CHALMERS

General purpose computing on graphics processing units using OpenCL

Motion detection using NVIDIA Fermi and the OpenCL programming framework

Master of Science thesis in the Integrated Electronic System Design programme

MATS JOHANSSON
OSCAR WINTER

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, June 2010

General purpose computing on graphics processing units using OpenCL
Motion detection using NVIDIA Fermi and the OpenCL programming framework

MATS JOHANSSON, mats@gluteus.se
OSCAR WINTER, oscar@gluteus.se

Examiner: ULF ASSARSSON

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden June 2010

Abstract

General-Purpose computing using Graphics Processing Units (GPGPU) has been an area of active research for many years. During 2009 and 2010 much has happened in the GPGPU research field with the release of the Open Computing Language (OpenCL) programming framework and the new NVIDIA Fermi Graphics Processing Unit (GPU) architecture.

This thesis explores the hardware architectures of three GPUs and how well they support general computations; the NVIDIA Geforce 8800 GTS (the G80 architecture) from 2006, the AMD Radeon 4870 (the RV700 architecture) from 2008 and the NVIDIA Geforce GTX 480 (the Fermi architecture) from 2010. Special concern is given to the new Fermi architecture and the GPGPU related improvements implemented in this architecture. The Lukas-Kanade algorithm for optical flow estimation has been implemented in OpenCL to evaluate the framework and the impact of several different parallel application optimizations.

The RV700 architecture is not well suited for GPGPU. The performance of the G80 architecture is very good taking its relative age into account. However, much effort must be spent optimizing a parallel application for the G80 before full performance is obtained, a task that can be quite tedious. Fermi excels in all aspects of GPGPU programming. Fermi's performance is much higher than that of the RV700 and the G80 architectures and its new memory hierarchy makes GPGPU programming easier than ever before.

OpenCL is a stable and competent framework well suited for any GPGPU project that would benefit from the increased flexibility of software and hardware platform independence. However, if performance is more important than flexibility, NVIDIA's Compute Unified Device Architecture (CUDA) or AMD's ATI Stream might be better alternatives.

Sammanfattning

Generella beräkningar med hjälp av grafikprocessorer (General-Purpose computation using Graphics Processing Units, GPGPU) har varit ett aktivt forskningsområde under många år. Stora framsteg har gjorts under 2009 och 2010 i och med lanseringen av programmeringsramverket Open Computing Language (OpenCL) och NVIDIAs nya GPU-arkitektur Fermi.

Denna tes utforskar hårdvaruarkitekturen hos tre grafikprocessorer och hur väl de är anpassade för generella beräkningar; NVIDIA Geforce 8800 (G80-arkitekturen) utgivet 2006, AMD Radeon 4870 (RV700-arkitekturen) utgivet 2008 och NVIDIA Geforce GTX 480 (Fermi-arkitekturen) utgivet 2010. Stort fokus läggs på Fermi och de GPGPU-relaterade förbättringar som gjorts på denna arkitektur jämfört med tidigare generationer. Ramverket OpenCL och den relativa påverkan hos flertalet olika optimeringar av en parallell applikation har utvärderats genom att implementera Lukas-Kanades algoritm för uppskattning av optiskt flöde.

RV700-arkitekturen är ej lämpad för generella beräkningar. Prestandan hos G80-arkitekturen är utmärkt trots dess relativa ålder. Mycket möda måste dock tillägnas G80-specifika optimeringar av den parallella applikationen för att kunna uppnå högsta möjliga prestanda. Fermi är överlägsen i alla aspekter av GPGPU. Fermis nya minneshierarki tillåter att generella beräkningar utförs både lättare och snabbare än tidigare. På samma gång är Fermis prestanda mycket högre än hos de två andra arkitekturerna och detta redan innan några hårdvaruspecifika optimeringar gjorts.

Programmeringsramverket OpenCL är ett stabilt och kompetent ramverk väl anpassat för GPGPU-projekt som kan dra nytta av den ökade flexibiliteten av mjuk- och hårdvaruoberoende. Om prestanda är viktigare än flexibilitet kan dock NVIDIAs Compute Unified Device Architecture (CUDA) eller AMDs ATI Stream vara bättre alternativ.

Acknowledgements

We would like to thank Combitech AB for making this thesis possible. We would also like to thank our supervisors Peter Gelin and Dennis Arkeryd at Combitech AB as well as Ulf Assarsson at Chalmers University of Technology for their help and feedback. Finally we would like to give a special thanks to our families for their love and support.

Contents

Abbreviations

ALU	Arithmetic Logic Unit
CPU	Central Processing Unit
CTM	Close To Metal (AMD programming framework)
CUDA	Compute Unified Device Architecture (NVIDIA programming framework)
ECC	Error Correcting Code
FPU	Floating Point Unit
GFLOPS	Giga Floating-point Operations Per Second
GPGPU	General-Purpose computing using Graphics Processing Units
GPU	Graphics Processing Unit
HPC	High-Performance Computing
ID	Identification number
LK-method	The Lucas-Kanade method for optical flow estimation
LS-method	The Least-Squares method
MADD	Multiply And Add
OpenCL	Open Computing Language
OpenCV	Open Source Computer Vision library
RGB	Red Green Blue (a color model)
SFU	Special Function Units
SIMD	Single Instruction Multiple Data
SM	Streaming Multiprocessor
SP	Streaming Processor
TFLOPS	Tera Floating-point Operations Per Second
VLIW	Very Long Instruction Word

1. Introduction

General-Purpose computing using Graphics Processing Units (GPGPU) has been an area of active research for many years. During 2009 and 2010, there have been major breakthroughs in the GPGPU research field with the release of the Open Computing Language (OpenCL) programming framework and the new NVIDIA Fermi Graphics Processing Unit (GPU) architecture.

OpenCL is an open, royalty-free standard for cross-platform, parallel programming of modern processors found in personal computers, servers and handheld/embedded devices. The OpenCL standard has support from both NVIDIA and AMD and implementations exists on many operating systems including Windows, Mac OS X and Linux [1].

The NVIDIA Fermi architecture is a major leap forward for the GPGPU research field. NVIDIA calls the Fermi architecture "the world's first computational GPU" and has improved almost every aspect of the hardware architecture compared to previous generation of architectures. NVIDIA's Fermi architecture was released in April 2010 [2].

The background, purpose and delimitations of this thesis will follow in section 1. Section 2 is dedicated to previous work within the GPGPU research field. A history of graphics hardware and the GPGPU field as well as a description of recent GPU hardware architectures can be found in section 3. Section 3 also describes the OpenCL programming framework.

Section 4 will focus on our work to implement a parallel GPGPU application and to make this application fully exploit the available parallelism within a GPU. A description of several different application optimizations aimed at increasing performance can also be found in section 4.

Section 5 describes the results of our work. Conclusions, discussion and suggestions for future work can be found in sections 6 and 7.

1.1. Background

During the last decades, microprocessors based on single Central Processing Units (CPU) have been driving rapid performance increases in computer applications. These microprocessors brought Giga Floating-point Operations Per Second (GFLOPS) to desktop computers. Between each new generation of CPUs the clock frequency has increased. However, in the last few years the increase in clock frequency has slowed down because of issues with energy consumption and heat dissipation.

Instead of increasing clock frequency between different generations of CPUs, the development has moved more and more towards increasing the number of processing units used in each processor. Using more than one processing unit in each CPU changes the way

that the CPU should be utilized in order to get maximum performance. To fully utilize a multi-core CPU architecture, most applications must be redesigned to explicitly exploit parallel execution [3].

Parallel execution is an execution model where the different processing units in a processor simultaneously run different parts of an application or algorithm. This is accomplished by breaking the application into independent parts so that each processing unit can execute its part of the application or algorithm simultaneously with the other processing units. Using parallel execution usually results in increased performance compared to executing the application in a sequential fashion.

Programming an application to exploit parallel execution (parallel programming) has been a popular programming technique for many years, mainly in High-Performance Computing (HPC) using computer clusters or "super computers". There are three main reasons to utilize parallel programming; 1) To solve a given problem in less time, 2) To solve bigger problems within a given amount of time and 3) To achieve better solutions for a given problem in a given amount of time. Lately, the interest in parallel programming has grown even more in popularity because of the availability of multi-core CPUs [3].

In addition to the multi-core architecture of CPUs there is a microprocessor architecture called many-core. The many-core architecture was developed to increase execution throughput of parallel applications in single CPUs. The cores in a many-core microprocessor are usually heavily multithreaded unlike the single-threaded cores in a multi-core architecture. The most common many-core microprocessors are the ones used in Graphics Processing Units (GPU) [3]. Recent GPUs have up to 1600 processing cores and can achieve over 2.7 TFLOPS compared to recent CPUs which usually have 4 or 8 cores and can reach around 150 GFLOPS. Note that these numbers represent throughput in single-precision floating point calculations [3][4].

With their immense processing power, GPUs can be orders of magnitude faster than CPUs for numerically intensive algorithms that are designed to fully exploit the parallelism available. The GPGPU research field has found its way into fields as diverse as artificial intelligence, medical image processing, physical simulation and financial modeling [5].

During the last years several programming frameworks have been developed to help programmers accelerate their applications using the processing power of GPUs. The two most notable frameworks are NVIDIA’s Compute Unified Device Architecture (CUDA) released in 2007 and AMD’s ATI Stream released in 2008. A common problem with the CUDA and ATI Stream frameworks is that they can only be used with NVIDIA’s or AMD’s GPUs respectively. A possible solution to this problem came in December 2008 with the specification of a new standardized programming framework called OpenCL. The OpenCL standard has since been implemented on many different platforms by a long range of manufacturers. OpenCL can be used with both AMD’s and NVIDIA’s GPUs [6].

1.2. Purpose

This thesis explores the hardware architectures of three GPUs and how well they support general computations; the NVIDIA Geforce 8800 GTS (the G80 architecture) from 2006, the AMD Radeon 4870 (the RV700 architecture) from 2008 and the NVIDIA Geforce GTX 480 (the Fermi architecture) from 2010. Throughout this thesis these three GPU architectures are referred to as *G80*, *RV700* and *Fermi* respectively. This thesis further explores the impact of several parallel application optimizations; how can a parallel application be optimized to utilize the most recent advances in GPU hardware and how much knowledge concerning the underlying hardware is the programmer required to have before being able to fully utilize it? A secondary goal for this thesis is to explore the OpenCL programming framework.

To compare different hardware architectures and to evaluate different application optimizations and the OpenCL framework a parallel application will be developed. The Lucas-Kanade method (LK-method) for optical flow estimation published in 1981 has previously been implemented as a parallel GPU application using the NVIDIA CUDA framework [6][7][8][9]. The LK-method is well suited for a parallel implementation because it is computationally intensive and because all of the calculations in the method can be done in parallel. Implementing the LK-method in OpenCL will make it possible to evaluate parallel application optimizations as well as the OpenCL framework itself.

While there have been several papers published concerning a parallel implementation of the LK-method none of the previous papers have described an OpenCL implementation. Neither have any publications been made using both OpenCL and the NVIDIA Fermi architecture. This thesis describes our OpenCL implementation of the LK-method, the application optimizations evaluated to obtain the highest possible performance of the application and the differences in performance when running the application on the G80, Fermi and RV700 architectures.

1.3. Delimitations

Since the original LK-method was published numerous improvements have been suggested [8][9][10]. These improvements are outside the scope of this thesis and will not be implemented in our parallel application.

The new NVIDIA Fermi architecture is the main GPU architecture described in this thesis and the graphics hardware is described using NVIDIA terminology. The concepts described are however also to a large extent valid for the AMD graphics hardware. It is often more obvious to compare the Fermi architecture to the G80 architecture because they are produced by the same manufacturer and share the same architectural foundation. Optimizations and benchmarks are carried out and tested on all three GPU architectures.

Due to the limited time frame benchmarks are only measured on the Microsoft Windows platform. It should however be possible to run the application on any OpenCL compatible

platform. All benchmarks are measured using a 1080p “FullHD” video with a resolution of 1920x1080 pixels at 23.98 frames per second.

The aim is to be thorough evaluating application hardware optimizations. Should however time become a limiting factor the optimizations believed to have the greatest impact on performance will be evaluated first.

2. Previous work

Research regarding general-purpose computation on computer graphics hardware has been conducted for many years, beginning on machines like the Ikonas in 1978, the Pixel Machine in 1989 and Pixel-Planes 5 in 1992 [11][12][13]. The wide deployment of GPUs in the last several years has resulted in an increase in experimental research using graphics hardware. [14] gives a detailed summary of the different types of computations available on recent GPUs.

Within the realm of graphics applications, programmable graphics hardware has been used for procedural texturing and shading [13][15][16][17]. Methods of using GPUs for ray tracing computations have been described [17][18]. The use of rasterization hardware for robot motion planning is described in [19]. [20] describes the use of z-buffer techniques for the computation of Voronoi diagrams. The PixelFlow SIMD graphics computer was used to crack UNIX password encryption in 1997 [21][22]. GPUs have also been used in computations of artificial neural networks [22][23][24].

The official NVIDIA CUDA website "CUDA Zone" today features more than 1000 CUDA applications spanning a wide range of different research fields [25]. Several papers regarding GPGPU and optical flow can be found at the CUDA Zone. A few of these papers describe CUDA implementations of the LK-method. The two most notable papers are a real time multi-resolution implementation of the LK-method and "FOLKI-GPU", which uses a window-based iterative multi-resolution Lucas-Kanade type registration for motion detection [8][9].

3. General-Purpose computing on Graphics Processing Units

Section 3 describes the history of graphics hardware and the GPGPU research field. It also describes recent graphics hardware architectures in detail and ends with a description of the OpenCL programming framework.

3.1. History

Section 3.1 describes the history of graphics hardware and the origin of the GPGPU research field.

3.1.1. History of graphics hardware

In the beginning of the 1970s, the only purpose of graphics hardware was to map text and simple graphics to the computer display. On Atari 8-bit computers, chips provided hardware control of graphics, sprite positioning and display. At this time the general purpose CPU had to handle every aspect of graphics rendering. In the 1980s the computer game industry started using simple graphics hardware to accelerate the graphics computations. The Amiga featured a full graphics accelerator, offloading practically all graphics functions to hardware. In the early and mid-1990s, CPU-based real-time 3-dimensional (3D) graphics were becoming more and more common in computer games. This led to an even higher demand for GPU-accelerated 3D graphics. The graphics frameworks Microsoft DirectX and Khronos Group OpenGL were released and together with the market's demand for high-quality real-time graphics the frameworks became a driving force for the development of graphics hardware [26].

Early in the evolution of 3D-rendering, GPUs used a fixed-function graphics pipeline to render 3D-scenes. Scenes were rendered using the rasterization technique in which a scene is represented by a large number of triangles. The first stage in the fixed-function graphics pipeline converted triangle data originating from the CPU to a form that the graphics hardware could understand. The next pipeline stages colorized and applied textures to the triangles, rasterized the 3D-scene to a 2D-image and colorized the pixels in the triangles. In the final stage in the pipeline, the output image was stored in a frame buffer where it resided until it was displayed on the screen [3].

