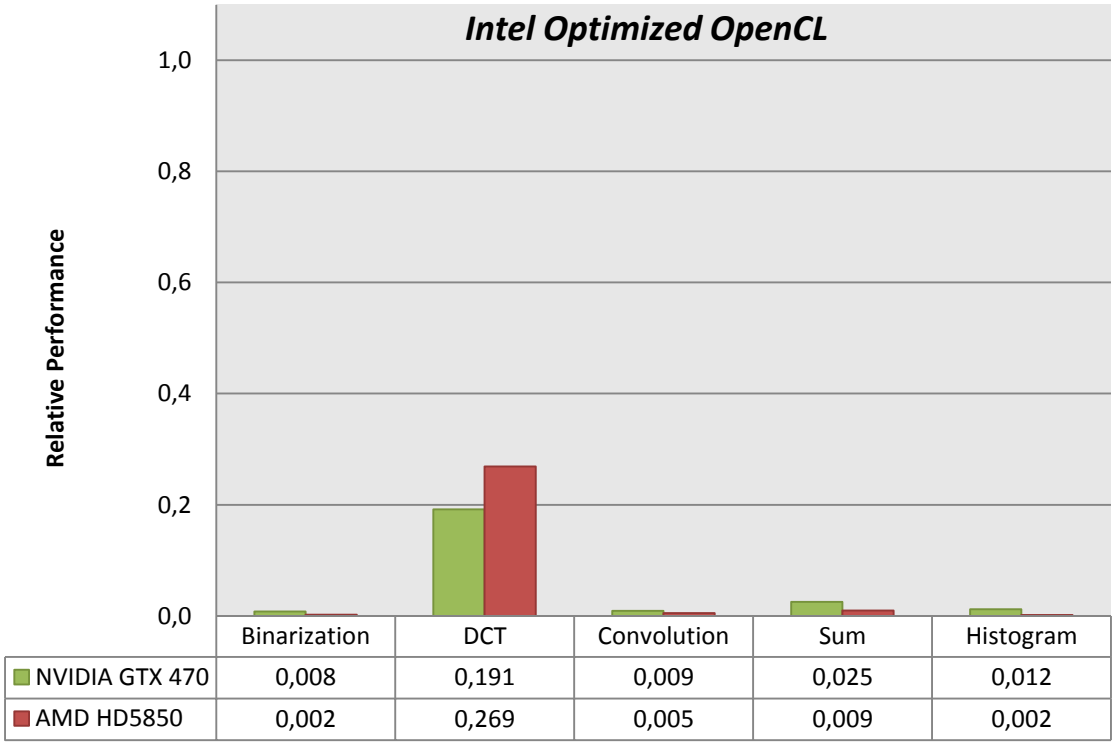
Continuing, the AMD GPU optimized implementations are ported to the other two architectures, for which the relative performances are given in Figure 5-10. The results are similar to porting the NVIDIA GPU optimized implementations to the other architectures. A note has to be made on running the AMD implementation of Sum on the NVIDIA GPU. It unrolls the last 64 threads by assuming that these are executed concurrently. This assumption is not valid for the NVIDIA GPU (on which 32 threads execute concurrently), resulting in non-functional code. Such architecture specific assumptions could be risky when porting code to another architecture. The AMD optimized DCT implementation uses explicit vector data types, which map good on the Intel CPU vector units.



	Binarization	DCT	Convolution	Sum	Histogram
NVIDIA GTX 470	1,000	0,854	0,992	0,939	1,000
Intel Core i7-930	0,902	0,985	0,284	0,174	0,326

Figure 5-10: AMD GPU optimized OpenCL on other architectures

Finally, the Intel optimized OpenCL implementations of the algorithms are ported to both GPUs, as shown in Figure 5-11. The performance portability is poor, worst case only 0,2% of the performance of the optimal implementation is reached. The significant differences between the CPU architecture and the GPU architectures are the main cause for this. For most algorithms, the Intel optimized implementations only launch one thread per *Compute Unit*, which results in heavy under-utilized *Compute Units* on the GPUs. Also, no local memories are used which on the GPUs are paramount for good performance.



	Binarization	DCT	Convolution	Sum	Histogram
NVIDIA GTX 470	0,008	0,191	0,009	0,025	0,012
AMD HD5850	0,002	0,269	0,005	0,009	0,002

Figure 5-11: Intel CPU optimized OpenCL on other architectures

To conclude, reaching performance portability from a single optimized OpenCL implementation is not always guaranteed. Currently, this can be considered a main limitation for developers that want to use OpenCL as a single source code base for targeting multiple devices. Porting code across GPUs can in most cases be done with a minor loss in performance, due to the similarity in mapping OpenCL onto GPU architectures. Care has to be taken regarding architecture specific assumptions, which could lead to non-functional code on another architecture where this assumption is not valid. On the other hand, porting code between a CPU and a GPU could lead to unacceptable performance losses. This is especially visible when porting CPU optimized code to a GPU. The following paragraph will introduce a method that uses code annotations to improve the performance portability of a single OpenCL implementation.

5.4 GENERIC KERNEL DEVELOPMENT

Differences in architectures and optimization techniques that target architectural specifics make it hard to develop a single OpenCL kernel that runs good on multiple target devices. Preferably, an optimization technique applies to all target devices or, when it doesn't, it should at least not affect the performance in a negative way. Unfortunately, this is not true for all optimizations as was

concluded from the results in chapter 5.3. Since code optimizations are essential for good algorithmic performance, a method is required that selects which optimizations are applied to a device and which not. In this work code annotations are used. Two types of code annotations can be used, run-time and compile-time annotations. A run-time annotation will be evaluated at run-time of the kernel code. For example, by adding a parameter to a kernel that can be used inside the kernel to optimize the code for the target device. A disadvantage of run-time annotations is that they will always be evaluated, even when it is not desirable. This will lead to additional overheads. For this reason, compile-time annotations are preferable. Compile-time annotations are evaluated during the compilation of a kernel and will only allow the optimizations to be used if relevant for the target device. The compilation of an OpenCL kernel takes place at run-time of the host code. Just before compilation of the kernel code the target device is known, which enables just-in-time changes in the OpenCL kernel file. *#define* directives will be placed on top of the OpenCL kernel file, which can be used inside the kernels to optimize the code for the specified target device.

Figure 5-12 gives an example of how the code annotations are used in an OpenCL kernel file. Two *#define* directives are placed on top of the kernel file (lines 1-2). *device_type* defines the type of device that will be used to run the kernel upon. It will distinguish between a CPU and a GPU, but it could of course also include other devices types (e.g. the Cell B.E. processor). *elements_per_item* declares how many data elements will be processed per OpenCL work-item, which enables a variable workload per thread for different devices. The variable *elements_per_item* is calculated in the host code by dividing the input data size by the global work-size. Depending on the target device, the variable *stride* will generate a sequential memory access pattern for the CPU or a coalesced memory access pattern for the GPU (lines 12-18). Furthermore, the for-loop inside the kernel (lines 21-22) will iterate as many times as defined by *elements_per_item*. For a CPU, *elements_per_item* will be large since only a few threads are used. On a GPU, thousands of threads will be launched, resulting in a small value for *elements_per_item*. In the case where *elements_per_item* is one, the for-loop is expected to introduce additional overheads. But as the code will be evaluated at compile-time, the compiler will optimize the code by omitting the control of the for-loop. Since the trip count of the loop is known at compile-time time, the compiler can also unroll the loop if possible.

```
 1  #define device_type ...
 2  #define elements_per_item ...
 3
 4  #define CL_DEVICE_TYPE_CPU 2
 5  #define CL_DEVICE_TYPE_GPU 4
 6
 7  __kernel void Algorithm(__global uint* input, __global uint* output)
 8  {
 9    int global_id   = get_global_id(0);
10    int global_size = get_global_size(0);
11
12  #if device_type == CL_DEVICE_TYPE_CPU
13    int index = global_id * pixels_per_item;
14    int stride = 1;
15  #elif device_type == CL_DEVICE_TYPE_GPU
16    int index = global_id;
17    int stride = global_size;
18  #endif
19
20    #pragma unroll
21    for(int n = 0; n < elements_per_item; n++, index += stride)
22      output[index] = function( input[index] );
23  }
```

