UNIVERSITETET I OSLO
Institutt for informatikk

Exploring computational capabilities of GPUs using H.264 prediction algorithms.

Masteroppgave

Magnus Funder Halldal

Abstract

In recent years, there has been a drive towards parallel architectures to further increase computational performance. The many-core architecture of modern Graphics Processing Units (GPU) can be used for general computations in addition to graphics processing, and provide good performance for data parallel computations.

In this thesis, we explore the processing powers of two generations of GPUs by implementing H.264 prediction algorithms. We have implemented motion vector search and motion vector prediction on the GPU, and discuss how they to the parallel architecture.

Acknowledgements

First and foremost, I want to thank my supervisors; Håkon Kvale Stensland, Pål Halvorsen and Carsten Griwodz, for their help, guidance and positive attitude during the work with this thesis.

I also want to thank the Media Performance Group at Simula for providing fun and interesting courses at the University of Oslo.

Finally, I want to thank the guys and girls at the lab for a fun work environment and interesting conversations.

Contents

List of Figures

Listings

.

Chapter 1

Introduction

1.1 Background

Since the invention of the microprocessor in the 1970s, there has been a tremendous increase in computational performance. However, in recent years the progress in conventional, one core, processing units has slowed down due to physical limits[14]. To further increase the performance of microprocessors, the manufacturers have started a horizontal scale, increasing the number of processing cores capable of concurrent executing instructions instead of the traditional approach of increase in clock speed and number of transistors of one processing core. As stated in the concurrency revolution[20] - programs does not longer necessarily get an automatic performance increase in sequential programs with the next generation of hardware, and programmers must learn parallel programming to continue to increase performance of applications.

The Central Processing Units (CPU) have become multicore architectures with the new generations of CPUs. The new generations of CPUs offer increased processing powers to parallel applications while maintaining the execution speed of sequential applications. In addition to the CPU, the

Graphics Processing Unit (GPU) has also become a highly parallel computing device. In contrast to the CPUs, the development of GPUs does not focus on sequential execution speed, and the GPUs have evolved towards massively parallel processors specialized on data parallel tasks. This evolution, and the design of GPUs have been driven by the demands for graphics rendering where the same instructions are executed on large independent datasets[8].

Over the last generations, the GPUs have turned into programmable devices with good parallel processing power, and as a result there was a desire to utilize the GPU for general purpose computations in addition to graphics processing. When NVIDIA introduced the CUDA framework, silicon was specific devoted to computations and from now GPUs were designed for general computations in addition to graphics. Application developers can potentially benefit from increased performance by offloading compute intensive data parallel computations from the CPU to the GPU. An incentive for targeting the GPU as a device for application speed up, is its highly availability among consumer computers at a relatively fair price. The processing powers are available in normal computers among consumers, and not only in super computer clusters at scientific labs or at large companies[8].

1.2 Problem Definition / Statement

To understand the capabilities and limitations of a GPU as a device for general purpose computations, we want to implement and evaluate compute intensive and data independent tasks and compare the result to a less compute intensive task with data dependencies. We will compare the result to a CPU implementation and measure the differences in performance. We also want to to understand how well suited the computations in the different stages of video encoding are for offloading to a GPU.

We will implement parts of the H.264 video encoder, and explore the possibilities for using the GPU for general purpose computations. H.264 is a

widely used standard for distributing and compressing video with high efficiency, which also results in a high complexity and a high compute intensive encoding pipeline[21]. We will implement the motion estimation and motion vector prediction parts of the encoding process, which is used to enhance the compression ratio by predicting information based on previous data. Motion estimation is an example of an algorithm which can be implemented with high computational requirements and no data dependencies. Motion vector prediction on the other hand, has lower computational cost. However, this process is highly data dependent, something which probably makes it less suitable for parallel execution.

We will also experiment with different adjustments to the implementations and evaluate how the differences affects performance. Different implementations of the same computations may have big differences in performance[19] The implementations will be tested on two different generations of NVIDIA GPUs and the results compared. Newer generations of GPUs are more adapted to general computations, and we want to examine how the improvements in hardware improve performance and ease the development.