The pipeline stages that apply lighting, color and textures to triangles and pixels are called "shaders". In the early fixed-function pipelines two shaders existed, a vertex (triangle) shader and a pixel shader. In 2001 new shader functions was introduced by Microsoft when they released version 8 of the DirectX framework. For GPUs to support the new shader-functions the fixed-function hardware shaders had to be made programmable. Programmable shaders increased the level of freedom for graphics programmers and made it possible to generate more realistic scenes. In the shaders, each triangle and pixel could now be processed by a short program before it was sent to the next stage in the pipeline. Each shader had its own dedicated pool of processor cores that executed the shader program. The processor cores were

quickly becoming more and more flexible and the possibility to run floating point math was soon introduced.

In 2006 version 10 of the DirectX framework was released. DirectX 10 introduced a unified processor architecture where all shaders were executed by a single shared pool of processor cores. DirectX 10 also introduced an additional shader called a "geometry shader". The geometry shader further increased the level of freedom for programmers and made the graphics pipeline more flexible [3].

3.1.2. History of the GPGPU research field

With the introduction of the unified processor architecture GPUs started to resemble high-performance parallel computers and many research groups started to work on ways to use the inherent computing power in GPUs for general purpose computing. In the beginning of the evolution of GPGPU, it was difficult to develop applications because of the lack of any real programming framework. Developers were forced to express their computational problems in terms of graphics based frameworks like DirectX or OpenGL. To solve this problem a few research groups started building API layers on top of the graphics frameworks to simplify GPGPU programming and make it available for mainstream developers. One of the most notable projects was BrookGPU, developed at Stanford University in 2003 [27].

In November 2006, AMD released their Close To Metal (CTM) low-level programming interface. CTM gave developers access to the native instruction set and memory in AMD's GPUs. By opening up the architecture, CTM provided developers with the low-level, deterministic and repeatable access to hardware necessary to develop essential tools such as compilers, debuggers, math libraries and application platforms. With the introduction of CTM, a whole new class of applications was made possible. For example, a GPU-accelerated client for the distributed computing project "Folding@home" was created that was 30x faster than the CPU version [28][29].

In 2007 NVIDIA released a GPGPU framework called Compute Unified Device Architecture (CUDA). CUDA is a complex C-language based GPGPU computing framework and the computing engines in NVIDIA's GPUs. CUDA was the first standalone GPGPU framework that was not layered on top of the existing graphics frameworks; this made it a more streamlined product. Shortly after NVIDIA, AMD released a GPGPU framework based on the BrookGPU research. AMD's framework is now called "ATI Stream".

With AMD's release of CTM and ATI Stream and NVIDIA's release of the CUDA framework the GPGPU research field gained much popularity. The popularity contributed to an increased pressure on GPU manufacturers to improve their GPU architectures for general computing by adding more flexibility to the programming model. Two examples of improvements that has been made solely for the reason of GPGPU is the addition of double precision floating point calculations and IEEE compliance for floating point calculations [3].

The specification of a new GPGPU framework, OpenCL, was released 2009. OpenCL is developed by the Khronos Group with the participation of many industry-leading companies and institutions including Apple, Intel, AMD and NVIDIA. OpenCL is an open, royalty-free standard for cross-platform, parallel programming of modern processors found in personal computers, servers and handheld/embedded devices. OpenCL in many ways resemble the CUDA and ATI Stream frameworks but has the advantage that it is platform independent and hence can be run on both NVIDIA and AMD hardware. OpenCL implementations exist on Windows, Linux and Mac OS X as well as on many other operating systems. The OpenCL framework is described in section 3.3 [1].

3.2. Graphics hardware

As stated in section 3.1, the graphics pipeline has evolved from a pipeline with fixed-function hardware to a pipeline with programmable hardware. However, some stages of the graphics pipeline are still located in fixed-function hardware. The fixed-function stages are only used to render graphics and are not relevant for GPU computing. Section 3.2 describes the GPU hardware architecture from a GPU computing point of view. Figure 3.1 shows an overview of a recent GPU hardware architecture.

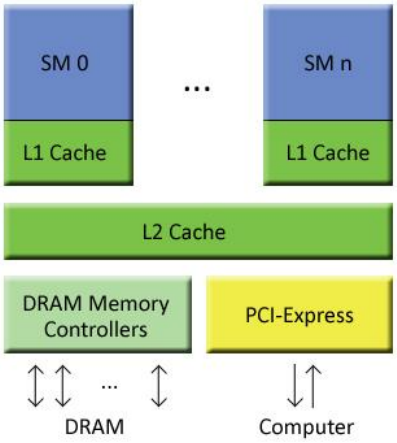

Figure 3.1. An overview of a recent GPU hardware architecture.

Processing cores in a unified processor architecture are located in a unified processor pool and are grouped into clusters called Streaming Multiprocessors (SM). The SM work scheduling and execution model is described in section 3.2.1 and 3.2.2 and the SM hardware is described in detail in section 3.2.3.

Graphics hardware contains several different types of memory. The main memory resides in large memory areas outside the actual GPU chip. Special memory controllers handle main memory accesses. To speed up memory accesses a many-level memory hierarchy is used, described in section 3.2.4. Section 3.2 is concluded with a description of some major improvements made specifically in the new NVIDIA Fermi architecture.

3.2.1. Global scheduling

A parallel program executing on a GPU is called a kernel. For a kernel to fully utilize the GPU processing power it has to be executed by several thousand parallel threads. The GPU has a global scheduler that issues parallel threads to the SMs where they are executed. Scheduling is done by splitting the threads into thread blocks with a predefined size. The global scheduler then assigns blocks to the SMs (a single block cannot be split between two SMs). The global scheduler has to take several hardware constraints into consideration, for example the amount of blocks and the amount of threads that can be scheduled to a single SM is limited.

A SM in the NVIDIA Fermi architecture can execute a maximum of 8 thread blocks or 1536 threads at a time. If there are 128 threads per block the scheduler will be constrained by the maximum amount of blocks and 8 blocks will be scheduled to each SM. However, if a block contains 512 threads the scheduler will be constrained by the amount of threads and only 3 blocks can be scheduled to each SM. How to determine the thread block size for optimal performance is described in section 4.5.3. In addition to the maximum block and thread constraints the global scheduler also has to consider thread register usage and block shared memory usage [30][31].

3.2.2. Execution model

The SMs in a GPU execute instructions using a thread level Single Instruction Multiple Data (SIMD) model. A predefined number of threads form a SIMD unit. All threads in the same unit are executed together; the same instruction is executed for all the threads. The SIMD units are called warps using NVIDIA terminology and wavefronts using AMD terminology. On some GPU architectures execution is split into half-warps. In that case the threads in the first half of the warp execute the instruction first immediately followed by the threads in the second half of the warp.

On recent GPUs, warps consist of 32 threads and wavefronts of 64 threads. The threads in a warp are made up by 32 consecutive threads aligned to even multiples of 32, beginning at thread 0. For instance, threads 0-31 and threads 96-127 each make up single warps but threads 3-34 do not since they do not form an even multiple of 32. It is important to note the difference between a thread block (see section 3.2.1) and a warp. Threads are divided into thread blocks by the programmer while warps are a micro architectural entity that is not visible to the programmer [30][31].

3.2.3. Streaming multiprocessors

The SMs in a GPU are highly threaded SIMD processors. The SMs execute all computational work submitted to the GPU. Each SM contains an instruction cache, an instruction queue, a warp scheduler, registers, memory and functional units. Figure 3.2 shows the hardware architecture of a SM.

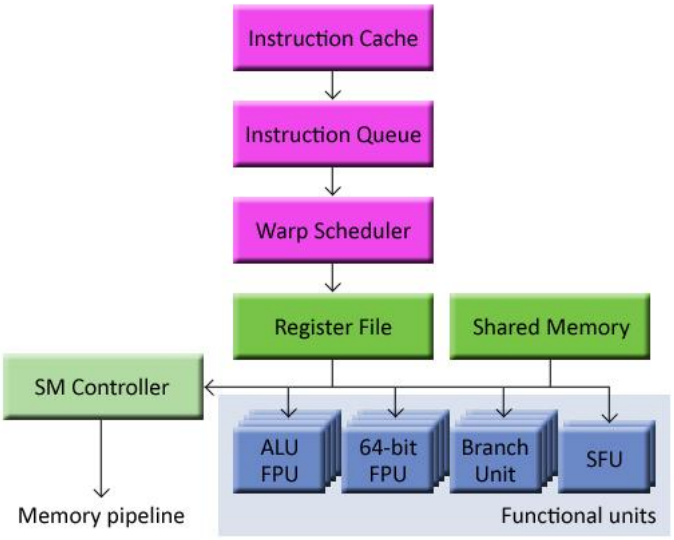

Figure 3.2. The hardware architecture of a SM.

3.2.3.1. Scheduling

The instruction cache and the instruction queue contain kernel instructions for the currently executing warps. The warp scheduler selects the instructions in the instruction queue that should be executed during the next clock cycle. The selection is made depending on the status and priority of the instructions in the queue.

Instructions in the instruction queue can be in one of two states reflected by a status flag; "ready" and "not ready". When an instruction enters the queue its status is set to "not ready". The instruction status is changed to "ready" when all operands and memory areas for the instruction are available. The priority of an instruction is determined using a modified round-robin algorithm that takes several different factors into account. The warp scheduler selects and issues the highest priority instruction with status "ready".

Another important purpose of the warp scheduler is to "hide" memory latencies that arise when data is accessed in main memory. Hiding latencies can be achieved by executing other warps while one warp is waiting for a memory instruction to complete. However, this requires that there are sufficient independent arithmetic instructions that can be issued while waiting for the memory instruction to complete. Switching execution between two warps does not imply any notable overhead time [30][31].

3.2.3.2. Memory

There are two kinds of memory inside a SM, a register file and a shared memory area. The register file contains thousands of high bandwidth registers. The registers are divided among all threads scheduled to a SM. The Fermi architecture has a 128 KB large register file per SM which is split into 32768 4-byte registers. If a Fermi SM has 1536 active threads, each thread can use a maximum of 20 registers (80 bytes). If a thread requires more than 20 registers, the number of threads scheduled per SM has to be lowered, i.e. the scheduler has to lower the number of thread blocks scheduled to each SM.

The shared memory area is used for communication between threads in a thread block. Shared memory is partitioned between the blocks scheduled to a SM. Shared memory entries are organized into memory banks that can be accessed in parallel by threads in a warp. However if two threads access the same bank at the same time a bank conflict will occur and the accesses have to be serialized. Accessing shared memory is as fast as accessing registers if no bank conflicts occur. Recent GPUs often have at least 16 KB shared memory. If a SM has 8 active blocks, each block can use have a maximum of 2 KB shared memory. If a kernel uses more than 2 KB of shared memory the number of blocks scheduled per SM has to be lowered, reducing performance. Optimizations for conflict free shared memory access are described in section 4.5.2.4 [30][31].

3.2.3.3. Functional units

The functional units in a SM execute the instructions that are issued by the warp scheduler. The functional units consist of Streaming Processors (SP), branch units and Special Functions Units (SFU). The Fermi architecture has 32 SPs in each of its 15 SMs (480 SPs in total).

A SP has two dedicated data paths, an integer data path and a floating point data path. The integer data path contains an Arithmetic Logic Unit (ALU) and the floating point data path contains a Floating Point Unit (FPU). The two data paths cannot be active simultaneously. Since integer and floating point instructions are the most common instructions in GPU computing, most of the work assigned to a SM is executed in its SPs. Support for double precision floating point calculations in the SPs is implemented in different ways on different architectures. Some architectures use separate double precision floating point units and others split the double precision (64-bit) instruction into multiple single precision (32-bit) instructions.

Depending on the context, accuracy can be an important factor when doing calculations. The IEEE 754-2008 floating point standard specifies requirements that must be fulfilled to achieve a high level of accuracy for both single and double precision calculations [32]. Only the most recent GPU architectures conform to the IEEE 754-2008 floating point standard. Because the standards specified in IEEE 754-2008 are not important for graphics rendering this is an example of an architectural improvement made for the sole purpose of GPGPU.

The branch units in a SM execute control flow instructions and the SFUs execute less common math instructions. The SFUs are clusters of several units that handle for example transcendental, reciprocal, square root, sine and cosine functions. Because these instructions are quite rare, a GPU has fewer SFUs than SPs. The Fermi architecture only has 4 SFUs per SM [30][31].

3.2.4. Memory pipeline

Global memory load and store instructions are processed in the SMs but involve many different parts of the hardware, see figure 3.3. The load and store instructions are sent to the SM controller which arbitrates access to the memory pipeline and acts as a boundary between the memory and the SMs. The memory instructions can either access the registers, shared

memory or the off-chip main memory. The main memory is implemented in off-chip DRAM modules. The Fermi architectures implement main memory in GDDR5 DRAM and have six GDDR5 memory controllers, supporting up to 6 GB of main memory.

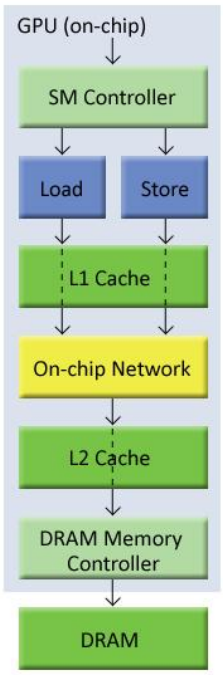

Figure 3.3. The different stages of the memory pipeline.

DRAM has high latency making accesses to global memory slow. Recent GPU architectures have a two level cache hierarchy designed to speed up main memory accesses. In previous generations of GPUs, the cache was only utilized when accessing texture data for graphics rendering. Fermi is the first GPU architecture to provide caching for all memory accesses.

The level 1 (L1) cache is located close to the SMs. The purpose of the L1 cache is to speed up main memory accesses. The L1 cache implementation differs between different GPU architectures; sometimes the L1 cache is shared between several SMs while other implementations have a dedicated L1 cache for each SM. Compared to the L1 cache the L2 cache is located closer to the main memory and is utilized by all the SMs. The purpose of the L2 cache is not only to cache data from main memory but also to optimize main memory accesses into as few transactions as possible, improving bandwidth utilization. The Fermi L2 cache is 768 KB [30][31].

3.2.5. The NVIDIA Fermi architecture

The Fermi architecture has undergone major improvements compared to previous generations of graphics architectures. Prior to Fermi, GPUs could only schedule one kernel at a time. In Fermi the global scheduler has been improved and can schedule many kernels simultaneously. When executing multiple kernels, it is however required that each SM executes blocks from the same kernel. The Fermi architecture has 15 SMs and can hence execute a maximum of 15 kernels simultaneously. Executing multiple kernels simultaneously means that smaller kernels can be more efficiently dispatched to the GPU. To be able to execute many kernels efficiently

it is possible to copy data between host memory and device memory concurrently with kernel execution on the Fermi architecture.