Figure 5-12: Code annotations inside a kernel

Besides the example as given in Figure 5-12, code annotations can also be used for other purposes. Local memory usage on a GPU is necessary for a hardware efficient implementation. A copy of a block from global memory will be placed in local memory (see Figure 5-13), which is closer to the processing elements resulting in faster memory access times. On a CPU though, the use of local memory is discouraged, since local memory has the same access time as global memory. Code annotations can be used here to declare local memory only when the target device is a GPU.

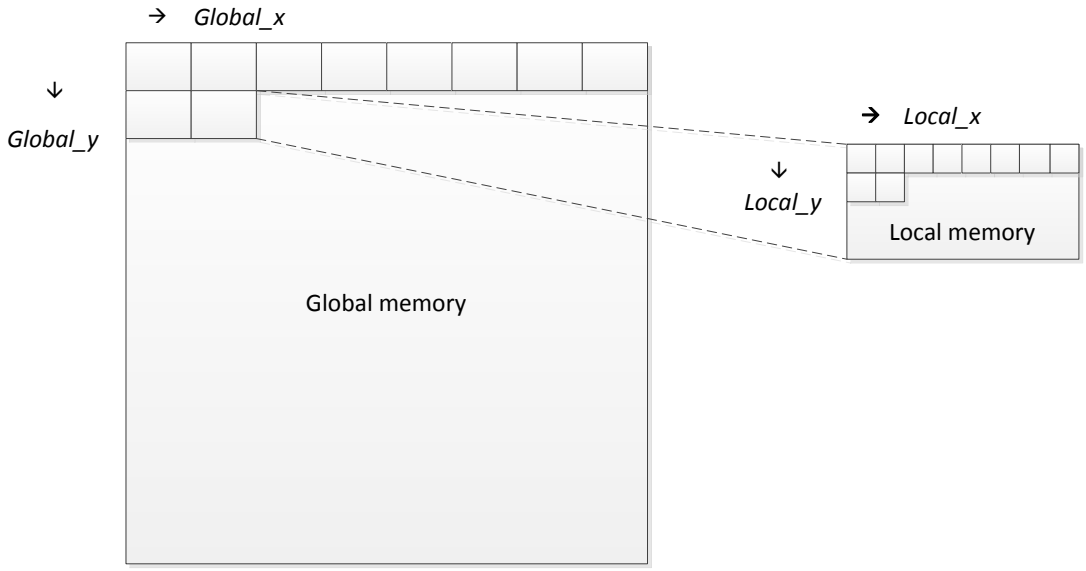


Figure 5-13: Local memory usage

The code listing in Figure 5-14 gives an example of how code annotations can help. It starts by declaring three *#define* directives on top of the OpenCL kernel file (lines 1-3), which are placed there from the host code. The first *#define* directive (*device_type*) specifies the type of the target device, i.e. CPU or GPU. In this example a two dimensional array interpretation is used. The other *#define* directives (*elements_per_itemX*, *elements_per_itemY*) declare the amount of elements that will be processed by a single thread, one for the *X* dimension and one for the *Y* dimension in the input array. Inside the kernel code, the variable *device_type* can be used to declare local memory only when the target device is a GPU (lines 30-35). For a CPU, a thread will read directly from global memory (which will be cached by hardware). Though, a problem occurs when reading data from the input array, where a GPU thread should read from local memory and a CPU thread from global memory. This is solved by performing array index transformations using MACRO definitions (lines 8-22). By using MACROs as indexes for the arrays (line 39), the compiler's pre-processor can replace these by the correct indexes for either a CPU or a GPU.

```
 1  #define device_type ...
 2  #define elements_per_itemX ...
 3  #define elements_per_itemY ...
 4
 5  #define CL_DEVICE_TYPE_CPU 2
 6  #define CL_DEVICE_TYPE_GPU 4
 7
 8  #if (device_type == CL_DEVICE_TYPE_CPU)
 9    INPUT    input
10    INPUT_X  Global_x
11    INPUT_Y  Global_y
12    OUTPUT_X Global_x
13    OUTPUT_Y Global_y
14    WIDTH    global_width
15  #elif (device_type == CL_DEVICE_TYPE_GPU)
16    INPUT    input_local
17    INPUT_X  Local_x
18    INPUT_Y  Local_y
19    OUTPUT_X Thread_x
20    OUTPUT_Y Thread_y
21    WIDTH    local_width
22  #endif
23
24  __kernel void Algorithm(__global uint* input, __global uint* output, int width)
25  {
26    int Thread_x = get_global_id(0);
27    int Thread_y = get_global_id(1);
28    int offset   = Thread_x * elements_per_itemY;
29
30  #if (device_type == CL_DEVICE_TYPE_GPU)
31    int Local_x = get_local_id(0);
32    int Local_y = get_local_id(1);
33    __local input_local[Local_y * local_width + Local_x] = input[Thread_y * width + Thread_x];
34    barrier(CLK_LOCAL_MEM_FENCE);
35  #endif
36
37    for (int Global_y = offset; Global_y < (offset + elements_per_itemY); Global_y++){
38      for (int Global_x = 0; Global_x < elements_per_itemX; Global_x++){
39        output[OUTPUT_Y * global_width + OUTPUT_X] = function(INPUT[INPUT_Y * WIDTH + INPUT_X])
40      }
41    }
42  }
```

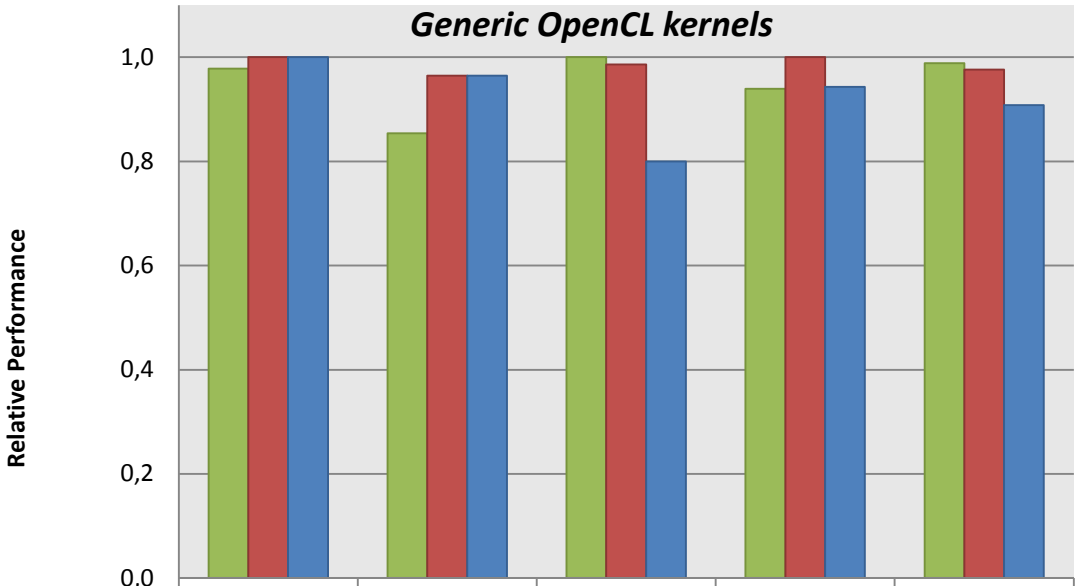
Figure 5-14: Code annotations for local memory usage

After applying the code annotations to the algorithms, the performances are obtained by running the generic kernels on each target device, as shown in Figure 5-15. The performance portability is now significantly improved. Most mappings reach the peak or near-peak achievable performance. A few points can be noted regarding the performances.