1.3 Limitations

We concentrate on the motion vector search and the motion vector prediction process of the video encoding to explore the computational capabilities of the GPU.

1.4 Research Method

We implement and test prototypes of motion vector search and motion vector prediction. We make adjustments to the implementations to explore how that affects performance.We measure the performance against a CPU imple-

mentation and also on two generations of GPU hardware. The time against the CPU is to get an understanding of how well suited the task is for GPU offloading or if it should rather be executed on the CPU.

1.5 Main Contributions

We have tested a compute intensive motion vector search implementation on two generations of GPUs and discussed improvements made in the latest version. Then we present an approach to motion vector prediction where the predictions are processed in a diagonal pattern in the search frame. We present how CUDA threads can be managed for a data dependent problem, where data elements must be accessed in a specific order, and compare two different approaches. Finally, we discuss which computations are suited for GPU offloading and what are the limitations of the GPU, and how this have evolved over the last GPU generations.

1.6 Outline

In chapter 2, we give an introduction to GPUs and the CUDA framework. Chapter 3 gives an overview of the H.264 video coding standard. Chapter 4 summarizes previous work related to video encoding on GPUs. Chapter 5 gives a more detailed description of the motion estimation and motion vector prediction algorithms. Implementation is described in chapter 6, and in chapter 7 experiments and results is discussed. Finally, conclusion and further work is presented in chapter 8.

Chapter 2

GPU and CUDA

2.1 GPUs

A GPU i specialized for parallel computations which has been driven by a need for highly parallel computations which is needed for graphics rendering. Compared to a CPU, a GPU is designed with more transistors dedicated to data processing than to caching and flow control.

A GPU is well suited for problems where the same program is executed on many data elements. This is known as data parallelism, and is the reason why a smaller number of transistors need to be devoted to flow control and more transistors can be devoted to computations. This can be utilized to speed up computations in applications which process large amounts of data, and the same instructions are executed on all or most of the data. Examples of usage are in signal processing, physics simulation, computational finance and computational biology.

In addition to performance, presence in the market and availability for customers is also an important consideration when deciding to use the GPU as a platform for general purpose computations. Traditionally, parallel computing systems have not been available to the vast majority due to high

costs. This has changed with the mass production of GPUs. Now, almost all new personal computers comes with a GPU which can be utilized for computations[8].

2.1.1 Multicore and Manycore architectures

Traditionally, most computer programs were written as sequential programs. The programmer could rely on that advances in hardware would also increase the performance of the sequential application as the execution speed of a CPU increased for each new generation. The same software simply runs faster, and there was no need to make adaptations in the software for utilizing the increased performance in the new hardware. This slowed down in 2003 du to energy-consumption and head-dissipation issues[8]. The performance witch could be achieved by one single CPU was limited.

Instead, manufacturers switched to multi-core processing units, and the overall performance in increased by adding more processing cores. This can be referred to as horizontal scaling as opposed to vertical scaling where the performance of one unit is increased.

After the introduction of horizontal scaling of CPUs, were more processing cores is added to increase performance, the performance of a sequential written application does not automatically increase. This is because a sequential program only runs on one of the cores and does not fully utilize the processing power of a multicore CPU. To fully benefit from the increased processing powers, the sequential program needs to be rewritten into a parallel program which runs across multiple cores. This is of course only possible if the computations can be run in parallel, something which is not always the case. Although, more cores are added and the potential for parallelism increases, the multi-core architectures as CPUs still seek to maintain the execution speed of sequential applications.

Many-core processor is a terminology which can be used to distinguish a

processing unit architecture from the architecture used in CPUs with several cores. In multicore processors, the serial execution performance is maintained while the architecture is expanded with multiple cores for parallel execution of programs. The intel i7, with four processor cores, is an example of a multicore processor. NVIDIA uses the terminology many-core architecture when referring to their GPU architectures.

Many-core processors are designed to maximize the execution throughput of parallel applications. To utilize the processing power of many-core processors a parallel implementation of the application is required, and problems which are not possible to parallelize, and thus require a serial implementation may achieve a better performance on the multicore processor than on a many-core processor as the multi-core processor still seeks to maintain sequential program execution speed.