The SMs have undergone many improvements in the Fermi architecture. The number of SMs has been increased and there are more functional units inside each SM. The greatest improvement is the transition from a single-issue pipeline to a dual-issue pipeline. The dual-issue pipeline can be seen in figure 3.4

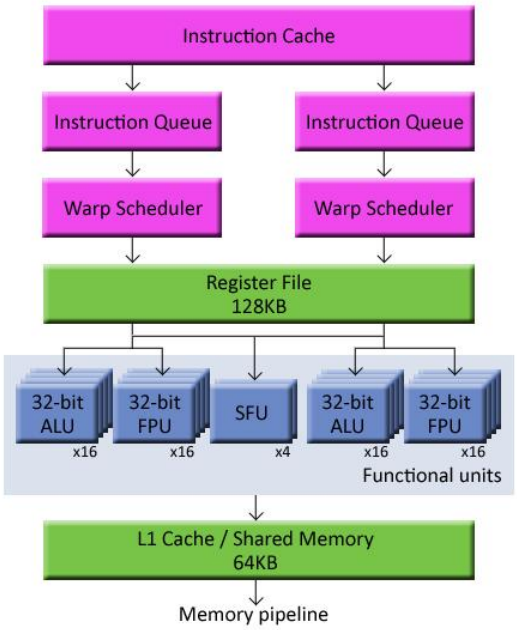

Figure 3.4. The Fermi SM hardware architecture.

Each SM features two instructions queues and two warp schedulers allowing two warps to be issued and executed concurrently. Fermi's dual warp scheduler selects two warps and issues one instruction from each warp to a group of 16 cores, 16 load/store units and 4 SFUs. Intra instruction dependencies cannot exist between two warps because they execute independently of each other.

Most instructions can be executed concurrently in the dual-issue pipeline. However, double precision instructions do not support dual dispatch with any other instructions because both pipelines are used when executing double precision instructions. The possibility of executing double precision instructions using two pipelines has greatly increased double precision performance compared to previous architectures.

Another architectural improvement in the SMs is the combined shared memory and L1 cache memory area. Each SM has 64 KB memory that can be configured as either 48 KB shared memory and 16 KB L1 cache or as 16 KB shared memory and 48 KB L1 cache. A configurable memory area makes it possible to fine tune the hardware depending on application. The Fermi L2 cache has also been improved to aid the coalescing of global memory accesses. Section 4.5.2.1 describes memory coalescing in detail.

Additional improvements to the Fermi architecture include Error Correcting Code (ECC) protection that can be enabled for all memory areas in the memory pipeline and a unified address space for registers, shared memory and main memory. With a unified address space Fermi supports pointers and object references, which are necessary for C++ and other high-level languages [2][31].

3.3. Open Computing Language (OpenCL)

A motivation for the development of OpenCL was that other CPU-based parallel programming frameworks did not support complex memory hierarchies or SIMD execution. Most existing GPU-based programming frameworks do support both memory hierarchies and SIMD execution; however they are limited to specific vendors or hardware. This section describes OpenCL as implemented on recent GPUs [1][3].

The OpenCL framework can be described using four models; the platform model describes how OpenCL handles devices, the memory model describes the different memory types on an OpenCL device, the execution model describes how the sequential and parallel parts of an application are executed by OpenCL and the programming model describes the different parallel execution models that are supported by OpenCL [1].

3.3.1. Platform model

The OpenCL parallel programming framework has a complex platform and device management model that reflects its support for multiplatform and multivendor portability. This differentiates OpenCL from other parallel programming frameworks. The OpenCL platform model consists of a host processor connected to one or more OpenCL compute devices. The host processor is usually a CPU. OpenCL compute devices are divided into compute units which in turn are further divided into processing elements. On a GPU, the compute units are implemented in the SMs and the processing elements are implemented in the functional units. Figure 3.5 shows the OpenCL platform model [1].

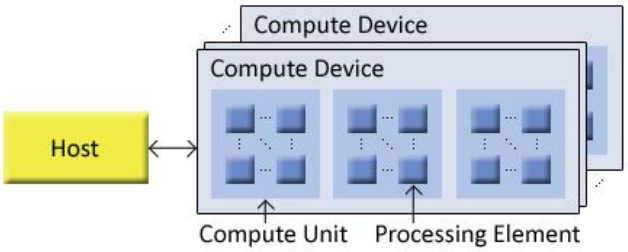

Figure 3.5. The OpenCL platform model.

All computations on a device occur within the processing elements. An OpenCL application runs on the host processor and submits commands to execute computations on the processing elements within a compute device.

3.3.2. Execution model

OpenCL applications consists of two parts; a sequential program executing on the host and one or more parallel programs (kernels) executing on the device(s). The host program defines

the context for the kernels and manages their execution. Each OpenCL device has a command queue where the host program can queue device operations such as kernel executions and memory transfers.

The core of the OpenCL execution model is defined by how kernels are executed. When a kernel is chosen for execution, an index space called “NDRange“ is defined and an instance of the kernel executes for each index in the NDRange. The kernel instance is called a work-item. Each work-item is identified by a global identification number (ID) that is its index in the NDRange. All work-items execute the same kernel code but the specific execution path through the code and the data operated upon can vary.

Work-items are organized into work-groups. Each work-group is assigned a unique work-group ID and each work-item within a work-group is assigned a local ID. A single work-item can be uniquely identified by its global ID or by a combination of its work-group ID and local ID. The work-items in a single work-group execute concurrently on the processing elements of a single compute unit. Work-items within a work-group can synchronize with each other and share data through local memory on the compute unit. On a GPU, the work-groups are implemented as thread blocks and the work-items as threads.

The NDRange is an N-dimensional index space, where N is 1, 2 or 3. The global, local and group IDs are N-dimensional tuples. For example in a 2D NDRange all IDs are 2D tuples on the form (x,y) and in a 3D NDRange on the form (x,y,z). The size and dimension of the NDRange as well as the size of work-groups are chosen by the programmer. Choosing a size and dimension that best fit the parallel application will result in increased performance. For example, if the application data set is on the form of a linear array then a 1D NDRange will usually yield the best performance while for 2D data set a 2D NDRange usually is better suited. How to find the best NDRange dimensionality and size is described in more detail in section 4.5.3. Figure 3.6 shows an example where a host executes two kernels with different NDRanges [1][3].

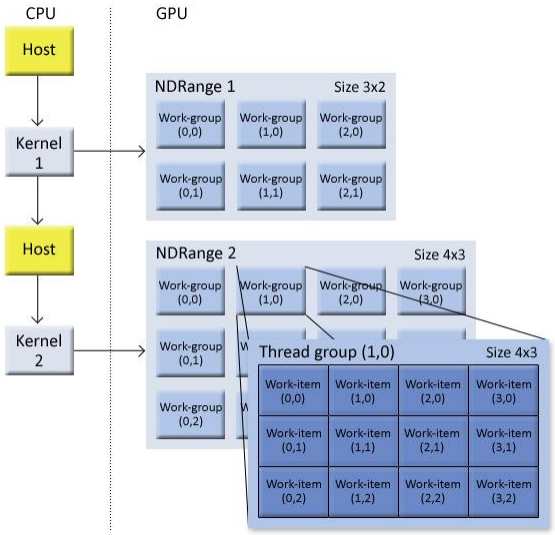

Figure 3.6. A host executing two kernels with different NDRanges.

3.3.3. Memory model

There are four different memory areas that a work-item can access during execution; global memory, constant memory, local memory and private memory. The OpenCL memory model can be seen in figure 3.7.

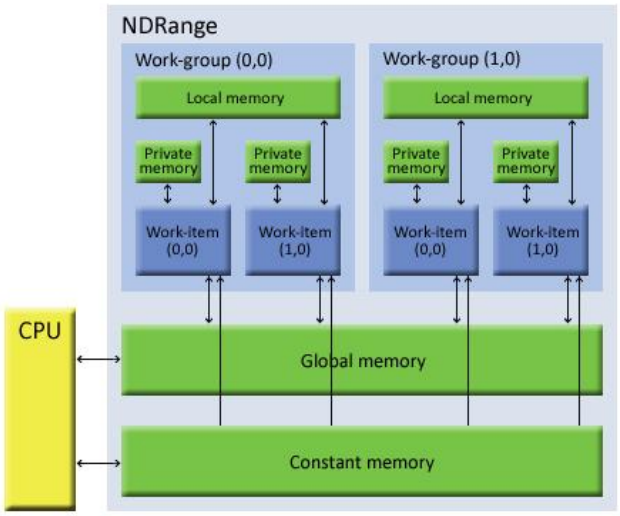

Figure 3.7. The OpenCL memory model.

The global memory is the main memory of the device. All work-items have read and write access to any position in this memory. A part of global memory is allocated as constant memory and remains constant during execution of a kernel. The constant memory is cached which results increased performance compared to global memory when memory accesses result in a cache hit. The global and constant memory can also be accessed from the host processor before and after kernel execution.

Each work-group has local memory that is shared among the work-items in the work-group. The local memory may be implemented as dedicated regions of memory on the OpenCL device. Alternatively, the local memory region may be allocated as a part of the global memory. The current version (1.0) of OpenCL supports up to 16 KB local memory. Each work-item also has private memory. Variables defined in private memory are not visible to other work-items. Local variables created in a kernel are usually stored in the private memory.

The different memory types in OpenCL are implemented in different ways depending on the device type. For example, a GPU usually has global memory implemented in off-chip main memory, local memory in the shared memory area and private memory in the register files. The different memory implementations have very different performance and size. The main memory is large but has high access latency while the local registers are small but have very short access latency. Memory optimizations in OpenCL will be described in section 4.5.2 [1][3].

3.3.4. Programming model

The OpenCL execution model supports the data parallel and the task parallel programming models. The data parallel model is currently driving the design and development of OpenCL.

A data parallel programming model defines a computation as a sequence of instructions applied to multiple data elements (also known as the SIMD execution model, see section 3.2.2). Each work-item execute the same instructions but on different data elements. In a strictly data parallel model, there is a one-to-one mapping between work-items and data elements during kernel execution. OpenCL implements a relaxed version of the data parallel programming model where a strict one-to-one mapping is not required; a work-item can access any data element.

Task parallelism is achieved when different work-items execute different kernels on the same or different data elements. The task parallel programming model is difficult to implement on a GPU using OpenCL because the OpenCL GPU implementations does not yet support execution of multiple kernels at the same time. GPUs do however support task parallelism within a kernel since different work-items are able to follow different execution paths [1][3].

4. Implementing and optimizing an OpenCL application

Section 4 describes the parallel application developed during this thesis; an OpenCL implementation of the Lucas-Kanade method (LK-method) for optical flow estimation. The LK-method was first implemented as a sequential application running on a CPU and was then parallelized and accelerated using a GPU. Section 4 further describes the application optimizations implemented to achieve optimal performance for the parallel application.

4.1. Application background

Optical flow is the pattern of apparent motion of objects, surfaces and edges in a visual scene caused by the relative motion between the observer (an eye or a camera) and the scene [33][34]. There are many uses of optical flow but motion detection and video compression have developed as major aspects of optical flow research. The LK-method describes an image registration technique that makes use of the spatial intensity gradient to find a good match between images using a type of Newton-Raphson iteration. The method has proven much useful and has been used in a great variety of different applications.

The LK-method tries to estimate the speed and direction of motion for each pixel between two image frames. The accuracy of the original LK-method from 1981 is not perfect and several alternative methods exists [35][36][37]. Numerous suggestions for improvements and further developments of the LK-method have also been published. One interesting development is a multi-resolution implementation of the LK-method that utilizes iterative refinement to improve accuracy [8]. Extending the LK-method with the use of 3D tensors is another extension that has proven superior in producing dense and accurate optical flow fields [36][37]. Another interesting development is the combination the original LK-method and a feature detection and tracking algorithm [38][39]. This thesis describes the implementation of the original LK-method from 1981.

4.2. The Lucas-Kanade method for optical flow estimation

The LK-method uses a two-frame differential method for optical flow estimation. The LK-method is called differential because it is based on local Taylor-series approximations of the image data; that is, it uses partial derivatives with respect to the spatial and temporal coordinates. A grayscale image frame can be represented by the image intensity equation. By assuming that the intensity in two image frames taken at times t and t+δt is constant the image constraint equation can be formulated:

$$I(x, y, t) = I(x + \delta x, y + \delta y, t + \delta t) \tag{4.1}$$

Equation 4.1. The image constraint equation.

where I is the image intensity, δx and δy represent movement in the spatial domain and δt represents movement in the temporal domain. Effects other than motion that might cause changes in image intensity, such as a change in the lighting conditions of the scene captured

in the image, can be ignored if the interval between image frames is small enough. Because of this, a fast frame rate (about 15 frames per second or greater, depending on the camera used and the velocity of any moving objects) is essential for real-time use of the LK-method [40].

Assume that the movement of a pixel between the two image frames in the spatial domain is small and that the temporal difference is one time unit ($\delta t = 1$). Expressing equation (4.1) in differential form and expanding it using Taylor series, it can be written as

$$I(x + \delta x, y + \delta y, t + \delta t) = I(x, y, t) + \frac{\partial I}{\partial x}\delta x + \frac{\partial I}{\partial y}\delta y + \frac{\partial I}{\partial t}\delta t + H.O.T \tag{4.2}$$

Equation 4.2. The image constraint equation expressed in differential form and expanded using Taylor series.

Combining equation (4.1) and (4.2), equation (4.3) can be derived

$$\frac{\partial I}{\partial x}\delta x + \frac{\partial I}{\partial y}\delta y + \frac{\partial I}{\partial t}\delta t = 0 \tag{4.3}$$

Equation 4.3. Deriving equation (4.3) from equations (4.1) and (4.2).

Dividing equation (4.3) with δt and substituting the horizontal velocity $\delta x/\delta t$ with V_x and the vertical velocity $\delta y/\delta t$ with V_y yields

$$\frac{\partial I}{\partial x}V_x + \frac{\partial I}{\partial y}V_y + \frac{\partial I}{\partial t} = 0 \tag{4.4}$$

Equation 4.4. Dividing equation (4.3) with δt and substituting $\delta x/\delta t$ and $\delta y/\delta t$ with V_x and V_y.

Substituting the horizontal spatial image frame derivative $\partial I/\partial x$ with I_x, the vertical spatial derivative $\partial I/\partial y$ with I_y and the temporal derivative $\partial I/\partial t$ with I_t, equation (4.4) can be written as

$$I_x V_x + I_y V_y = -I_t \tag{4.5}$$

Equation 4.5. Substituting the spatial and temporal derivatives of (4.4) and reorganizing.