- DCT on the NVIDIA GPU reaches 86% of the peak performance, which is caused by the vectorization of DCT. Packing and unpacking of data elements into vector data types introduces additional overheads, against which there is no performance benefit from using vector data types on the NVIDIA GPU.

- Convolution on the Intel CPU reaches 80% of the peak performance. This is caused by the fact that in the Intel optimized implementation of Convolution some loops are manually unrolled, which is not applied to the generic kernel implementation.

- Histogram still uses the optimized histogram calculation as described in [**22**]. Code annotations are used for the CPU to discard local memory usage. The loss in performance is caused by the second kernel of histogram, which sums all the partial histograms into a final histogram (see Figure 5-4). This is done by 256 threads simultaneously, resulting in thread switching overheads on the CPU. But as the time spend in the second kernel is only a fraction of the total processing time, the performance loss is small.



	Binarization	DCT	Convolution	Sum	Histogram
NVIDIA GTX 470	0,978	0,854	1,000	0,939	0,988
AMD HD5850	1,000	0,965	0,986	1,000	0,976
Intel Core i7-930	1,000	0,964	0,800	0,943	0,908

Figure 5-15: Relative performances of generic OpenCL kernels

By applying code annotations during the development of an OpenCL kernel, it is possible to significantly improve the performance portability of the kernel across different devices. A disadvantage of this approach is that it requires the programmer to keep all possible target devices in mind at kernel development, along with device-specific optimization strategies. It requires some additional effort, but saves time when it is decided to target another device. An alternative would be to generate a separate kernel version for each target device, in which optimal performance can be reached for each algorithm-device mapping. This would require developing a whole new kernel when a new device will be targeted, whereas most parts of a kernel can be ported between devices without loss of functionality or performance.

6 RELATED WORK

Since the official release of the specification in December 2008, OpenCL has been exposed to many evaluations. As OpenCL has most of its functionality in common with the CUDA programming standard, CUDA versus OpenCL on NVIDIA GPUs is a frequently recurring comparison in literature [**23**] [**24**] [**25**] [**26**]. In [**23**] a CUDA implementation of a Monte Carlo simulation is translated to OpenCL, showing performance differences varying from 13% to 63%, all in benefit of CUDA. Compiler optimization capabilities are held responsible for the differences. The processing times of CUDA implementations of matrixMul, CP, MRI-Q and MRI-FHD algorithms are about 7,1, 3,2, 6,8, and 8,7 times shorter than their OpenCL implementations, respectively [**24**]. In [**26**] triangular solver (TRSM) and matrix multiplication (GEMM) are used to compare CUDA and OpenCL, resulting in a slight performance difference in favor of CUDA. The CUDA versus OpenCL comparison performed in this work shows similar differences in performance (although less extreme), but distinguishes itself from the other comparisons by giving a detailed explanation of the causes for the performance difference. Here, not only compiler optimizations are investigated, also the kernel launch times are taken into account. Furthermore, a comparison with currently the most recent NVIDIA SDK indicates that the differences in performance between CUDA and OpenCL on NVIDIA GPUs are decreasing.

In terms of portability, an agreement is being made about the usefulness of OpenCL as a single language for different architectures. As is also concluded in this thesis, performance portability of an OpenCL program is hard to achieve [**24**] [**27**]. To solve the problem of performance portability, several papers present auto-tuning techniques [**24**] [**26**] [**27**], in which selected parameters will be tuned for the architecture by performing extensive profiling, sometimes using search heuristics to efficiently explore the design space [**26**]. Besides time and effort needed to find the optimal parameters, some architectural specifics might be overseen or even not suitable for auto-tuning (e.g. finding the optimal memory access pattern).

Porting GPU optimized code to a multi-core CPU has been a hot topic of interest. MCUDA allows CUDA programs to be executed on multi-core CPUs [**28**]. It transforms a threadblock into a serial function and uses loop distribution/fission around synchronization commands to deal with synchronization points inside a loop. In [**29**], [**30**] and [**31**] similar methods are being introduced, but using OpenCL as a starting point. Furthermore, the Ocelot project [**32**] is a translation framework that can map NVIDIAs PTX assembly language to NVIDIA GPUs and AMD GPUs as well as x86 CPUs. The method proposed in this work differs from the above mentioned methods in the sense that it does not start with GPU optimized code to target a CPU. Instead, the OpenCL semantics that map poorly to a CPU are not used in CPU kernel development, which leads to more optimal performances, but requires some additional effort from the programmer.

7 CONCLUSION

This thesis gave an evaluation of OpenCL as a parallel programming framework for targeting current parallel architectures. Two conclusions can be drawn regarding the properties as mentioned in the problem description in chapter 1.1. Furthermore, a conclusion can be drawn from the proposed method of improving the performance portability of an OpenCL program.

- Compared to an architecture's native programming framework, a performance comparison was made against CUDA in targeting different NVIDIA GPUs. A performance difference in the benefit of CUDA was observed in targeting an older NVIDIA GPU (NVIDIA Quadro FX 770). In extreme cases an algorithm described in the CUDA language performs 16% better as compared to an OpenCL description of that same algorithm, but the overall differences are small. The observed differences are mainly caused by two contributors. First, there is a difference in the time that is needed to launch a CUDA kernel and an OpenCL kernel, as caused by the CUDA driver and the OpenCL driver, respectively. The latency of launching a CUDA kernel is smaller than that of an OpenCL kernel. Especially on small kernels that have low processing times, the relatively high kernel launch time of an OpenCL kernel could have a significant impact on the performance. The other factor responsible for the performance difference is found in the different compilers used to translate a CUDA kernel and an OpenCL kernel into an intermediate PTX assembly file. The produced PTX assembly files contain quite some differences, where there is room for improvement in the PTX assembly code as created by the OpenCL compiler. The performance differences can be largely dedicated to the immatureness of the OpenCL driver/compiler structure. When targeting a more recent NVIDIA GPU (NVIDIA GTX 470) in combination with an updated SDK, the performance differences decrease in benefit of OpenCL. Concluding, OpenCL can be considered a decent alternative of CUDA in targeting NVIDIA GPUs.

- Using OpenCL as a single language framework for targeting multiple devices would be an interesting option for developers. It directly involves the portability of an OpenCL program, where the functionality and performance across different devices is a matter of interest. This work showed that the portability of performance as well as functionality of an OpenCL program between devices cannot always be guaranteed. Architectural specifics - along with their optimization strategies - make it hard to develop portable OpenCL code. Partitioning the algorithmic problem size in individual threads must be tuned to the parallelism that is available on the device, which is very different for a CPU and a GPU. Compared to a GPU, a CPU contains only a limited number of cores, constraining the number of threads that should be used for optimal device performance. Furthermore, reaching peak bandwidths for bandwidth bound applications requires different memory access patterns, especially between a CPU and a GPU.