The architecture of a modern GPU consists of several Streaming Multiprocessors (SM). Each SM in turn has a number of Stream Processors (SP). Each SP in an SM share control logic and instruction cache. In addition to the processing units, a GPU also has a DRAM, referred to as global memory. For graphics applications, the DRAM contains video images and texture information and is also referred to as the frame buffer. When the GPU is used for computing, the DRAM, or global memory, it contains the input and output data of a general purpose computing application[8].

2.1.2 The GT200 architecture

NVIDIA introduced the GT200 architecture in the GeForce GTX 280, Quadro FX 5800 and the Tesla T10 GPUs in June 2008. Improvements compared to previously GPU architectures was the increased number of stream processor cores to 240. These cores are also referred to as CUDA cores. The register file of each processor was doubled in size which made the GPU capable of executing more threads on-chip. Coalesced access to global memory was

added to improve memory access efficiency, and support for double precision floating point operations was also added.

The GT200 architecture focuses on data parallel computations, and achieving a high throughput for these computations. A contrast is general purpose computations where the focus is on achieving the highest possible sequential performance for one single thread. The architecture of the GT200 is composed of 10 TPCs (Texture Processing Cluster), consisting of 3 SMs (Streaming Multiprocessor). Each group of 3 SMs making a TPC shares a memory pipeline. The SM consists of units for instruction fetch, decode, issue logic and 8 execution units which NVIDIA refers to as CUDA-cores or Streaming Processors (SP). Each SP has its own register file and instruction pointer, however as it does not have its own units for instruction fetch and schedule, it can not operate entirely independently.

Threads which are executed on the GPU are organized into thread blocks which are executed on the same SM. The GPU has a global block scheduler or work distribution unit which are responsible for scheduling the thread blocks among the SMs. The thread blocks are issued to SMs with available resources by the global scheduler depending on factors such as threads per block, amount of shared memory per block and the number of registers per thread. For a thread-block to be scheduled to an SM, the required amount of resources need to be available, and the global scheduler emphasizes uniformly distribution of the thread-blocks across the SMs and maximization of the parallel execution on the GPU.

On the SM, further scheduling of the threads and thread blocks are managed by the SMs thread scheduler. A total of 1024 threads and 8 thread blocks can be executed on an SM and the thread scheduler is responsible for scheduling the threads, which were previously scheduled to the SM by the global scheduler, among the available resources on the GPU. The threads are scheduled in groups of 32 threads, referred to as a warp, which are threads executed simultaneously on the hardware.

The GT200 architecture has 16KB of shared memory per SM which are accessible by all threads in a thread block. This shared memory can thus be used for inter-thread communication between threads in the same thread block. Another reason to use shared memory is to increase performance, as the access latency of shared memory is much lower than the access latency for global memory. The latency of the shard memory is the same as the latency for accessing registers. A high increase in performance can potentially be achieved by first reading data into shared memory before starting the computations. This can be data which need to be read several times by all the threads in the thread block.

Shared memory is also essential for atomic instructions. Atomic instructions are used when we need to assure that a thread writes its result before an other thread can access the same memory and avoid concurrency issues between threads. The GT200 architecture introduces atomic instructions using shared memory and the performance of these instructions are increased compared to previous architectures where atomic instructions had to use the higher latency global memory.

Another type of memory which also provides low memory access latency is The register file.The register file is used for data private for each thread and is 64 kb per SM. A single thread may use up to 128 register entries of 32 bit size or up to 64 register entries for double precision data types as they require two adjacent 32 bit register entries. When scheduling blocks of threads, the scheduler needs to consider the amount of free space in the register file. There need to be enough free space for the private data of all the threads in a thread block. If threads uses a lower amount of registers, this may allow more blocks of threads to be scheduled on an SM at the same time. [12][2]

2.1.3 The GF100 architecture

The NVIDIA GF100, or FERMI architecture was available from march 2010. This GPU was designed both for graphics rendering and for computational applications. The GPU has a true cache hierarchy with both an L2 and L1 cache, and also has the possibility to increase the amount of shared memory. The GPU can be configured to either use 48 kb for L1 cache and 16 kb for shared memory or 48 kb for shared memory and 16 kb for L1 cache. With 32 CUDA cores per SM and 16 SMs, the FERMI architecture can feature a total of 512 CUDA cores on the GPU, and each of these cores, containing one integer unit and one floating point unit, are able to execute an integer instruction of a floating point instruction per clock cycle for a CUDA thread.