Equation (4.5) is known as "the aperture problem" of optical flow algorithms. Equation (4.5) has two unknowns, V_x and V_y, so it cannot be solved directly. At least one additional equation is needed to solve (4.5), given by some additional constraint.

To solve equation (4.5), the LK-method introduces an additional constraint by assuming the optical flow to be constant in a local neighborhood around the pixel under consideration at any given time [6]. In other words, the LK-method assumes the optical flow V_x and V_y to be constant in a small window of size m x m, m>1, centered on the pixel (x,y). Each pixel

contained in the window produce one equation in the same form as (4.5). With a window size of m x m, $n=m^2$ equations are produced.

The n equations form a linear over-determined system. Expressing the equations in matrix form yields

$$\boldsymbol{X\beta} = \boldsymbol{y} \tag{4.6}$$

Equation 4.6. The n equations of the linear over-determined system expressed in matrix form.

Where **X**, **β** and **y** are

$$\boldsymbol{X} = \begin{bmatrix} I_{x1} & I_{y1} \\ I_{x2} & I_{y2} \\ \vdots & \\ I_{xn} & I_{yn} \end{bmatrix} \quad \boldsymbol{\beta} = \begin{bmatrix} V_x \\ V_y \end{bmatrix} \quad \boldsymbol{y} = -\begin{bmatrix} I_{t1} \\ I_{t2} \\ \vdots \\ I_{tn} \end{bmatrix}$$

An over-determined system such as (4.6) usually has no exact solution so the goal is to approximate V_x and V_y to best fit the system. One method of approximating V_x and V_y is the Least Squares method (LS-method). Using the LS-method the best approximation of V_x and V_y can be found by solving the quadratic minimization problem

$$\arg\min_{\beta} \sum_{i=1}^{m} \left| y_i - \sum_{j=1}^{n} X_{ij}\beta_j \right|^2 = \arg\min_{\beta} \|\mathbf{y} - \mathrm{X}\boldsymbol{\beta}\|^2 \tag{4.7}$$

Equation 4.7. The quadratic minimization problem suggested by the LS-method.

Provided that all the n equations of (4.7) are linearly independent, the minimization problem has a unique solution where the error of the approximation is minimized. Rewriting equation (4.6) according to equation (4.7) results in:

$$(\boldsymbol{X^T X})\widehat{\boldsymbol{\beta}} = \boldsymbol{X^T y} \tag{4.8}$$

Equation 4.8. Rewriting equation (4.6) according to equation (4.7).

Solving (4.8) for $\widehat{\boldsymbol{\beta}}$ yields the final velocity vectors for the pixel under consideration

$$\widehat{\boldsymbol{\beta}} = (\boldsymbol{X^T X})^{-1}\boldsymbol{X^T y} \tag{4.9}$$

Equation 4.9. Solving equation (4.8) for $\hat{\beta}$ yields to final velocity vectors V_x and V_y for the pixel under consideration.

In summary, to estimate the optical flow fields between two consecutive images in an image sequence using the LK-method the following steps must be executed:

1. Calculate the spatial and temporal derivatives for the images.
2. For each pixel, form a window of size m x m, centered on the pixel.

3. Use each pixel in the window to produce an equation in the same form as equation (4.5).
4. Use equation (4.5) to form the equation system (4.9). Solving equation (4.9) will yield the final velocity vectors V_x and V_y for the pixel under consideration.

4.3. Implementation of the Lucas-Kanade method

As stated in section 4.2, the LK-method uses a two-frame differential method for optical flow estimation. The optical flow is hence estimated by comparing two consecutive image frames in an image sequence or video. When estimating optical flow using the LK-method, color information in the image frames is of no use. Because of this, the image frames are converted to grayscale should they be color coded. For an image frame encoded using the Red Green Blue (RGB) color model the grayscale conversion is performed using equation (4.10)

$$i = \frac{(R + G + B)}{3} \tag{4.10}$$

Equation 4.10. The formula used when converting a RGB-color image to grayscale.

where i is the approximated grayscale value for the pixel and R, G and B are the red, green and blue values respectively. Next, the spatial derivatives I_x and I_y are approximated for each pixel. This is achieved by convolving each pixel in the first frame with the kernel listed in equation (4.11) to get the spatial derivative in the horizontal direction and equation (4.12) to get the spatial derivative in the vertical direction (do not confuse kernels used for convolution with OpenCL kernels which are parallel programs).

$$\frac{1}{4} * \begin{bmatrix} -1 & 1 \\ -1 & 1 \end{bmatrix} \tag{4.11}$$

Equation 4.11. The convolution kernel used for approximating the x-derivative, I_x, for a pixel.

$$\frac{1}{4} * \begin{bmatrix} -1 & -1 \\ 1 & 1 \end{bmatrix} \tag{4.12}$$

Equation 4.12. The convolution kernel used for approximating the y-derivative, I_y, for a pixel.

The temporal derivative, I_t, is approximated by subtracting the two frames. To make the implementation more robust against noise, the temporal derivative is convolved with the smoothing kernel in (4.13)

$$\frac{1}{4} * \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \tag{4.13}$$

Equation 4.13. The kernel used to smooth the temporal derivative, I_t, for a pixel.

When the spatial and temporal derivatives have been approximated for the whole frame, it is possible to start solving equation 4.9 for each pixel. By solving equation (4.9), the final

velocity vectors Vx and Vy can be established. See listing 4.1 for pseudo code of our sequential implementation of the LK-method.

```
WINDOW_SIZE = 9 // size of window, should be an odd number
WINDOW_RADIUS = (WINDOW_SIZE - 1) / 2

// value used to avoid singularities when using the cofactor method to inverse a 2x2 matrix
ALPHA = 0.001

// read image frame data from two frames taken at time t and t+delta_t
im1 = read frame1
im2 = read frame2

if <frames are color coded>
  convert frames to grayscale using equation (4.10)
endif

sub = im1 - im2 // frame difference, used for approximating the temporal derivative (It)

Ix = conv2(im1, kernel_x) // approximate the x-derivative using eq (4.11)
Iy = conv2(im1, kernel_y) // approximate the y-derivative using eq (4.12)
It = conv2(sub, kernel_t) // approximate the t-derivative using eq (4.13)

// loop over each pixel in frame
// only place window where it will fully fit inside the frame
for each pixel p at position (x,y) in im1 at least WINDOW_RADIUS pixels from the frame edges

  form window of size WINDOW_SIZE centered around pixel (x,y)

  // loop over each pixel inside the window
  for each pixel inside window
    IxWindow = Ix-values contained inside the window centered at (x,y) in Ix
    IyWindow = Iy-values contained inside the window centered at (x,y) in Iy
    ItWindow = It-values contained inside the window centered at (x,y) in It

    // Calculate all values needed in X^T * X from equation (4.9)
    Ix2 = sum(element_wise_multiplication(IxWindow * IxWindow))
    IxIy = sum(element_wise_multiplication(IxWindow * IyWindow))
    Iy2 = sum(element_wise_multiplication(IyWindow * IyWindow))

    // Calculate all values needed in X^T * y from equation (4.9)
    // negate the sum because y has a negative sign, see eq (4.6)
    IxIt = sum(element wise multiplication(IxWindow * ItWindow)) * -1
    IyIt = sum(element wise multiplication(IyWindow * ItWindow)) * -1

    // form matrix (X^T * X) according to eq (4.9)
    // (X^T * X) will always be a 2x2 square matrix. To inverse the XtX-matrix
    // it is possible to use the cofactor method. To avoid singularities
    // when using the cofactor method, add a small value alpha to position 0,0
    // and 1,1 of XtX
    XtX = matrix(2,2)
    XtX[0][0] = Ix2 + alpha
    XtX[0][1] = IxIy
    XtX[1][0] = IxIy
    XtX[1][1] = Iy2 + alpha


    // Caluclate the inverse of (X^T * X) using the cofactor method
    cofMat = matrix(2,2)
    cofMat[0][0] = XtX[1][1]
    cofMat[0][1] = -XtX[0][1]
    cofMat[1][0] = -XtX[1][0]
    cofMat[1][1] = XtX[0][0]

    // scalar value used to calculate the inverse of cofMat below
    s = 1 / (XtX[0][0] * XtX[1][1] - XtX[0][1] * XtX[1][0])

    // multiply matrix cofMat with scalar s - this yields the final inverted XtX matrix
    invXtX = scalar matrix multiply(s, cofMat)

    // form X^T * y
    Xty = matrix(2,1)
    Xty[0][0] = IxIt
    Xty[1][0] = IyIt

    // solve equation (4.9)
    beta = matrix_multiply(invXtX, Xty)

    // beta now contains our final velocity vectors Vx and Vy for pixel p at position (x,y)
    Vx = beta[0][0]
    Vy = beta[1][0]
end
```

Listing 4.1. Pseudo code for the sequential Lucas-Kanade implementation.

4.4. Parallel implementation of the Lucas-Kanade method

Not all applications can benefit from the computational power of the parallel processors inside a GPU. To be able to accelerate an application using a GPU, the application has to fit the parallel execution model, as described in section 3.2.2. For an application to fully benefit from the parallel processors inside a GPU, the data elements used in the application need to be independent so they can be processed in parallel. If, for example, half of the data elements in an application are independent and can be processed in parallel, it is said that the application is parallelizable to a degree of 50%.

How well an application performs when programmed for a GPU is highly dependent on how big the portion of the application is that can be parallelized. Assume that, for example, 95% of an application is parallelizable and is accelerated 100 times using a GPU. Further assume that the rest of the application remains on the host and receives no speedup. The application level-speedup is then

$$\frac{1}{(100\% - 5\%) + \frac{95\%}{100}} = \frac{1}{0.05 + 0.095} = \frac{1}{0.0595} \approx 17 \tag{4.14}$$

Equation 4.14. Speedup of an application where 95% is parallelizable and receives a speedup of 100x.

Equation (4.14) is a demonstration of Amdahl's law: The application level-speedup due to parallel computing is limited by the sequential portion of the application. In this case, even though the sequential portion of the application is quite small (5%), it limits the application speedup to 17x even though 95% of the program receives a 100x speedup [3].

$$S = \frac{1}{(1 - P) + \frac{P}{N}} \tag{4.15}$$

Equation 4.15. Amdahl's law specifies the maximum theoretical speed-up (S) that can be expected by parallelizing a sequential program. P is the portion of the program that can be parallelized and N is the factor by which the parallel portion is sped up.

Another important factor regarding the performance of a parallel application is its arithmetic intensity. Arithmetic intensity is defined as the number of arithmetic operations performed per memory operation. It is important for GPU accelerated applications to have high arithmetic intensity or the memory access latency associated with memory accesses will limit computational speedup. Ideal GPGPU applications have large data sets, high parallelism and minimal dependency between data elements [3].

As previously stated in section 1.2 the LK-method is well suited for parallel implementation. The LK-method is computationally intensive and all of the data elements (the pixels in the images) are independent so 100% of the application is parallelizable. It is hence theoretically possible to develop a fully parallel implementation of the LK-method without a sequential portion of the application to limit the speedup. Instead the speedup will be limited by other

factors such as the device processors performance, the memory bandwidth between the host and the device and the threading arrangement of the parallel implementation. The threading arrangement of the parallel implementation refers to how the computational work is decomposed into units of parallel execution. Section 4.4.1 will describe threading arrangement in more detail.

4.4.1. OpenCL implementation

As stated in section 3.3.2, OpenCL applications consists of two parts; a sequential program executing on the host and one or more parallel programs (kernels) executing on the device. To redesign a sequential application to a parallel application is not a straight forward task. Finding parallelism in large computational problems is often conceptually simple but can turn out to be challenging in practice. The key is to identify the work to be performed by each unit of parallel execution (a work-item in OpenCL) so the inherent parallelism of the problem is well utilized. One must take care when defining the threading arrangement of the parallel implementation. Different threading arrangements often lead to similar levels of parallel execution and the same execution results, but they exhibit very different performance in a given hardware system. The optimal threading arrangement for a parallel implementation is dependent on the computational problem itself [3].

Looking closer at the sequential implementation of the LK-method in listing 4.1, the different steps needed to estimate the optical flow for two image frames are:

1. Read the pixel data from both image frames.
2. Convert the image frame data to grayscale.
3. Subtract the pixel data in the second image frame from the pixel data in the first image frame.
4. Approximate the spatial and temporal derivatives by three convolution operations on the image frame data.
5. Loop over all pixels placing a window around each pixel.
6. Use each pixel inside the window to form a set of equations and approximate the best solution to these equations using the LS-method.

Our first naïve OpenCL implementation of the LK-method was divided into four kernels; a grayscale converter, a kernel for matrix addition and subtraction, a kernel for convolution and a kernel for the LS-method.

To be able to visualize the resulting motion vectors directly on the input frame a fifth kernel was introduced. The fifth kernel colors the pixels in the original input frames according to a certain color scheme. Using a full circle spanning all possible colors, motion in any direction can be represented by its own color tone, see figure 4.1. By letting the color intensity vary depending on the length of the calculated motion vector it is possible to illustrate how fast a pixel is moving. Using this scheme motion in any direction can be displayed by a unique color, see figure 4.2. Several other application optimizations were considered to increase performance of our parallel application. Section 4.5 describes these optimizations.

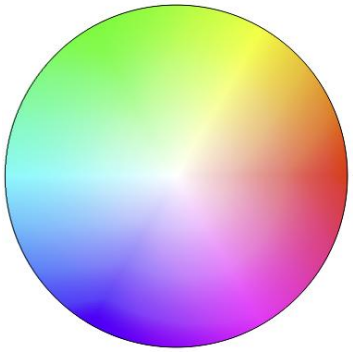

Figure 4.1. The color scheme used to visualize the motion vectors.

Figure 4.2. Screenshot of an image frame. The direction in which the pixels are moving can be determined by looking at the color scheme in figure 4.1. Pixels that are not moving are black.

4.5. Hardware optimizations

To achieve optimal performance when developing a parallel application for a GPU there are many different requirements that must be met. Section 4.5 describes these requirements and tries to explain why they must be met in order to achieve optimal performance. This section also aims to answer some of the questions stated in section 1.2. Henceforth our parallel application will be referred to as "the application".

4.5.1. Instruction performance

To process an instruction for a warp, a Streaming Multiprocessor (SM) must:

- Read the instruction operands for each work-item in the warp
- Execute the instruction
- Write the result for each work-item in the warp

Therefore, the effective instruction throughput depends on the nominal instruction throughput as well as the memory latency and bandwidth. It is maximized by:

- Minimizing the use of instructions with low throughput
- Maximizing the use of the available memory bandwidth for each category of memory
- Allowing the warp scheduler to overlap memory transactions with mathematical computations as much as possible to hide memory latency. This requires that:
    - The kernel executed by the work items has a high arithmetic intensity
    - There are many active work-items per SM

For a warp size of 32, an instruction is made of 32 operations. Therefore, if T is the number of operations per clock cycle, the instruction throughput is one instruction every 32/T clock cycles for one SM. The throughput must hence be multiplied by the number of SMs in the GPU to get throughput for the whole GPU [41].