- To improve the performance portability of an OpenCL kernel, code annotations are used to enable different device-dependent optimizations to reside in a single OpenCL implementation. Placing the annotations as *#define* directives in the OpenCL kernel file enables the pre-processor of a compiler to select the right optimizations for the target device. Also, it gives the compiler the opportunity to optimize the code for the target device by, for example, unrolling a loop for a known trip-count. The results are promising, at least

80% of the performance of the optimal implementation for an architecture is reached on all target devices. The benefit of having to develop only a single OpenCL implementation comes with a trade-off in code readability and development effort. If maximal performance is required, different kernel versions for each target platform need to be created.

7.1 FUTURE WORK

In the context of this research project, several areas that were not treated here might be interesting to further look into. Furthermore, resulting from the method introduced to improve the performance portability of an OpenCL program, extensions are possible.

- To extend the evaluation of OpenCL as a language alternative for current programming standards, a comparison against other native programming standards could be performed. For targeting multi-core CPUs, a comparison against OpenMP or Intel ArBB would be of interest. As AMD announced there will be no future support for their former standard of programming AMD GPUs (AMD CAL), it is expected that OpenCL will be the future standard for programming AMD GPUs.

- The proposed method for improving the performance portability of an OpenCL program could be extended by taking other architectural specifics into account. By further generalizing a kernel it can be better tuned for the target device. Possible extensions could include the local memory size, the amount of cores and the SIMD width as offered by a device. Most of these parameters can be obtained via simple OpenCL API calls. To relieve the effort that is needed by the programmer, possibilities to automate this process might be an interesting topic of research.

- As the list of vendors supporting OpenCL is growing, targeting other devices from OpenCL would be interesting. Upcoming heterogeneous architectures - which combine a CPU and a GPU on a single chip - as well as embedded processor architectures, might be very suitable as target devices for OpenCL.

BIBLIOGRAPHY

[1] NVIDIA Corporation, CUDA: Scalable Parallel Programming for High-Performance Scientific Computing, 2008.

[2] AMD Corporation, "ATI Stream Computing Programming Guide," March 2010.

[3] The OpenMP API Specification for Parallel Programming. http://openmp.org/wp/.

[4] Chris J. Newburn et al, "Intel's Array Building Blocks: A Retargetable, Dynamic Compiler and Embedded Language," *International Symposium on Code Generation and Optimization*, 2011.

[5] IBM Corporation. IBM SDK for Multicore Acceleration Version 3.1.

[6] Khronos OpenCL Working Group, "The OpenCL Specification, version 1.1, revision 44," Khronos Group, June 2011.

[7] Khronos Group. (2011, June) http://www.khronos.org/opencl/adopters/.

[8] NVIDIA Corporation, NVIDIA's Next Generation CUDA Compute Architecture: Fermi, 2009.

[9] NVIDIA Corporation, NVIDIA Tesla: A Unified Graphics and Computing Architecture, 2008.

[10] AMD, AMD Accelerated Parallel Processing: OpenCL Programming Guide, May 2011.

[11] http://en.wikipedia.org/wiki/X86.

[12] K. Asanovic et al, "A View of the Parallel Computing Landscape," *Communications of the ACM*, vol. 52, no. 10, October 2009.

[13] Samuel Webb Williams, "Auto-tuning Performance on Multicore Computers," University of California, Berkeley, 2008.

[14] Cedric Nugteren, Gert-jan van den Braak, Henk Corporaal, and Bart Mesman, "High performance predictable histogramming on GPUs: exploring and evaluating algorithm trade-offs," *GPGPU-4, Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, 2011.

[15] Cedric Nugteren, Henk Corporaal, and Bart Mesman, "Skeleton-based Automatic Parallelization of Image Processing Algorithms for GPUs," *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XI)*, 2011.

[16] C. Nugteren, "Classification of Image Processing and Vision Algorithms for Parallel Programming," Eindhoven University of Technology, 2011.

[17] http://en.wikipedia.org/wiki/Discrete_cosine_transform.

[18] J. van der Sanden, "Evaluating the Performance and Portability of OpenCL (preparation project)," Eindhoven University of Technology, March 2011.

[19] NVIDIA Corporation, PTX: Parallel Thread Execution, ISA Version 2.2, August 2010.

[20] NVIDIA Corporation, NVIDIA OpenCL Best Practices Guide, April 2010.

[21] Mark Harris, Optimizing Parallel Reduction in CUDA, 2008.

[22] Victor Podlozhnyuk, Histogram Calculation in CUDA, 2007.

[23] K. Karimi, N.G. Dickson, and F. Hamze, "A Performance Comparison of CUDA and OpenCL," 2010.

[24] K. Komatsu et al, "Evaluating Performance and Portability of OpenCL Programs," *iWAPT (International Workshop on Automatic Performance Tuning)*, June 2010.

[25] Matt Harvey, "Experiences porting from CUDA to OpenCL.," Imperial College London CBBL IMIM, December 2009.

[26] P. Du et al, "From CUDA to OpenCL: towards a performance-portable solution for multi-platform GPU programming," Electrical Engineering and Computer Science Department, University of Tennessee, Technical Report CS-10-656, 2010.

[27] S.Rul, H. Vandierendonck, J. D'Haene, and K. De Bosschere, "An Experimental Study on Performance Portability of OpenCL Kernels," *Symposium on Application Accelerators in High Performance Computing (SAAHPC)*, 2010.

[28] J. Stratton, S. Stone, and W. Hwu, "MCUDA: An Efficient Implementation of CUDA Kernels for Multi-Core CPUs," *Languages and Compilers for Parallel Computing*, vol. 5335/2008, pp. 16-30, 2008.

[29] J. Lee et al, "An OpenCL Framework for Heterogeneous Multicores with Local Memory," *Proceedings of the 19th international conference on Parallel architectures and compilation techniques (PACT '10)* , 2010.

[30] K. Daloukas, C. Antonopoulos, and N. Bellas, "GLOpenCL: OpenCL Support on Hardware- and Software-Managed Cache Multicores," *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC '11)*, 2011.

[31] AMD corporation, "Twin Peaks: A Software Platform for Heterogeneous Computing on General-Purpose and Graphics Processors," *Proceedings of the 19th international conference on Parallel architectures and compilation techniques (PACT '10)*, pp. 205-216, 2010.

[32] G. Diamos, A. Kerr, and S. Yalamanchili, "Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems," *Proceedings of the 19th international*

conference on Parallel architectures and compilation techniques (PACT '10), 2010.

[33] R. Domínguez, D. Schaa, and D. Kaeli, "Caracal: dynamic translation of runtime environments for GPUs," *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units (GPGPU-4)*, 2011.

APPENDIX A. PERFORMANCE EVALUATION OPENCL VS CUDA

This chapter discusses the approach and results obtained from the performance evaluation between OpenCL and CUDA programs, benchmarked on an NVIDIA GPU based on the G8x architecture.