CUDA cores resides on an SM - Streaming Multiprocessor. Each SM has its own instruction cache, warp scheduler, dispatch unit, register file and 64 kb of memory which can used for caching or shared memory. The SM has 32 CUDA cores, 16 load/store units, and 4 SFUs - Special Function Unit - for executing functions such as sin, cosine and square root[3].

GPUs based on the fermi architecture are dual natured processing units. GF100 was designed for increased computation performance creating not only a new processing unit for graphics but also a processing unit designed for computations. Some of the new features added which increases the usability of the GPU as a computational device are a new caching hierarchy and ECC support. Caching is important to graphics, but much more important to computing. ECC is important for protecting computational data.

The GF100 architecture increases the number of Streaming Multiprocessors on the GPU to a total of 16. Each SM now has its own memory pipeline and it is no longer shared among three SMs as it was in the previous GT200 architecture. The number of functional units, or CUDA-cores, per SM is increased to 32. These consist of an integer unit (ALU), and a floating point unit. The issue port are shared among the data path for these units, and thus

it it is not possible to issue instructions to the integer units and the floating point units simultaneously. A new feature of the GF100 architecture is a cache hierarchy with an L2 cache shared among all the SMs and an L1 cache per SM. The size of the L2 cache is 758 kb and the L1 cache is configurable of either 16 kb of L1 cache and 48 kb of shared memory per thread block or 48 kb L1 cache and 16 kb of shared memory[3].

The GF100 has a global scheduler which schedules work among each SM. This is similar as the GT200 architecture. However, the scheduler was only able to schedule thread blocks for one kernel so there could be only one kernel executing on the GPU at a time. The GF100 introduces the ability to execute multiple kernels concurrently as the global scheduler are able to maintain state for one kernel per SM; a total of up to 16 multiple kernels[3].

The execution units on the SM are shared between two execution pipelines and the SM has one thread scheduler for each pipeline for scheduling threads among the SM resources. The two schedulers are together able to dispatch two warps every clock cycle, one for each pipeline. The warp size in GF100 is 32, same as the warp size in GT200. To support the demand for registers for the increased amount of active threads the register file is increased to 128 kb with 32K 32 bit entries. Compared to the previous architecture, execution throughput is doubled and the latency for each warp is halved to two cycles [3].

Further improvements to the execution resources which increases performance are FMA - Fused Multiply ADD - instructions and increasing the number of Special Function Units (SFU) to 4 per SM, reducing the latency of a warp executing SFU instructions. SFU instructions can also execute simultaneously with ALU or FPU instructions enabling further utilization for the execution resources on the SM. The SFU are special function units for executing instructions e.g sin, cos and square root. A warp executes in eight clocks as the SFUs execute one instruction per thread, per clock[3].

According to NVIDIA, the performance of atomic operations are increased

by 5-20X compared to GT200. The L2 cache together with additional atomic execution units are used to improve the performance of atomic operations. The L2 cache also contributes to increased performance of non-atomic operations as it can reduce the amount of memory transactions from global memory[13].

2.2 CUDA

CUDA was introduced by NVIDIA in November 2006 and is a parallel programming framework for general purpose computations. For making parallel applications, CUDA allows developers to program in C/C++ with some extensions, and are intended to provide a low learning curve for programmers already familiar with these programming languages. The extensions to the standard programming languages C and C++ include allocating memory on the GPU, copying data to and from the GPU, making an abstraction for processing data in parallel on the GPU and synchronization and communicating between threads executing in parallel[18].

2.2.1 Programming model

The parallel code, which are executed on the GPU, are written in a function which is referred to as a kernel. When the kernel are launched, many threads share the same code and they start executing in parallel. All the threads which are launched are referred to as a grid, and the grid is divided into blocks of threads. A block of threads are a group of threads which are executed about the same time on the GPU. They have a shared memory for communication and fast access to data between threads, and it is possible to synchronize execution between the threads. The number of threads per block and the number of blocks per grid is important decisions when programming using the CUDA framework, as it affects both performance and how indexes

into the data can be computed for each thread.