4.5.1.1. Arithmetic instructions

For applications that only require single-precision floating point accuracy is it highly recommended to use the `float` data type and single-precision floating point mathematical functions. The single-precision `float` data type requires only half the bits compared to the double-precision floating point data type `double`. As described in section 3.2.5 single-precision floating point instructions are much faster to execute than double-precision instructions.

Many OpenCL implementations have "native" versions of most common math functions. The native functions may map to one or more native device instructions and will typically have better performance compared to the non-native functions. The increased performance usually comes at the cost of reduced accuracy. Because of this native math operators should only be used if the loss of accuracy is acceptable.

Whilst building OpenCL kernels it is possible to enable the `-cl-mad-enable` build option. This will allow `a * b + c` to be replaced by a "Multiply And Add" (MADD) instruction.

The MADD instruction computes `a * b + c` with increased performance but with reduced accuracy.

There are several other compiler optimization options available, many of them trading accuracy for performance. The `-cl-fast-relaxed-math` option implies both the `-cl-mad-enable` as well as several other compiler options. It allows optimizations for floating-point arithmetic that may violate the IEEE 754 standard and the OpenCL numerical compliance requirements. It should however be noted that not all OpenCL implementations have support for the `-cl-fast-relaxed-math` compiler option. For example, the current AMD implementation (ATI Stream SDK v2.01) does not. For more information about the compiler options, see [1].

4.5.1.2. Control flow instructions

Any flow control instruction (`if, switch, do, for, while`) can significantly impact the effective instruction throughput by causing work-items of the same warp to diverge (to follow different execution paths). If this happens, the different execution paths for the warps need to be serialized, i.e. executed one after another. When all different execution paths have been executed, the work-items converge back to the same execution path. Note that work-items within different warps can follow different execution paths without any performance penalty. To avoid warp divergence, control flow instructions should be kept to a minimum and must be designed so that all work-items within a warp follow the same execution path.

4.5.1.3. Memory instructions

A memory instruction is any instruction that reads from or writes to memory. Throughput for memory instructions to local memory is 8 clock cycles while memory instructions to global memory imply an additional memory latency of 400-600 clock cycles.

As stated in section 3.2.3.1, global memory latency can be “hidden” by the warp scheduler if there are sufficient independent arithmetic instructions that can be issued while waiting for the global memory access to complete. This does however require that there are enough unique warps to schedule within a SM. Section 4.5.3 lists several guidelines concerning how to increase the number of active warps per SM.

4.5.1.4. Synchronization instruction

The `barrier()` function will make sure that all work-items within a work-group reach the barrier before any of them execute further. Synchronizing all work-items within a work-group is sometimes needed to ensure memory consistency. Work-item synchronization using the `barrier()` function has a throughput of 8 clock cycles in the case where no work-item has to wait for any other work-items. It should be noted that synchronization instructions can make it more difficult for the warp scheduler to achieve optimal work-item scheduling.

4.5.2. Memory bandwidth

The AMD and NVIDIA architectures differ somewhat concerning the memory implementation. For both platforms though, the effective bandwidth of each memory space

depends significantly on the memory access pattern. This means that it is up to the programmer to ensure that memory within an application is accessed in a way that results in optimal performance. There are several requirements that must be met to achieve optimal memory performance in a GPU application. Section 4.5.2 covers the different types of memory available and discusses these requirements [41].

4.5.2.1. Global memory

Global memory is not cached and implies a 400-600 clock cycles memory latency. To obtain optimal performance when accessing global memory it is required to access memory according to a specific access scheme. Global memory bandwidth is used most efficiently when the simultaneous memory accesses (during the execution of a single read or write instruction) can be coalesced into a single memory transaction of 32-, 64-, or 128-bytes. For example, if 16 work-items each read one `float` data value (4 bytes) from memory, this can result in one single memory transaction if the read is coalesced, as opposed to a non-coalesced read that results in 16 memory transactions. It is not hard to imagine the impact on performance that coalesced memory accesses have, making it one of the most important factors to consider when optimizing a GPU application.

Unfortunately, it is no simple task to achieve coalesced memory accesses. The G80 architecture is very strict concerning which access patterns that result in coalesced memory assesses and which patterns that does not. There are three requirements that need to be fulfilled in order to coalesce memory accesses on the G80 architecture; 1) Work-items must access 4-, 8- or 16-byte words, 2) All 16 work-items in a half-warp must access words in the same memory segment and 3) Work-items must access words in sequence: The k:th work-item in a half-warp must access the k:th word. If these requirements are not fulfilled memory accesses will be serialized.

Global memory coalescing have been greatly improved in the Fermi architecture, allowing much more slack in the memory access patterns that result in coalesced memory accesses. On the Fermi architecture it is also possible for memory accesses to be partly coalesced. Section 5.8 describes the global memory access schemes implemented in the application in order to coalesce memory accesses.

4.5.2.2. Constant memory

Constant memory is cached so a read from constant memory will only result in a read from global memory on cache miss. If a cache hit occurs, the data is instead retrieved from the on-chip cache which has the same performance as reading from a register. By storing commonly used data in constant memory it is possible to circumvent the access latency associated with global memory accesses to increase performance.

4.5.2.3. Texture memory

Texture memory is allocated in the global memory area. Texture memory is cached just like constant memory so a read from texture memory will only result in a global memory read on cache miss. The texture cache is optimized for 2D spatial locality so work-items of the same

warp that read image addresses that are close together achieve best performance. When using the texture memory, data must be read and written using OpenCL image objects. Reading data through image objects can be beneficial during certain circumstances [41].

4.5.2.4. Local memory

Local memory is located inside the SMs and is much faster than global memory. Access to local memory is as fast as access to a register, provided that no local memory bank conflicts occur. Data stored in local memory is visible to all work-items within a work-group. To reduce global memory bandwidth in a kernel, a good technique is to modify the kernel to use local memory to hold the portion of global memory data that are heavily used in an execution phase of the kernel. The work-items within a work-group can cooperate in loading data from global to local memory. If so, it is important to synchronize all work-items after the data is loaded to ensure memory consistency. In OpenCL, work-item synchronization can be achieved using the **`barrier()`**, **`mem_fence()`**, **`read_mem_fence()`** and **`write_mem_fence()`** functions.

To achieve high bandwidth, the local memory is divided into equally sized memory modules called banks. Any memory access of n addresses that fall into separate banks can be serviced simultaneously, resulting in an effective bandwidth that is n times higher than that of a single bank. If however the n memory requests fall into the same bank, the requests must be serialized. This is called an n-way bank conflict.

On NVIDIA's GPUs the local memory is organized into 16 banks. When the work-items in a warp access local memory, the request is split into one request for the first half of the warp and a second request for the second half of the warp. Hence there can be no bank conflicts between a work-item belonging to the first half-warp and a work-item belonging to the second half-warp. On AMD's GPUs local memory is organized into 32 banks [41][42].

To avoid bank conflicts the local memory access pattern must be considered. For example, if an array of 1-byte **`char`** data values is accessed as in listing 4.2 bank conflicts will occur because each work-item in the warp read data with a stride of one byte, resulting in several threads reading the same memory bank. On NIVIDIA's GPUs 4 consecutive bytes are stored in the same bank, e.g. byte 0-3 is stored in bank 0, byte 4-7 is stored in bank 1 etc.

```
__local char local[32];
char data = local[WorkItemId];
```

Listing 4.2. A memory access pattern that results in bank conflicts.

However, accessing the same array as in listing 4.3 will result in a conflict free access because data is now accessed with a stride of 4 bytes which results in each work-item accessing separate memory banks.

```
char data = local[4 * WorkItemId];
```

Listing 4.3. A conflict free memory access pattern.

In general, it is possible to detect bank conflicts by analyzing the local memory access scheme using cyclic groups and Euler's totient function (φ(n)).

4.5.2.5. Registers

Registers are used to store data for a work-item. In general, accessing a register does not imply any latency. Register delays may however occur due to read-after-write dependencies or register bank conflicts. The compiler and the warp scheduler try to schedule instructions to avoid register bank conflicts. The best result is achieved when the number of work-items per work-group is a multiple of 64. Delays introduced by read-after-write dependencies can be ignored as soon as there are at least 192 active work-items per multiprocessor to hide them.

4.5.3. NDRange dimensions and SM occupancy

The NDRange determines how the work-items executing a kernel should be divided into work-groups. How the NDRange affects the execution time of a kernel generally depends on the kernel. It is possible to let the OpenCL implementation determine the NDRange even though that usually does not result in the best performance. The programmer is encouraged to experiment with the NDRange to find the optimal performance for each kernel.

There is a hardware limit on the number of work-items per SM and the number of work-items per work-group. If this limit is exceeded, the kernel will fail to launch. See Appendix A for the maximum number of work-items per SM for some different GPUs. A kernel will also fail to launch if the number of work-items per work-group requires too many registers or too much local memory. The total number of registers required for a work-group is equal to

$$ceil(R * ceil(T, 32), \frac{R_{max}}{32}) \tag{4.16}$$

Equation 4.16. The total number of registers required for a work-group.

Where R is the number of registers required by the kernel, R_{max} is the number of registers per SM given in Appendix A, T is the number of work-items per work-group and *ceil(x,y)* is equal to x rounded up to the nearest multiple of y. The total amount of local memory required for a work-group is equal to the sum of the amount of statically allocated local memory, the amount of dynamically allocated local memory and the amount of local memory used to pass the kernels arguments [41].

Given the total number of work-items in an NDRange, the number of work-items per work-group might be governed by the need to have enough work-items to maximize the utilization of the available computing resources. There should be at least as many work-groups as there are SMs in the device. Running only one work-group per SM will force the SM to idle during work-item synchronization and global memory reads. It is usually better to allow two or more

work-groups to be active on each SM so that the scheduler can interleave work-groups that wait and work-groups that can run. For work-groups to be interleaved, the amount of registers and local memory required per work-group must be low enough to allow for more than one active work-group.

For the compiler and the warp scheduler to achieve the best results, the number of work-items per work-group should be at least 192 and should be a multiple of 64. Allocating more work-items per work-group is better for efficient time slicing but too many work-items will exceed the available number of registers and memory and hence prevent the kernel from launching.

The ratio of the number of active warps per SM to the maximum number of active warps per SM is called the SM "occupancy". The compiler tries to maximize occupancy by minimizing the number of instructions, registry usage and local memory usage. In general there is a trade-off between occupancy and memory/registry usage which will be kernel dependent. For some kernels it will be worth sacrificing some memory to obtain a higher occupancy while for other kernels the same sacrifice will result in reduced performance. The programmer is encouraged to experiment to find the optimal trade-off.

To summarize, the optimal NDRange will be obtained by following these guidelines:

- **Required**: The number of work-items per SM must be below the hardware limit of the GPU.
- **Required**: The local memory and registry usage for a work-group must be below the hardware limit of the GPU.
- There should be at least as many work-groups as there are SMs on the device. Preferably there should be two or more work-groups per SM (local memory and registry usage will limit the maximum number of work-groups that can be run simultaneously on a SM).
- The number of work-items per work-group should be as high as possible and should be a multiple of 64. There should be at least 192 active work-items per SM to hide registry read-after-write dependencies.
- The occupancy for a SM should be as high as possible.
- The optimal NDRange for a kernel cannot be established without experimentation. The optimal configuration depends on the kernel.

4.5.4. Data transfer between host and device

The bandwidth between the device and the device memory is much higher than the bandwidth between the host and the device memory. Hence data transfers between host and device should be minimized. This can be achieved by moving code and data to the device kernels. Intermediate data structures may be created in device memory, operated on by the device and then destroyed without ever being mapped by the host or copied to host memory. Because of the overhead associated with each memory transfer it is beneficial to combine several smaller memory transfers into one large memory transfer whenever possible.

Higher performance between host and device memory is obtained for memory objects allocated in page-locked memory (also called "pinned" memory) as opposed to ordinary pageable host-memory (as obtained by `malloc()`). The page-locked memory has two main benefits: 1) Bandwidth is higher between page-locked memory and device memory than between pageable memory and device memory and 2) For some devices, page-locked memory can be mapped into the device address space. In this case there is no need to allocate any device memory and explicitly copy data between device and host memory.

In OpenCL, applications have no direct control over whether memory objects are allocated in page-locked or pageable memory. Using NVIDIA's OpenCL implementation, it is however possible to create memory objects using the `CL_MEM_ALLOC_HOST_PTR` flag and such objects are likely to be allocated in page-locked memory [41].

4.5.5. Warp-level synchronization

Work-items within a warp are implicitly synchronized and execute the same instruction at all times. This fact can be used to omit calls to the `barrier()` function to increase performance. The reader is referred to [41] for more information about warp-level synchronization.

4.5.6. General advice

OpenCL kernels typically execute a large number of instructions per clock cycle. Thus, the overhead to evaluate control-flow and execute branch instructions can consume a significant part of resources that otherwise could be used for high throughput compute instructions. For this reason loop unrolling can have a significant impact on performance.

The NVIDIA OpenCL compiler supports the compiler directive `#pragma unroll` by which a loop can be unrolled a given number of times. It is generally a good idea to unroll every major loop as much as possible. By using `#pragma unroll` it is possible to suggest the optimal number of loop unrolls to the compiler. Without the pragma the compiler will still perform loop unrolling but there is no way to control to which degree [41].

The AMD OpenCL compiler performs simple loop unrolling optimizations; however, for more complex loop unrolling, it may be beneficial to do this manually. The AMD compiler does not support the `#pragma unroll` so manual unrolling of loops must be performed to be sure that performance is optimal.

AMD's GPUs are five-wide Very Long Instruction Word (VLIW) architectures. For applications executing on AMD's GPUs, the use of vector types within the code can lead to substantially greater efficiency. The AMD compiler will try to automatically vectorize all instructions and the AMD profiler can display to which degree this is possible. Manual vectorization is however the best way to ensure that optimal performance is obtained for all data types. The NVIDIA architecture is scalar so vectorizing data types have no impact on performance [42].

5. Results

To be able to compare the different GPU architectures as well as the impact of the optimizations described in section 4 a set of benchmarks have been compiled. The benchmarks have been carried out on three different system configurations. The system configurations are listed in table 5.1.