A.1 Image processing algorithms

To analyze the performance of OpenCL and CUDA programs, some algorithms which are typically used in image processing will be used to benchmark the performances. As these algorithms have high numbers of exploitable parallelism, they are very suitable to be implemented on a GPU. Table 7 summarizes the used algorithms, indicating the classification of the algorithms according to [7] and the possibility of data-reuse. The existence of data-reuse is important since it enables the use of local/shared memory, which can have a significant improvement on the performance. All algorithms take a picture as input and transform it to either a picture of the same size, a vector or just a value. The algorithms are implemented in CUDA and optimized for performance on an NVIDIA GPU. Optimizations include memory coalescing, shared memory usage, loop unrolling and the usage of special function units [8]. The reduction to scalar (*Sum*, *Max*) and the reduction to vector (*Histogram*) algorithms are optimized using a reduction tree [9] [10].

Algorithm	Classification	Data-reuse
Blur	Neighborhood to pixel	Yes
Erode	Neighborhood to pixel	Yes
Gradientx	Neighborhood to pixel	Yes
Dilate	Neighborhood to pixel	Yes
Sobel	Neighborhood to pixel	Yes
Transpose	Pixel to pixel	No
Copy	Pixel to pixel	No
Binarization	Pixel to pixel	No
Sum	Reduction to scalar	Yes
Max	Reduction to scalar	Yes
Histogram	Reduction to vector	Yes

Table 7. Image processing algorithms

A.2 Translating CUDA kernel to OpenCL kernel

To make a fair comparison between CUDA and OpenCL, the CUDA kernels are ported to OpenCL kernels in a straightforward way. Translating a CUDA kernel to an OpenCL kernel only involves changing the keywords, which have different names under CUDA and OpenCL but share the same semantics. Table 8 summarizes the differences in keywords used by CUDA and OpenCL. The OpenCL keywords *get_global_id()* and *get_global_size()*, used for retrieving the id of a work-item in a global space and querying the size of a global space, respectively, are not supported in CUDA. Instead, in CUDA these have to be calculated using other keywords. For example, to get the global id of a thread one can use the *blockIdx*, *blockDim* and *threadIdx* keywords to calculate this manually.

CUDA	OpenCL
__shared__	__local
__device__	__global
__global__	__kernel
threadIdx	get_local_id()
blockIdx	get_group_id()
gridDim	get_num_groups()
blockDim	get_local_size()
__syncthreads()	Barrier()
Calculate manually	get_global_id()
Calculate manually	get_global_size()

Table 8. Different keywords between CUDA and OpenCL

As the translation from a CUDA kernel to an OpenCL kernel is a one-to-one mapping where no functional elements of the algorithm or implementation are changed, it enables a fair comparison between the two languages when evaluating the performances. Besides the translation of the kernels, in OpenCL some more work is required to be able to execute a kernel on a compute device. Due to the generality of OpenCL, the programmer is required to first explore the system for OpenCL supported devices and then choose the device which will be used to execute the kernels upon. Furthermore, OpenCL requires the declaration of a context used by the OpenCL runtime for managing objects such as command-queues and memory-, program- and kernel objects.

A.3 Benchmark results

Both implementations (CUDA and OpenCL) of the image processing algorithms as discussed in chapter A.1 are benchmarked on the NVIDIA device which is part of the system as specified in Table 9. To see the impact of the size of a kernel (i.e. the number of threads/work-items used to execute the kernel), different input sizes are fed to the image processing algorithms.

CPU	Intel Core 2 Duo T9600 @ 2,80 GHz
GPU (Compute Capability)	NVIDIA Quadro FX 770M, 512 MB (1.1)
OS	Windows 7 Enterprise, build 7600
Platform	NVIDIA CUDA 3.2.1 (OpenCL 1.0 support)
Driver	NVIDIA driver version 260.99
SDK Revision	7027912

Table 9. System specification

In this work, only the processing times of the kernels running on the GPU are measured in evaluating the performance. Figure 0-1 shows the results after performing the benchmarks of both the CUDA- as the OpenCL implementation. The horizontal axis indicates the image processing algorithms. For each algorithm four input sizes are benchmarked (indicated on the right). The vertical axis indicates the performance difference between OpenCL and CUDA, normalized to CUDA processing times. The graph can be read as follows: values above 1,0 indicate that OpenCL runs slower compared to CUDA, while values below 1,0 indicate that OpenCL runs faster than CUDA (value 1,0 means they run equally fast).

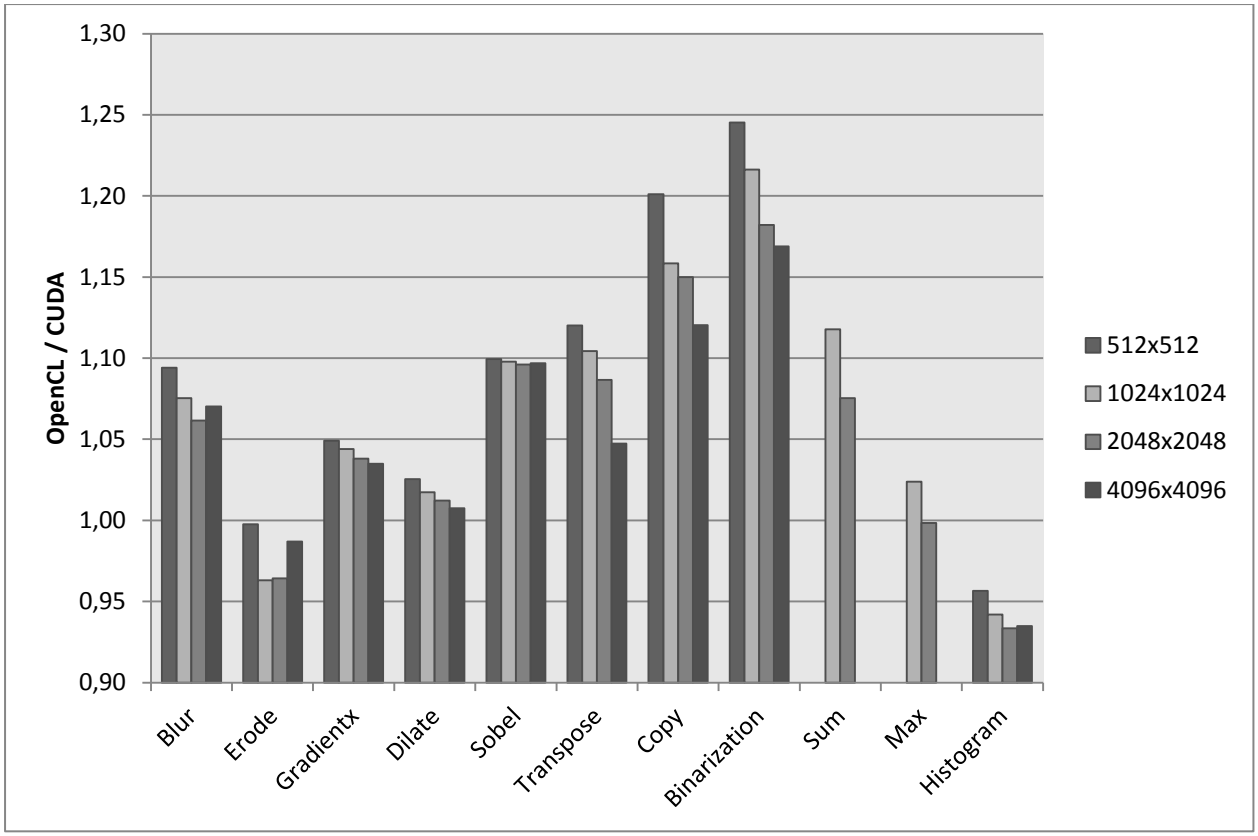


Figure 0-1. Performance comparison between OpenCL and CUDA

The results from Figure 0-1 indicate that there can be quite some performance difference between a CUDA program and an OpenCL program executed on the same device. In most cases OpenCL is slower than CUDA (up to 25%), and in some cases OpenCL is faster than CUDA (up to 7%). On average the OpenCL programs run 6,3% slower than the CUDA programs. What immediately strikes from Figure 0-1 is that especially for small input sizes OpenCL seems to perform poorer than for larger input sizes. This observation is caused by a difference in kernel launch times between CUDA and OpenCL, which will be further discussed in chapter A.4.