	Configuration 1 (CFG1)	**Configuration 2 (CFG2)**	**Configuration 3 (CFG3)**
CPU	Intel Core 2 Duo 6700 @ 2.66 GHz	Intel Core 2 Duo 8400 @ 3.00 GHz	Intel Core i5 750 @ 2.67 GHZ
RAM	4x1GB DDR2 PC2-6400 @ 800 MHz	2x2GB DDR2 PC2-8500 @ 1066 MHz	2x2GB DDR3 PC3-12800 @ 1600 MHz
Hard drive	Western Digital Raptor 74 GB @ 10000 RPM	Samsung Spinpoint F1 1 TB @ 7200 RPM	Samsung Spinpoint F3 1 TB @ 7200 RPM
PSU	700W	650W	750W
Operating system	Microsoft Windows 7 Professional 64-bit	Microsoft Windows Vista Business 64-bit	Microsoft Windows 7 Professional 64-bit
GPU	**NVIDIA Geforce 8800 GTS (G80)**	**AMD Radeon 4870 (RV700)**	**NVIDIA Geforce GTX 480 (Fermi)**
GPU Driver	NVIDIA WHQL-certified Geforce driver v197.13	Catalyst version 9.11 Driver version 8.690.0.0	NVIDIA WHQL-certified Geforce driver v197.41
GPU SDK	NVIDIA CUDA Toolkit v3.0 64-bit	ATI Stream SDK v2.01 64-bit	NVIDIA CUDA Toolkit v3.0 64-bit

Table 5.1. The three different system configurations used for benchmarking the OpenCL application.

Section 5 starts with presenting the benchmarks for our sequential implementation followed by the benchmarks for our naïve parallel implementation. The parallel application optimizations will then be presented one by one with comments regarding their impact on performance. Section 5 will be concluded by a table summarizing the results and showing the relative performance impact of all the optimizations.

The benchmarks have been measured using a ~10 second (240 frames) "FullHD" 1920x1080 video at 23.98 FPS. Video decoding is handled by the CPU using the Open Source Computer Vision (OpenCV) library version 2.0 [43]. To be able to play video at 23.98 FPS in real time the application must decode a frame, process it and display on the screen in less than 41.7 ms.

5.1. A naïve sequential implementation on the CPU

Our naïve sequential LK-method implementation (see listing 4.1) is extremely slow and requires ~21 seconds to process a frame on CFG1. The results are slightly better on CFG2 and CFG3. This implementation is indeed naïve and much could be done to improve it. According to [43] a speedup of at least a magnitude should be possible if the application is optimized for

the CPU. CPU optimization is however outside the scope of this thesis. The benchmarks for the sequential implementation can be seen in table 5.2.

	CFG1 (ms)	CFG2 (ms)	CFG3 (ms)
Average processing time per frame	21344	18750	16896

Table 5.2. Processing times for the naïve sequential LK-method implementation.

5.2. A naïve parallel implementation on the GPU

Our naïve parallel implementation was our first attempt creating a GPU accelerated application. Parallelizing the LK-method implementation showed a great performance increase compared to the sequential implementation. On CFG1 the performance increased 42x. CFG3 showed an even higher performance increase of 1374x.

As previously described in section 4.4.1, our first parallel implementation was divided into five kernels; a grayscale converter, a kernel for matrix operations, a kernel for convolution, a least-squares method implementation and a kernel for visualizing the motion vectors. After considering all the performance requirements listed in section 4.5.3 and experimenting with the NDRange, a work-group size of 64x8 was found to be the best configuration for CFG1 and CFG3. This was however not possible on CFG2 due to hardware limits. An NDRange of 32x8 was used for CFG2.

The Fermi GPU excels immediately and the naïve implementation is already fast enough to execute in real time. This is however not possible on neither the G80 nor the RV700. The processing times of the naïve parallel implementation can be seen in table 5.3.

	CFG1 (avg ms / frame)	CFG2 (avg ms / frame)	CFG3 (avg ms / frame)
Grayscale	4.13	0.84	0.17
Matrix	0.65	0.77	0.19
Convolution	12.93	0.92	0.2
Least-squares	402.71	23.17	3.76
Visualize	6.53	3.01	0.39
Device time (all kernels)	465.29	49.5	8.55
Host time (entire application)	46.29	8.22	3.75
Total processing time	511.58	57.72	12.3

Table 5.3. Processing times for the naïve parallel implementation.

5.3. Local memory

As stated in section 4.5.2.1, global memory accesses have a latency of 400-600 clock cycles making them very costly. By introducing local memory buffers in the least-squares kernel it was possible to reduce the number of global memory reads from 81 per pixel to just 2 reads per pixel. This means that the number of global memory reads for the least-squares kernel was reduced by 98%.

As stated in section 4.5.2.4, it is important to consider the memory access scheme when accessing local memory to avoid bank conflicts. For example, a 2-way bank conflict will reduce the local memory bandwidth by half. Several different local memory access schemes were evaluated. The final local memory access scheme resulted in 20x less bank conflicts than the naïve one. Optimizing the local memory access scheme had most impact on the G80 architecture.

The reduced usage of global memory should in theory yield a great performance increase because of global memory latency. Indeed, table 5.4 shows that this optimization results in a very large performance increase on CFG1. Due to the improved caching in the Fermi architecture no notable change can be seen on CFG3. On CFG2, the increased use of local memory has a negative effect on performance. After modifying the kernel the increased registry and local memory usage required that the NDRange for the least-squares kernel was reduced to 16x24 for CFG1 and to 8x8 for CFG2.

	CFG1 (avg ms / frame)	**CFG2 (avg ms / frame)**	**CFG3 (avg ms / frame)**
Grayscale	4.28	0.84	0.17
Matrix	0.72	0.77	0.19
Convolution	12.86	0.92	0.2
Least-squares	21.24	154.12	3.43
Visualize	6.67	3.01	0.39
Device time (all kernels)	84.37	179.51	8.25
Host time (entire application)	44.97	9.83	3.79
Total processing time	129.34	189.34	12.04

Table 5.4. Processing times after the local memory optimizations.

5.4. Constant memory

As stated in section 4.5.2.2, constant memory is cached. Global memory reads that result in a cache hit are as fast as reads from a register. Storing commonly used data in the constant memory should reduce processing time significantly due to the large performance penalty associated with global memory accesses.

The variables containing the kernels used to perform convolution was stored in constant memory. Since all work-items use the same three kernels for calculating spatial and temporal derivatives, the cache hit rate should be high. Moving the convolution kernels to constant memory reduce the total number of global accesses by 4 per convolution (12 in total). This reduced the number of global reads by approximately 50% for the OpenCL kernel handling convolution. This reduction is not as high as for the local memory optimization in the least-squares kernel but should still be high enough to result in a visible performance increase.

The use of constant memory results in a performance increase on CFG1. No notable change in performance can be seen for CFG3, most likely because of the improved memory caching in the Fermi architecture. CFG2 displays a minor performance increase. A peculiar side effect of the use of constant memory is that the host time decreases by 74% on CFG1! The results are presented in table 5.5.

	CFG1 (avg ms / frame)	CFG2 (avg ms / frame)	CFG3 (avg ms / frame)
Grayscale	4.28	0.84	0.17
Matrix	0.72	0.77	0.19
Convolution	1	0.92	0.19
Least-squares	18.78	129.01	3.43
Visualize	6.66	3.0	0.39
Device time (all kernels)	45.36	155.67	8.25
Host time (entire application)	11.48	9.65	3.62
Total processing time	56.84	165.32	11.87

Table 5.5. Processing times after the constant memory optimizations.

5.5. Native math

As described in section 4.5.1.1, many OpenCL implementations have "native" versions of most common math functions. To increase performance in the application all math functions were replaced by the native versions. For example, floating point division was replaced with the `native_divide` function and the exponential functions were replaced with `native_pow`.

The introduction of the native OpenCL instructions did not affect the performance of the application on any of the architectures. One reason for this is that the compiler already utilized the native instructions everywhere possible. The use of native instructions could however result in a performance increase on other platforms with other compilers why the native instructions were left in the application for eventual future use.

5.6. Merging kernels

To increase the arithmetic intensity and allow the Grayscale, Matrix Subtraction and Convolution kernels to share frame data in local memory these three kernels were combined into one single large kernel called "GSC". With fewer kernels to start, the host overhead time associated with launching a kernel should be reduced. Merging several smaller kernels into one large kernel is however no guarantee for increased performance. Because a larger kernel requires more memory, fewer work-items can execute the kernel at the same time. This might lead to reduced performance and problems in scheduling the kernel. The optimal kernel size is therefore a trade off and the programmer is required to experiment to find this optimal size for each kernel.

In the GSC kernel, the work-items in the different work-groups cooperate to load global memory data into local memory, much like in the local optimization described in section 5.3. Loading data in this way results in fewer global memory accesses than before. One drawback of using local memory this way is that synchronization instructions must be introduced to ensure local memory consistency. As stated in section 4.5.1.4, explicit work-item synchronization can make it more difficult for the scheduler to achieve optimal work-item scheduling. The GSC kernel was configured with an NDRange of 64x8 for CFG1 and CFG3. This was however not possible on CFG2 so the NDRange for CFG2 was set to 32x8.

Merging the kernels had a positive impact on the host time on all three hardware configurations. However, almost no change at all could be seen for the device time on CFG1 or CFG3, most likely because the arithmetic intensity before the merge was already high enough to hide the memory latencies. A slight performance decrease can be noticed for CFG2. See table 5.6 for the processing times after the kernels were merged.

	CFG1 (avg ms / frame)	**CFG2 (avg ms / frame)**	**CFG3 (avg ms / frame)**
GSC	12.31	26.54	0.66
Least-squares	19.04	128.46	3.41
Visualize	6.56	2.92	0.39
Host to device memory transfer time	3.25	7.6	1.52
Device to host memory transfer time	4.43	9.93	1.64
Device time (all kernels)	45.66	175.34	7.61
Host time (entire application)	2.84	4.34	1.67
Total processing time	48.5	179.68	9.28

Table 5.6. Processing times after the kernel merge optimization.

5.7. Non-pageable memory

As stated in section 4.5.4, there are several benefits of using non-pageable memory as opposed to pageable memory. Memory bandwidth is higher between non-pageable memory and device memory than between pageable memory and device memory. For each frame to be processed by the device, the frame must first be decoded by the host, transferred from host to device memory, processed by the device, transferred back from device to host memory and then displayed on the screen. Each video frame is approximately $1920 * 1080 * 3 \approx 5.93$MB so before a frame is processed and can be displayed to the user ~11.86 MB data must be transferred between the host and the device.

Using pageable memory when transferring data between host and device memory results in a memory bandwidth of around 1.82 GB/s on CFG1 and 3.90 GB/s on CFG3. To be able to copy data to and from the host non-pageable memory to device memory, the data must first be copied from pageable memory to non-pageable memory on the host. Copying data between different parts of host memory is a synchronous operation so execution must be stalled until all data is available in non-pageable memory. This stall introduces some overhead time in the application.

The usage of non-pageable host memory results in increased memory bandwidth, 2.49 GB/s on CFG1 and 5.70 GB/s on CFG3. However, the increase in host processing time associated with copying data between pageable and non-pageable memory overshadows the decreased transfer time due to increased bandwidth, see table 5.7.

	CFG1 (avg ms / frame)	CFG2 (avg ms / frame)	CFG3 (avg ms / frame)
GSC	12.45	26.64	0.69
Least-squares	19.58	129.66	3.62
Visualize	6.56	2.92	0.39
Host to device memory transfer time	2.38	7.6	1.04
Device to host memory transfer time	3.27	9.75	1.02
Device time (all kernels)	51.73	176.48	6.76
Host time (entire application)	9.7	10.19	3.87
Total processing time	61.43	186.67	10.63

Table 5.7. Processing times after the non-pageable memory optimization.

5.8. Coalesced memory accesses

As stated in section 4.5.2.1, coalescing global memory accesses can greatly increase the performance of a parallel application. There are three requirements that need to be fulfilled in order to coalesce memory accesses on the G80 architecture; 1) Work-items must access 4-, 8- or 16-byte words, 2) All 16 work-items in a half-warp must access words in the same memory segment and 3) Work-items must access words in sequence: The k:th work-item in a half-warp must access the k:th word. If these requirements are not fulfilled none of the memory accesses will be coalesced. On the Fermi architecture it is possible to have partly coalesced memory accesses, even if the access pattern does not fulfill all three requirements. For example, 16 global accesses to two different memory segments would be coalesced into two global accesses on Fermi while it would be 16 (non-coalesced) global accesses on the G80.

When each work-item only access one single 4-byte word (for example a `float` data value) coalescing is trivial. The work-items unique global NDRange ID can be used to calculate the global memory address that each work-item should access. The first requirement to achieve coalesced memory accesses is fulfilled because the work items access 4-byte words. The second requirement is fulfilled if the work-group x-dimension size is evenly divisible by 16. The third requirement is fulfilled since the NDRange ID is used to divide the work-items into warps (i.e. work-item 0-31 will form a warp). Global memory accesses according to this scheme is implemented for all global memory writes in the GSC and least-square kernels and all global memory reads in the Visualize kernel. See figure 5.1 for an example of the memory access scheme.

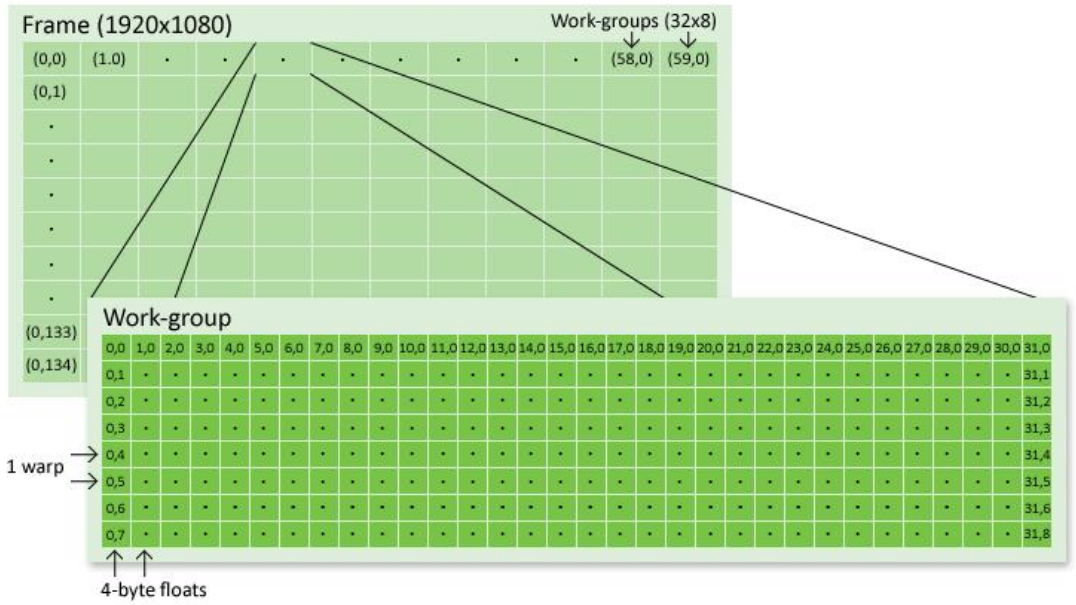

Figure 5.1. Example of an access scheme for a work-group with size x = 32 and y = 8. The light green boxes display which work-group that reads which part of the image frame. Each work-item reads the memory position that map to its local ID within the work-group.

It is much more difficult to coalesce memory accesses when the work-items do not access 4-, 8- or 16-byte words. The work-items in the GSC and Visualize kernel both access 3x 1-byte words (the three 1-byte RGB values for each pixel in the frame). The GSC kernel reads 1-byte words from global memory and the Visualize kernel writes 1-byte words to global memory. To coalesce these memory accesses a more advanced access scheme is required than in the trivial case.

To fulfill the first requirement for coalesced memory accesses the work-items in each work-group read or write an **`uchar4`** (4x 1-byte RGB values) from/to global memory. The second requirement implies that the global memory addresses need to be aligned to 16x 4-byte words. Memory address alignment together with the third requirement is fulfilled by padding the accessed memory area and adjusting the work-group x-dimension size. The work-group x-dimension and the padding must be chosen so that they solve equation (5.1) and (5.2)

$$\frac{(\mathrm{WGD_x} + \mathrm{P}_x) * 3}{4} \equiv 0 \ (\mathrm{mod}\ 16) \tag{5.1}$$

$$(\mathrm{WGD_x} + \mathrm{P}_x) * 3 * WG_{xid} \equiv 0 \ (\mathrm{mod}\ 64) \tag{5.2}$$

where $\mathrm{WGD_x}$ is the work group x-dimension size, $\mathrm{P_x}$ is the padding in the x-direction and $\mathrm{WG_{xid}}$ is the work group x-dimension ID.

When $\mathrm{WGD_x}$ and $\mathrm{P_x}$ are chosen to solve equation (5.1) and (5.2) all the requirements will be fulfilled and the global memory accesses will be coalesced. If the padded memory has more **`uchar4`** words than there are work-items, the work-items need to read more than one **`uchar4`**

each. In that case the work-items read a second **`uchar4`** with a set memory offset. See figure 5.2 for an example of this memory access scheme.

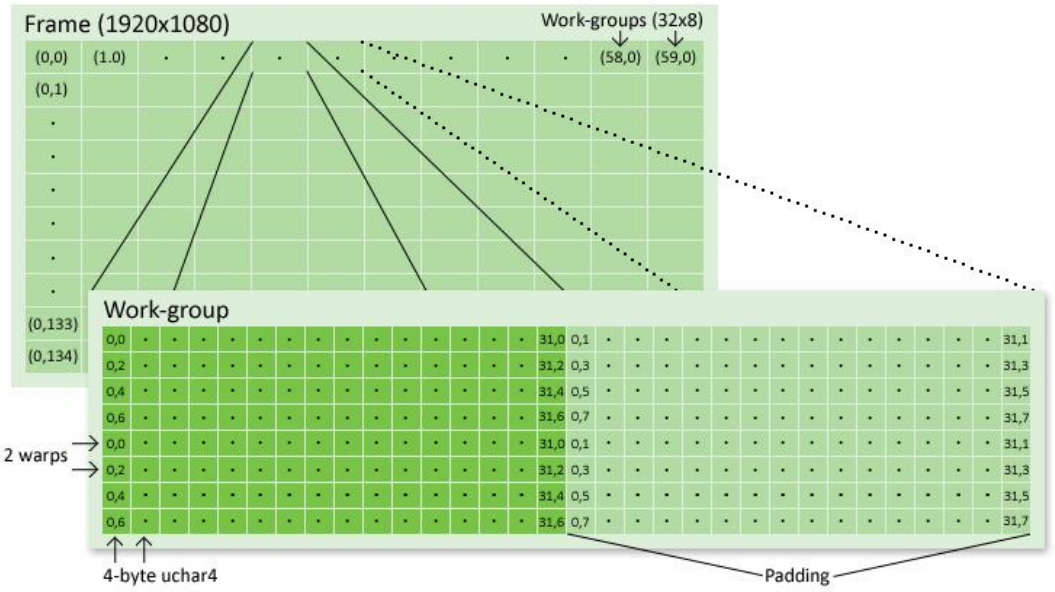

Figure 5.2. Example of an access scheme for a work-group with size x = 32, y = 8 and padding = 32.The light green boxes display which work-group that reads which part of the image frame. Each work-item reads the memory positions that map to its local ID within the work-group.

As can be seen in table 5.8, coalesced global memory accesses only increase performance on CFG1. According to the NVIDIA visual profiler all global memory accesses are 100% coalesced for the GSC and visualize kernels. For the least-squares kernel only the global memory writes are 100% coalesced. As stated in section 3.2.5, the Fermi architecture has special hardware to optimize global memory accesses; it is however not possible to see to which degree memory accesses are coalesced. On CFG1, the coalesced memory access scheme greatly reduces the total number of global memory accesses. However, because the advanced access scheme pads the memory area, the total number of 4-byte words accessed is somewhat increased. This result in extra memory accesses which reduces performance on CFG2 and CFG3 architecture compared to the trivial memory access scheme without padding.

	CFG1 (avg ms / frame)	CFG2 (avg ms / frame)	CFG3 (avg ms / frame)
GSC	6.33	102.32	0.78
Least-squares	19.69	128.7	3.65
Visualize	3.11	5.85	0.49
Host to device memory transfer time	2.37	7.57	1.04
Device to host memory transfer time	3.29	9.67	1.02
Device time (all kernels)	34.78	254.1	6.98
Host time (entire application)	9.4	11.74	3.74
Total processing time	44.18	265.84	10.72

Table 5.8. Processing times after the coalesced memory optimizations.

5.9. Using the CPU as both host and device

Using AMD's OpenCL implementation it is possible to use the CPU as both host and device. This will run the entire application on the CPU, not utilizing the GPU at all. Using the CPU as both host and device on CFG2, using only pageable memory, yields an average processing time of 781 ms. Comparing to the naïve sequential implementation on CFG2 this is a decrease in execution time of 96%. When comparing to the naïve parallel implementation on CFG1 the decrease in processing time obtained by utilizing a GPU is only ~33%. On the other hand, if comparing to Fermi utilizing a GPU decreases processing time by 98%. Note that NVIDIA's OpenCL implementation does not yet support using the CPU as both host and device.

5.10. Summary of results

To summarize, CFG3 has the best performance of the three system configurations evaluated. Optimizing the parallel application had a positive yet limited impact on performance for CFG3. For CFG1 the optimizations made a huge difference and significantly reduced the processing time of the application. For CFG2, all optimizations proved to have a negative effect on performance which results in the naïve implementation having the best performance. Table 5.9 summarizes the device times for the three configurations with regard to application optimizations and table 5.10 summarizes the total time (device and host time). Figure 5.2 plots the device processing times and figure 5.3 plots the total processing times for the respective application versions.

When analyzing the results from table 5.9 and 5.10 it can be seen that the best performance for CFG1 is obtained by using the naïve implementation as a base and then applying all optimizations except the use of non-pageable memory. For CFG3 the best performance is obtained by applying all optimizations except non-pageable memory and coalesced memory. Table 5.11 lists the application processing times measured for CFG1, CFG2 and CFG3 using these optimization configurations.

Device only processing times									
Application version	**CFG1 (G80)**			**CFG2 (RV700)**			**CFG3 (Fermi)**		
	Exec. time (ms)	**Rel. time change (%)**	**Time change rel. naïve (%)**	**Exec. time (ms)**	**Rel. time change (%)**	**Time change rel. naïve (%)**	**Exec. time (ms)**	**Rel. time change (%)**	**Time change rel. naïve (%)**
Naïve	465.29	-	-	49.5	-	-	8.55	-	-
Local mem.	84.39	-82	-82	179.51	263	263	8.25	-4	-4
Constant mem.	45.36	-46	-90	155.67	-13	214	8.25	0	-4
Kernel merge	45.66	1	-90	175.34	13	254	7.61	-8	-11
Non-pageable mem.	51.73	13	-89	176.48	1	257	6.76	-11	-21
Coalesced mem.	34.78	-33	-93	254.1	44	413	6.98	3	-18

Table 5.9. Summary of device processing times for the three configurations.

Total processing times (host + device)									
Application version	**CFG1 (G80)**			**CFG2 (RV700)**			**CFG3 (Fermi)**		
	Exec. time (ms)	**Rel. time change (%)**	**Time change rel. naïve (%)**	**Exec. time (ms)**	**Rel. time change (%)**	**Time change rel. naïve (%)**	**Exec. time (ms)**	**Rel. time change (%)**	**Time change rel. naïve (%)**
Naïve	511.58	-	-	57.72	-	-	12.3	-	-
Local mem.	129.34	-75	-75	189.34	228	228	12.04	-2	-2
Constant mem.	56.84	-56	-89	165.32	-13	186	11.87	-1	-3
Kernel merge	48.5	-15	-91	179.68	9	211	9.28	-22	-25
Non-pageable mem.	61.43	27	-88	186.67	4	223	10.63	15	-14
Coalesced mem.	44.18	-28	-91	265.84	42	361	10.72	1	-13

Table 5.10. Summary of the total processing times for the three configurations.

Best performance obtained						
	CFG1 (G80)		**CFG2 (RV700)**		**CFG3 (Fermi)**	
	Exec. time (ms)	**Time change rel. naïve (%)**	**Exec. time (ms)**	**Time change rel. naïve (%)**	**Exec. time (ms)**	**Time change rel. naïve (%)**
Device time	37.05	-92	49.5	0	7.48	-13
Host time	1.45	-97	8,22	0	1.2	-68
Total time	38.51	-93	57.72	0	8.68	-29

Table 5.11.The best processing times obtained for the three configurations.

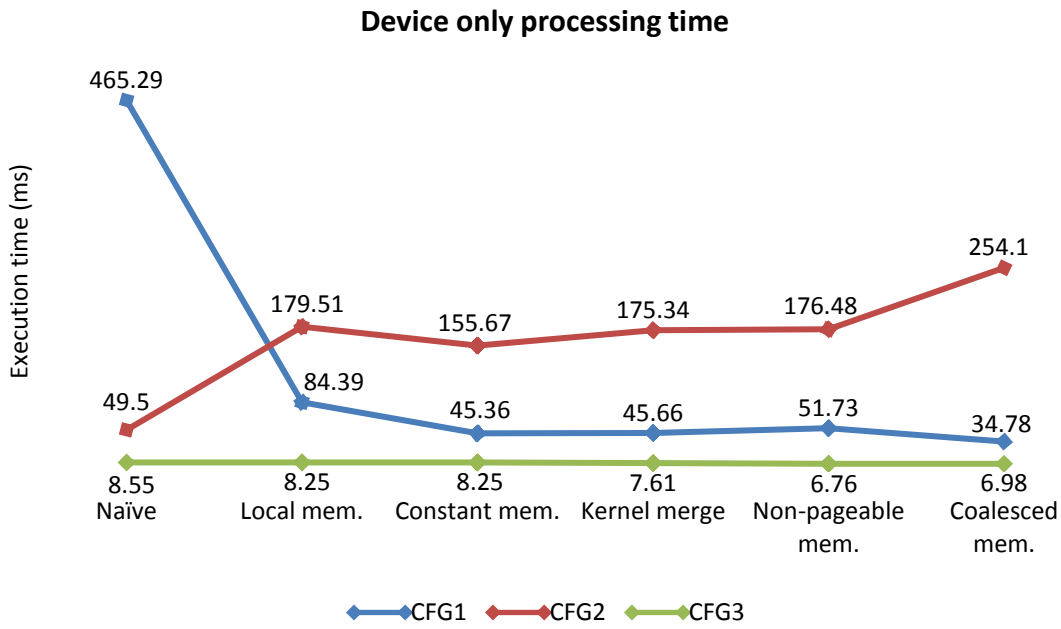

Figure 5.2. Device only processing times for the three GPUs

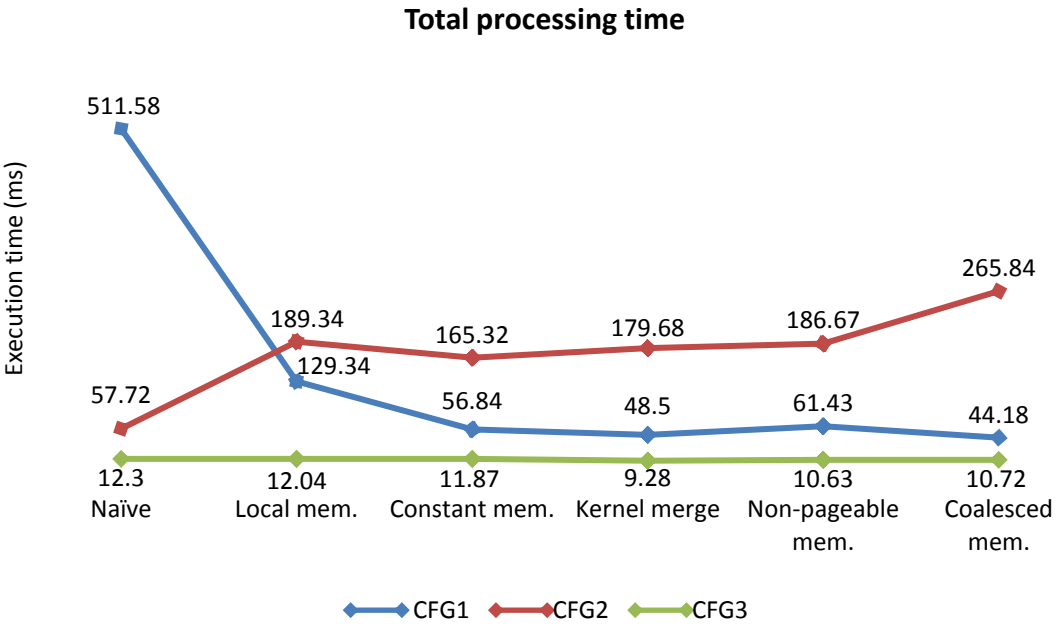

Figure 5.3. Total processing time for the three GPUs

6. Conclusions

As the results in section 5 shows, the new NVIDIA Fermi GPU excels in all aspects of GPGPU. Fermi delivers higher performance than previous generation GPU architectures and is at the same time easier to program. As the results show the application scales very well. Replacing a G80-based GPU with a Fermi-based GPU will surely increase performance for almost any GPU application. Before the Fermi architecture, there was much to gain from optimizing GPU applications. With Fermi there are still some gains to be made from optimizing but the performance is already quite good with a naïve implementation, much due to the improved caching and memory architecture. It should be noted that Fermi requires more power and generates more heat than previous generation GPUs. Heat and power dissipation is however outside the scope of this thesis.

Local memory bank conflicts and non-coalesced memory accesses have been common and tedious problems up until now. With Fermi's improved memory architecture, it is easier to coalesce global memory accesses and less knowledge about the hardware is required from the programmer. However, even with Fermi's improved memory architecture, bank conflicts and non-coalesced memory accesses can still lower performance. To achieve maximum performance the programmer must therefore still have some knowledge about the hardware to be able to address these issues. Hopefully, in coming GPU architectures the programmer will not have to consider memory access patterns at all.