What is remarkable is that there can be such a large performance difference between CUDA and OpenCL programs, especially when considering that when porting a CUDA kernel to an OpenCL kernel, the functionality of the kernel is not changed (only keywords are changed). Thus the same (high-level) code is run on the same architecture. In the following paragraphs some more insight into the performance difference between CUDA and OpenCL will be gained by analyzing the kernel launch times of the kernels and the compilation flow performed by CUDA and OpenCL.

A.4 Kernel launch

The processing time of a kernel is build up of two components: the kernel launch time and the time spend on executing the code inside the kernel. According to [13] the kernel launch time depends on the number of threadblocks/work-groups, the number of kernel arguments and to a lesser extent on the number of threads/work-items in a threadblock/work-group. In the case of the image processing algorithms, most kernels launch more threadblocks/work-groups when the input size grows, while the number of threads/work-items per threadblock/work-group and the number of arguments per

kernel remain the same. A method to measure the kernel launch time is to launch an empty kernel. This was done for the image processing algorithms and the results of the kernel launch times for the *Copy* algorithm is shown in Figure 0-2. The kernel launch times are normalized to the kernel launch time of the CUDA kernel. As can be seen, for small input sizes, OpenCL has significantly higher kernel launch times compared to CUDA (up to 28%). For larger input sizes the difference saturates to negligible values.

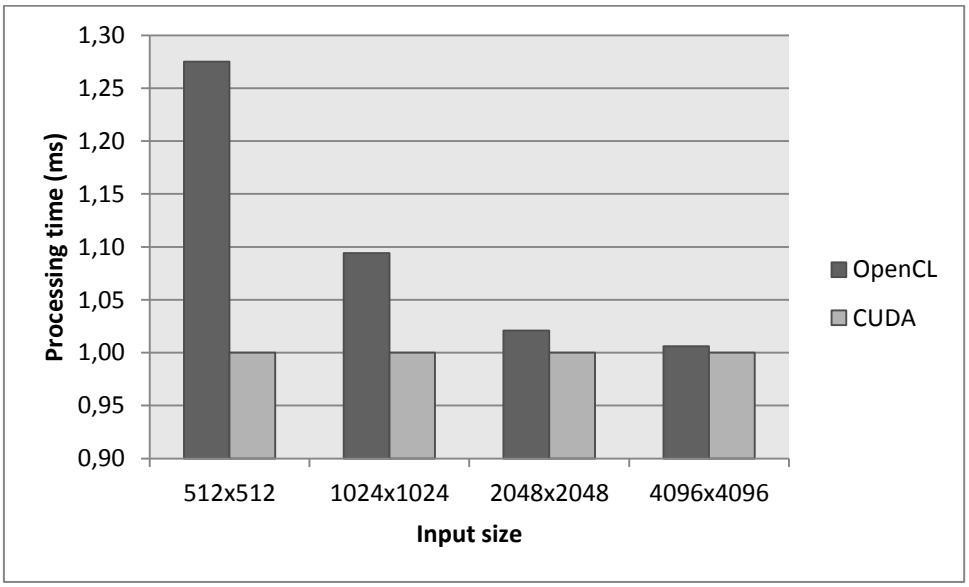


Figure 0-2. Kernel launch comparison for the Copy algorithm

From this observation it can be concluded that OpenCL has a slightly higher overhead when launching a kernel, which is especially obvious when launching small kernels. It must be stated that the kernel launch time is only a small fraction of the total processing time of a kernel (less than 5%), and the influence on the performance is therefore limited. When the kernel launch times are subtracted from the total processing times of the kernels, the raw processing times of the kernels are obtained. From these raw processing times, a new comparison between CUDA and OpenCL is performed which is shown in Figure 0-3. When comparing this graph with the earlier presented graph in Figure 0-1, it can be seen that especially for small input sizes the performance of OpenCL compared to CUDA increases (the high peaks in the graph disappear).

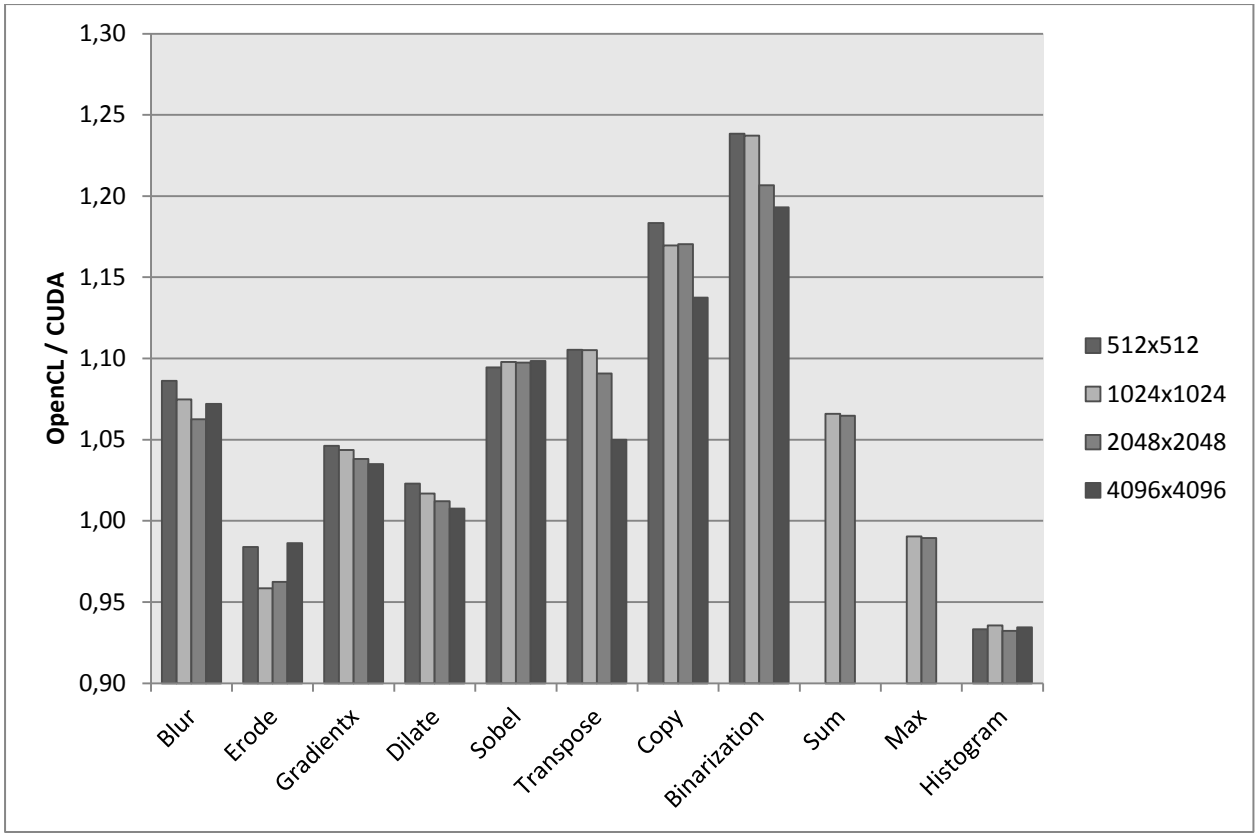


Figure 0-3. Performance difference CUDA and OpenCL (kernel launch corrected)

The analyses of the kernel launch times proves that OpenCL has a slightly higher kernel launch overhead compared to CUDA, which is particularly visible on small input sizes. As there is still quite some performance difference after omitting the kernel launch times from the total processing times (on average OpenCL still performs 6,0% slower), this is only a small part contributing to the total performance difference between CUDA and OpenCL.

A.5 Compilation

To explain the differences in performance observed by running the benchmark algorithms both in CUDA and OpenCL on the same NVIDIA device, a more detailed investigation into the compilation flow of both languages is required. The compilation flow of both languages is depicted in Figure 0-4, which is a simplified abstraction of the actual more complicated compilation flow.

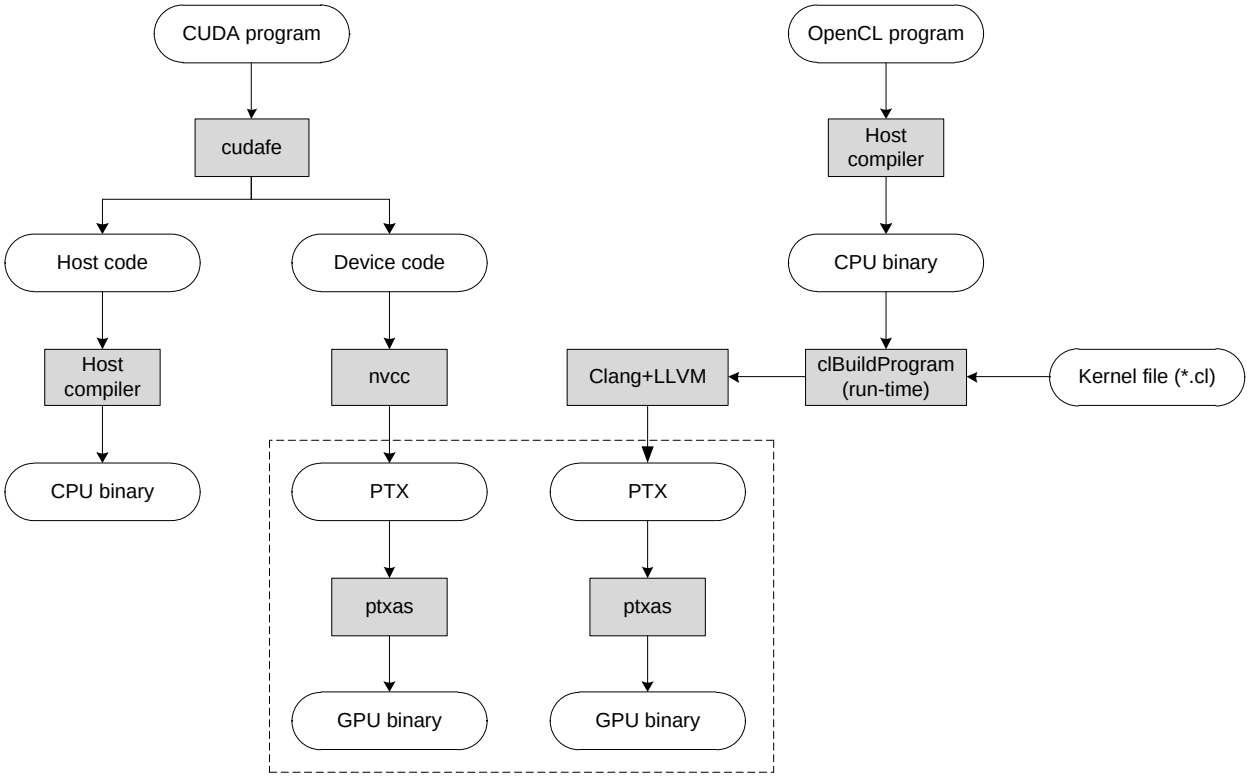


Figure 0-4. CUDA and OpenCL compilation flow

A CUDA program is first split by the CUDA frontend compiler (*cudafe*) into host- and device-code. The host code is compiled into a CPU binary by the compiler which is available on that platform. The device code is compiled by *nvcc* into a PTX file. The PTX code represents an intermediate assembly language, which is still hardware independent and can be run on any CUDA enabled GPU [11]. The next compiler, *ptxas*, performs a hardware specific compilation of the PTX code into a GPU binary, which can be run on a specific NVIDIA GPU.

Since OpenCL supports different architectures from different vendors, an OpenCL program must be compiled for the device which is specified in the program itself. At compile-time, it is unknown for which architecture(s) the code must be compiled for. For this reason, OpenCL uses a run-time compilation flow, where a kernel file (*.cl) is compiled during run-time of the program, by invoking the appropriate API call. As a result, the compilation time is always included in the execution time of the complete OpenCL program. Which compiler is then used depends on the target device. NVIDIA uses a combination of *Clang* and *LLVM* [12] to compile a kernel file into a PTX file. As can be seen in Figure 0-4, from there on the compilation flow is equal to the last step of the CUDA compilation flow.

A.6 PTX Assembly

Since NVIDIA uses different compilers for CUDA and OpenCL code, the interesting parts in the compilation flow to look at are the intermediate PTX files produced by the CUDA compiler and the OpenCL compiler. Any difference in the PTX codes can be dedicated to the different compilers used, and could have an impact on the performance. Figure 0-1 showed that the *Binarization* algorithm

caused the largest performance difference between OpenCL and CUDA. The translation of the high level code of the *Binarization* kernel into the intermediate PTX assembly language is shown in Figure 0-5, both for the CUDA compiler as the OpenCL compiler. In between the CUDA PTX and the OpenCL PTX the original high-level code is visible, which in this case is the CUDA code (the high-level OpenCL code is similar, only other keywords are used). The marked instructions are the same across the OpenCL and the CUDA PTX. The remaining instructions all differ. As can be seen the computation of a thread/work-item identification (the middle part) is done quite different by OpenCL and CUDA, while the actual binarization computation (the bottom part) is done quite similar.