On the G80 architecture the benefits from optimizing the application were immense. Optimized use of local and constant memory resulted in large performance increases. Coalescing global memory accesses also proved to be one of the most important factors to consider to achieve optimal performance. These optimizations are however not possible to implement without a deep understanding of the hardware. It is not easy to achieve maximum performance on the G80 architecture. Optimizing memory access patterns can be tedious work, especially if the original algorithm does not map very to the GPU execution model. With the right optimizations though, the performance of the G80 GPU did exceed that of the RV700 GPU, a quite remarkable result since the G80 was released two years prior to the RV700.

The RV700 architecture is not well suited for GPGPU. The AMD OpenCL implementation does not expose the local memory of the RV700 to the programmer. Instead, local memory is emulated in global memory. As a result of this, local memory optimizations results in a performance decrease on the RV700. Instead of data being copied to on-chip memory, it is only copied to another part of the global memory – an utterly pointless operation. The only optimization that showed a positive impact on performance was the use of constant memory. The use of constant memory does however only result in increased performance on the RV700 if local memory is utilized. Hence, the naïve version of the application has the best performance on the RV700.

None of the hardware configurations did benefit from the use of non-pageable memory. The requirement to copy data between pageable and non-pageable memory on the host system did result in overhead time that overshadowed the increased host to device bandwidth. Usage of non-pageable memory could however prove to increase performance should more data be transferred every frame.

OpenCL is a stable and competent parallel programming framework. The fact that the framework is supported by all major GPU manufacturers (AMD and NVIDIA) as well as the possibilities to use the framework on most major operating systems are strong incentives. Yet, there are some differences between the OpenCL implementations from the different GPU manufactures. Only one operating system (Microsoft Windows) was used in this thesis but even then some differences between the implementations could be noted. For example, it is possible to use the CPU as both host and device when using AMD's OpenCL implementation while this is not possible using NVIDIA's. Other differences between the OpenCL implementations are support for compiler options like `-cl-fast-relaxed-math`, support for compiler directives like `#pragma unroll` and support for OpenCL image objects.

As long as both AMD and NVIDIA prioritize their own flagship frameworks (ATI Stream and CUDA) their OpenCL implementations will be slightly behind. The NVIDIA implementation of OpenCL does not yet fully benefit from all the improvements in the Fermi architecture even though this will probably be remedied in future implementations. One can speculate that the performance of OpenCL is slightly worse than that of ATI Stream or CUDA. Even though OpenCL is platform independent maximum performance will not be obtained without knowledge about the underlying software and hardware platform.

To summarize, a Fermi GPU is well worth the investment for any GPGPU project. The slightly higher initial cost will be motivated many times over by the fact that performance is much better and that much less effort is required from the programmer. OpenCL is a stable and competent framework well suited for any GPGPU project that would benefit from the increased flexibility of software and hardware platform independence. If performance is more important than flexibility, the CUDA and ATI Stream frameworks might be better alternatives.

7. Future work

Because NVIDIA's OpenCL implementation do not fully support all of the latest features of the Fermi architecture it was not possible to explore the full feature range of Fermi. In the future it will hopefully be possible to fully explore all aspects of the Fermi architecture using OpenCL. The OpenCL performance was only measured on the Microsoft Windows platform. OpenCL implementations exist on both Linux and Mac OS X as well as on other platforms and operating systems. It would be interesting to compare the relative OpenCL performance between different operating systems. Another interesting aspect would be to compare the OpenCL performance to that of CUDA and ATI Stream.

The LK-method implementation could be optimized further. For example, the global memory reads in the least-squares kernels are not coalesced to 100%. Another optimization could be to store the least-square kernel input data in OpenCL image objects which might increase performance. Further optimizations could include moving the video decoding from the CPU to the GPU. Utilizing the GPU for video decoding should reduce the amount of data transferred between the host and the device and increase performance.

In this thesis, the focus has been on optimizations for NVIDIA GPU hardware. There are however other GPU manufacturers and other types of devices that can be used with OpenCL. It would be very interesting to see if AMD's newest GPU hardware architecture "Cypress" can match the performance of the Fermi architecture. It would also be interesting to see how much the performance of the OpenCL CPU version of the LK-method implementation could be improved using CPU optimizations.

References

[1] The Khronos Group. (2010) The OpenCL Specification. http://www.khronos.org/registry/cl/specs/opencl-1.0.29.pdf. Accessed 2010-05-28

[2] NVIDIA Corporation. (2010) NVIDIA's Next Generation CUDA Compute Architecture: Fermi. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf. Accessed 2010-05-28

[3] Kirk, D. B. and Hwum, W. W. (2010) Programming Massively Parallel Processors. Burlington: Elsevier. ISBN 9780123814722.

[4] ATI Stream SDK v2.01 Performance and Optimization. http://developer.amd.com/gpu/ATIStreamSDK/assets/ATI_Stream_SDK_Performance_Notes.pdf. Accessed 2010-06-02.

[5] GPGPU.org. GPGPU Developer Resources. http://gpgpu.org/developer. Accessed 2010-05-28.

[6] The Khronos Group. Open Standards, Royalty Free, Dynamic Media Technologies. http://www.khronos.org. Accessed 2010-05-28.

[7] Lucas, B. D. and Kanade, T. (1981) An iterative image registration technique with an application to stereo vision. In *Proceedings of Imaging understanding workshop*, pp 121 - 130.

[8] Marzat, J., Dumortier, Y. and Ducrot, A. (2008) Real-Time Dense and Accurate Parallel Optical flow using CUDA. http://julien.marzat.free.fr/2008_Stage_Ingenieur_INRIA/WSCG09_Marzat_Dumortier_Ducrot.pdf. Accessed 2010-05-28.

[9] Besnerais, G. L. and Champagnat, F. Dense optical flow estimation by iterative local window registration. In *Proceedings of IEEE ICIP05*. September, 2005, Italy, Genova. Vol. 1.

[10] Bruhn, A., Joachim, W. and Schnörr, C. (2003) Lucas/Kanade Meets Horn/Schunck: Combining Local and Global Optic Flow Methods. http://www.mia.uni-saarland.de/Publications/bruhn-ijcv05c.pdf. Accessed 2010-05-28.

[11] England, J.N. (1978) A system for interactive modeling of physical curved surface objects. In *Proceedings of* SIGGRAPH 78. 1978. pp 336-340.

[12] Potmesil, M. and Hoffert, E.M. (1989) The Pixel Machine: A Parallel Image Computer. In *Proceedings of SIGGRAPH 89.* 1989. pp 69-78.

[13] Rhoades, J., Turk, G., Bell, A., State, A., Neumann, U. and Varshney, A. (1992) Real-Time Procedural Textures. In *Proceedings of Symposium on Interactive 3D Graphics.* 1992. pp 95-100.

[14] Trendall, C. and Steward, A.J. (2000) General Calculations using Graphics Hardware, with Applications to Interactive Caustics. In *Proceedings of Eurographics Workshop on Rendering*. 2000. pp 287- 298.

[15] Olano, M. and Lastra, A. (1998) A Shading Language on Graphics Hardware: The PixelFlow Shading System. In *Proceedings of SIGGRAPH*. 1998. pp 159-168.

[16] Peercy, M.S., Olano, M., Airey, J. and Ungar, P.J. (2000) Interactive Multi-Pass Programmable Shading. In *Proceedings of SIGGRAPH.* 2000. pp 425-432.

[17] Proudfoot, K., Mark, W.R., Tzvetkov, S. and Hanrahan, P. (2001) A Real-Time Procedural Shading System for Programmable Graphics Hardware. In *Proceedings of SIGGRAPH*. 2001. pp 159-170.

[18] Carr, N.A., Hall, J.D. and Hart, J.C. (2002) The Ray Engine. In *Proceedings of SIGGRAPH / Eurographics Workshop on Graphics Hardware*. 2002.

[19] Lengyel, J., Reichert, M., Donald, B.R. and Greenberg, D.P. (1990) Real-Time Robot Motion Planning Using Rasterizing Computer Graphics Hardware. In *Proceedings of SIGGRAPH*. 1990. pp 327-335.

[20] Hoff, K.E.I., Culver, T., Keyser, J., Lin, M. and Manocha, D. (1999) Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware. In *Proceedings of SIGGRAPH*. 1999. pp 277- 286.

[21] Eyles, J., Molnar, S., Poulton, J., Greer, T. and Lastra, A. (1997) PixelFlow: The Realization. In *Proceedings of SIGGRAPH / Eurographics Workshop on Graphics Hardware. 1997. pp* 57-68.

[22] Kedem, G. and Ishihara, Y. (1999) Brute Force Attack on UNIX Passwords with SIMD Computer. In *Proceedings of The 8th USENIX Security Symposium.* 1999.

[23] Bohn, C.-A. (1998) Kohonen Feature Mapping Through Graphics Hardware. In *Proceedings of 3rd Int. Conference on Computational Intelligence and Neurosciences*. 1998.

[24] GPGPU.org. History of GPGPU. http://gpgpu.org/oldsite/data/history.shtml. Accessed 2010-05-28.

[25] NVIDIA Corporation. NVIDIA CUDA Zone. http://www.nvidia.com/object/cuda_home_new.html. Accessed 2010-05-28.

[26] History of GPU (Graphics Processing Unit). http://strojstav.com/tag/history-of-gpu-graphics-processing-unit/. Accessed 2010-05-28.

[27] Stanford University. BrookGPU. http://graphics.stanford.edu/projects/brookgpu/index.html. Accessed 2010-05-28.

[28] Stanford University. Folding@home. http://folding.stanford.edu/. Accessed 2010-05-28.

[29] Advanced Micro Devices. A Brief History of General Purpose (GPGPU) Computing. http://ati.amd.com/technology/streamcomputing/gpgpu_history.html. Accessed 2010-05-28.

[30] Real World Technologies. NVIDIA's GT200: Inside a Parallel Processor. http://www.realworldtech.com/page.cfm?ArticleID=RWT090808195242&p=1. Accessed 2010-05-28.

[31] Real World Technologies. Inside Fermi: NVIDIA's HPC Push. http://www.realworldtech.com/page.cfm?ArticleID=RWT093009110932. Accessed 2010-05-28.

[32] IEEE Xplore. 754-2008 IEEE Standard for Floating-Point Arithmetic. http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?reload=true&arnumber=4610935. Accessed 2010-05-28.

[33] Burton, A. and Radford, J. (1978) Thinking in Perspective: Critical Essays in the Study of Thought Processes. Routledge. ISBN 0416858406.

[34] Warren, D. H. and Strelow, E. R. (1985) Electronic Spatial Sensing for the Blind: Contributions from Perception. Springer. ISBN 9024726891.

[35] Horn, B. K. P. and Schunck, B. G. (1981) Determining optical flow. Artificial Intelligence. Vol 17, pp 185. 1981.

[36] Farnebäck, G. (2000) Fast and Accurate Motion Estimation Using Orientation Tensors and Parametric Motion Models. In *15th International Conference on Pattern Recognition*. 2000. Vol. 1, pp. 1135.

[37] Haiying, L., Rama, C. and Azriel, R. (2003) Accurate dense optical flow estimation using adaptive structure tensors and a parametric model. In *IEEE Trans. Image Processing*. October, 2003. Vol. 12, pp. 1170.

[38] Tomasi, C. and Kanade, T. (1991) Detection and Tracking of Point Features. Carnegie Mellon University Technical Report CMU-CS-91-132. April, 1991.

[39] Shi, J. and Tomasi, C. (1994) Good Features to Track. In *IEEE Conference on Computer Vision and Pattern Recognition*. 1994. pp. 593-600.

[40] Bodily, J. M. (2009) An Optical Flow Implementation Comparison Study. http://contentdm.lib.byu.edu/ETD/image/etd2818.pdf. Accessed 2010-05-28.

[41] NVIDIA Corporation. NVIDIA OpenCL Programming Guide for the CUDA Architecture. http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_OpenCL_ProgrammingGuide.pdf. Accessed 2010-05-28.

[42] Advanced Micro Devices. ATI Stream SDK v2.01 Performance and Optimization. http://developer.amd.com/gpu/ATIStreamSDK/assets/ATI_Stream_SDK_Performance_Notes.pdf. Accessed 2010-05-28.

[43] Open Source Computer Vision library. http://opencv.willowgarage.com/. Accessed 2010-05-28.

[44] Anguita, M., Díaz, J., Ros, E. and Fernández-Baldomero, F. J. Optimization Strategies for High-Performance Computing of Optical-Flow in General-Purpose Processors. In *IEEE Transactions On Circuits And Systems For Video Technology*. October, 2009. Vol. 19, no. 10.

[45] GPUReview.com. Video Card Reviews and Specifications. http://www.gpureview.com/. Accessed 2010-05-28.

[46] NVIDIA Corporation. NVIDIA Geforce Family. http://www.nvidia.co.uk/object/geforce_family_uk.html. Accessed 2010-05-28.

[47] Advanced Micro Devices. ATI Radeon and ATI FirePro Graphics Cards from AMD. http://www.amd.com/us/products/Pages/graphics.aspx. Accessed 2010-05-28.

Appendix A

Table A.1 shows a comparison of modern graphics hardware [45][46][47].

	GeForce 8800 GTS	GeForce GTX 285	GeForce GTX 470	GeForce GTX 480	Radeon HD 4870	Radeon HD 5850	Radeon HD 5870
Manufacturer	NVIDIA	NVIDIA	NVIDIA	NVIDIA	AMD	AMD	AMD
Architecture	G80	GT200	Fermi	Fermi	R700	Evergreen	Evergreen
GPU	G80	GT200b	GF100	GF100	RV700	Cypress	Cypress
Production year	2006	2009	2010	2010	2008	2009	2009
SMs	14	30	14	15	10	18	20
Cores / SM (Total)	8 (112)	8 (240)	32 (448)	32 (480)	80 (800)	80 (1440)	80 (1600)
Max threads / SM	768	1024	1536	1536	1024	1536	1536
Process (nm)	90	55	40	40	55	40	40
Transistors (M)	681	1400	3200	3200	956	2154	2154
Main memory type	GDDR3	GDDR3	GDDR5	GDDR5	GDDR5	GDDR5	GDDR5
Main memory size (MB)	640	1024	1280	1536	1024	1024	1024
Memory bus width (bit)	320	512	320	384	256	256	256
Memory bandwidth (GB/s)	64	159	134	177	115	128	154
Local memory (B)	16384	16384	49152	49152	16384	32768	32768
Register file size / SM (B)	8192	16384	32768	32768	262144	262144	262144
Core clock (MHz)	500	648	607	700	750	725	850
Memory clock (MHz)	800	1242	1674	1848	1800	2000	2400
Max power (W)	147	183	215	250	150	158	188
Theoretical TFLOPS	~0.4	~0.7	~1.1	~1.35	~1.2	~2.1	~2.7

Table A.1. Comparison of 7 recent GPUs.