CUDA PTX		Binarization Kernel	OpenCL PTX	
.entry _Z16GPU_BinarizationPjS_jj (.param .u32 __cudaparm_imageA, .param .u32 __cudaparm_imageD, .param .u32 __cudaparm_image_w, .param .u32_image_h)		__global__ void GPU_Binarization(unsigned int* imageA, unsigned int* imageD, unsigned int image_w, unsigned int image_h)	.entry GPU_Binarization(.param .u32 .ptr .global .align 4 GPU_param_0, .param .u32 .ptr .global .align 4 GPU_param_1, .param .u32 GPU_param_2, .param .u32 GPU_param_3	
mov.u16	%rh1, %ctaid.x;		mov.u32	%r1, %ctaid.y;
mul.wide.u16	%r1, %rh1, 32;		mov.u32	%r2, %envreg1;
mov.u16	%rh2, %ctaid.y;		add.s32	%r3, %r1, %r2;
mul.wide.u16	%r2, %rh2, 16;		shl.b32	%r4, %r3, 4;
cvt.u32.u16	%r3, %tid.x;		mov.u32	%r5, %tid.y;
add.u32	%r4, %r3, %r1;		mov.u32	%r6, %ctaid.x;
cvt.u32.u16	%r5, %tid.y;	unsigned int w = blockIdx.y*BLOCKDIM_P2P_Y + threadIdx.y;	mov.u32	%r7, %envreg0;
add.u32	%r6, %r5, %r2;	unsigned int h = blockIdx.x*BLOCKDIM_P2P_X + threadIdx.x;	add.s32	%r8, %r6, %r7;
ld.param.u32	%r7, [__cudaparm_image_h];	unsigned int p = h+w*image_h;	add.s32	%r9, %r4, %r5;
mul.lo.u32	%r8, %r7, %r6;		mov.u32	%r10, %tid.x;
add.u32	%r9, %r4, %r8;		ld.param.u32	%r11, [GPU_param_3];
mul.lo.u32	%r10, %r9, 4;		mad.lo.s32	%r12, %r9, %r11, %r10;
			shl.b32	%r13, %r8, 5;
			add.s32	%r14, %r12, %r13;
			shl.b32	%r15, %r14, 2;
mov.u32	%r11, 255;			
mov.u32	%r12, 0;			
ld.param.u32	%r13, [__cudaparm_imageA];		ld.param.u32	%r16, [GPU_param_0];
add.u32	%r14, %r13, %r10;		add.s32	%r17, %r16, %r15;
ld.global.u32	%r15, [%r14+0];		ld.global.u32	%r18, [%r17];
mov.u32	%r16, 170;			
setp.gt.u32	%p1, %r15, %r16;	imageD[p] = (imageA[p] > 170) * 255;	setp.gt.u32	%p1, %r18, 170;
selp.u32	%r17, %r11, %r12, %p1;		selp.u32	%r19, 1, 0, %p1;
ld.param.u32	%r18, [__cudaparm_imageD];		ld.param.u32	%r20, [GPU_param_1];
			mul.lo.s32	%r21, %r19, 255;
add.u32	%r19, %r18, %r10;		add.s32	%r22, %r20, %r15;
st.global.u32	[%r19+0], %r17;		st.global.u32	[%r22], %r21;
exit;			ret;	

Figure 0-5. CUDA PTX vs OpenCL PTX of the *Binarization* kernel

To find the consequences of the differences in the PTX code, it would be useful to adapt the OpenCL PTX file manually and then run it in the OpenCL program. OpenCL provides support for such kind of execution, where instead of loading an OpenCL source file (*.cl), one can directly load a PTX file (*.ptx) and run it in the OpenCL program. This enables step by step adaptations in the OpenCL PTX code until it becomes similar to the CUDA PTX, revealing the causes for the performance difference. Table 10 gives the changes that are performed in the OpenCL PTX and the consequences on the performance. The last entry gives the processing time of the CUDA kernel, which is used as baseline for comparing the performance of the OpenCL PTX.

#	PTX optimization	Processing time (ms)	OpenCL/CUDA
1	Original OpenCL PTX	0,264	1,245
2	Removed offset instructions	0,252	1,189
3	Removed redundant *mul*. instruction	0,239	1,127
4	Replaced *mul.lo.u32* by *mul.wide.u16* (2x)	0,223	1,052
5	CUDA PTX (run in OpenCL program)	0,223	1,052
-	CUDA PTX (run in CUDA program)	0,212	-

Table 10. Performance improvements by changing OpenCL PTX

In OpenCL it is possible to give an index space offset, which is zero by default. This offset seems to be always translated into PTX instructions, even if it is not defined. This results in four extra instructions in the OpenCL PTX (two *mov.* and two *add.* instructions), which are not there in the CUDA PTX. Removing these does not change the functionality of the kernel, but has a positive impact on the performance of the OpenCL PTX. Furthermore, at the end of the OpenCL PTX, there is a redundant *mul.* instruction, which can be removed by changing the *selp.* instruction (two instructions above it) into:

```
selp.u32   %r19, 255, 0, %p1;
```

Another remarkable difference between the CUDA PTX and the OpenCL PTX is that the CUDA PTX does some instructions on 16-bit values, whereas the OpenCL PTX only on 32-bit values. The reason behind this is that the CUDA PTX is defined in the PTX ISA version 1.4 [15], which still uses some special registers of 16 bits wide. On the other hand, the OpenCL PTX is defined in the PTX ISA version 2.2, which defines the special registers as 32-bit wide. When changing the OpenCL PTX instructions to be like the CUDA PTX instructions, it appears that changing the *mul.wide.u16* instruction into a *mul.lo.u32* is decisive in terms of performance. This implies that the NVIDIA hardware can do a multiply on 16 bits faster than a multiply on 32 bits. The processing time of the OpenCL PTX is now equal to the processing time of running the CUDA PTX in the OpenCL program. Figure 0-6 visualizes the impact of the changes in the OpenCL PTX code on the performance difference between CUDA and OpenCL. Each number on the horizontal axes corresponds to the change as described in Table 10.

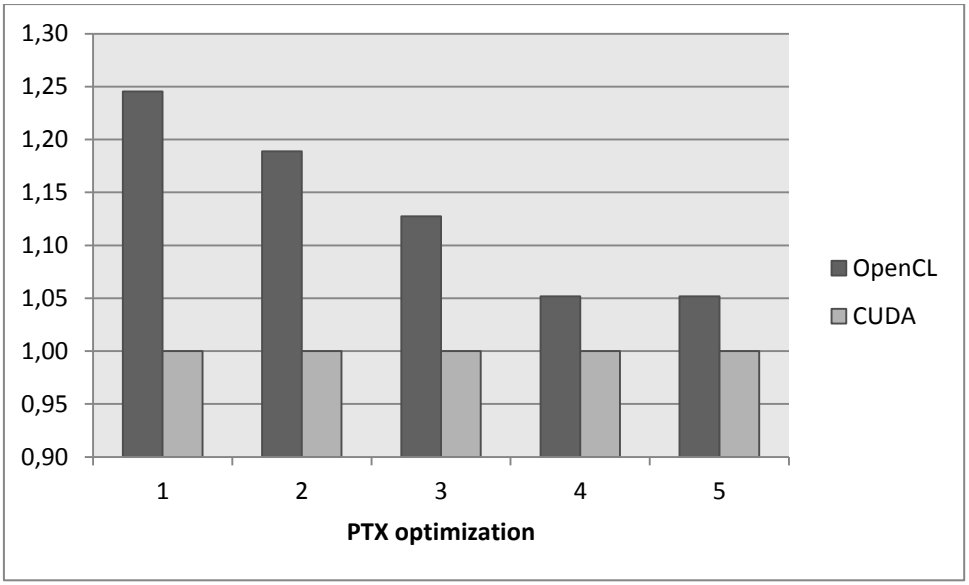


Figure 0-6. Performance differences OpenCL vs CUDA by changing OpenCL PTX

The remaining performance difference between OpenCL and CUDA is caused by the difference in kernel launch times, as was discussed in chapter A.4. When subtracting the difference in kernel launch times between CUDA and OpenCL from the processing time of the final OpenCL PTX, the processing times of the OpenCL and CUDA kernels become equal, meaning there is no performance difference anymore.

This section showed that the compilers used for translating a CUDA kernel and an OpenCL kernel into intermediate PTX assembly code, produce quite different results. This is partly caused by targeting different PTX ISA versions, but is also caused by the fact that the OpenCL compiler is still immature, leaving room for further improvements. As in this work only one kernel is explored in detail, the results are not representative for all algorithms. There might be cases where the OpenCL compiler produces better code than the CUDA compiler. In fact, this is true for the *Erode* and *Histogram* kernels, which perform better in OpenCL than in CUDA (see Figure 0-1).