CUDA provides built in variables for accessing the thread index and the block index for one thread. These variables are used in the CUDA kernel code to make each thread operate on different parts of the data. The thread blocks are scheduled on the GPU by the hardware and it is not possible to know when the thread blocks are executed in relation to each other. Therefore, it is important to organize the threads such that each thread block can execute independent of each other, and thus the results of the computations does not depend on execution order, synchronization or communication between thread blocks[18].

2.2.2 Scalability

If the parallelism of multicore CPUs and manycore GPUs continue to scale with Moore's law, this gives an increase in the number of cores on the architectures over time. CUDA addresses this situation as it scale the parallelism of an application as the number of cores on the hardware increases.

CUDA provides a hierarchy of thread groups as an abstraction to the programmer, and the programmer needs to partition the problem into subproblems which can be solved by a block of threads independently from other thread-blocks.

This enables automatic scalability when the CUDA program is executed on different GPUs. When the program is executed on a more cores, the run time environment schedules more CUDA blocks to be executed concurrently. Each CUDA block can be executed on any core and in any order. A CUDA program can thus scale a cross a wide range of different GPUs.

Figure 2.1 from [12] shows an example of how a cuda program can be scheduled on a GPU with two cores and a GPU with four cores. This shows the importance of dividing a cuda program into smaller blocks for utilizing the

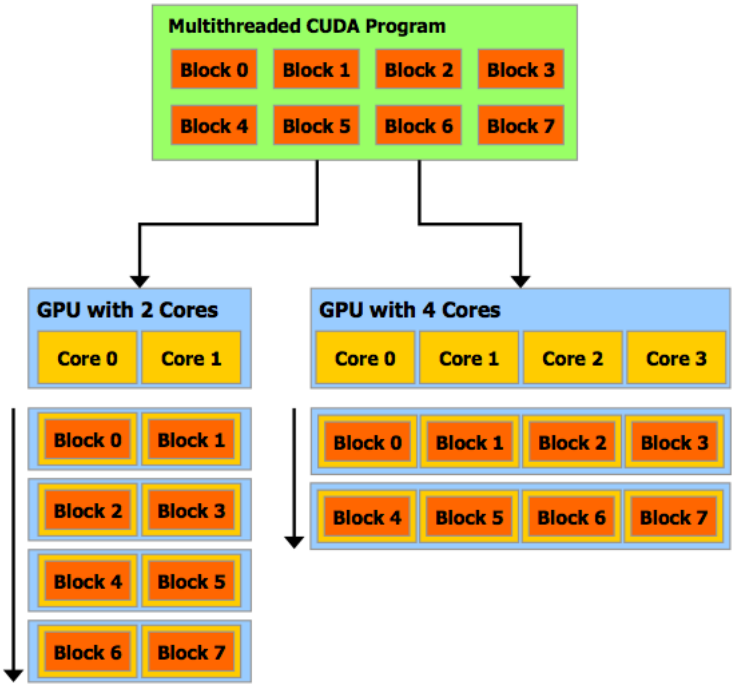

Figure 2.1: Kernel scalability

scalability capabilities of cuda. If the program consisted of only two blocks, it would not be possible to schedule the blocks among more cores on the four-core GPU, and the increased number of cores on this GPU would not be utilized.

2.2.3 Kernels

Kernels is an extension to C which allows the programmer to define a function which are executed in parallel by all the CUDA threads. The kernel is launched from the CPU code with parameters defining the numbers of thread-blocks and number of threads per block. The dimensions of the thread blocks and the grid is also defined here, whether it is one, two or three dimensions.

The code in listing 1.1 are executed in parallel by the CUDA threads which are launched from the host. Every index i in array a is added with index i in

Listing 2.1: Kernel example.

```
__global__ add(int *a, int *b, int *c)
{
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  int c[i] = a[i] + b[i];
}
```

array b and stored in array c. The variable i is computed form the treadIdx, blockIdx and blockDim variables. The value of these variables are unique for each thread and can be used to make the threads running in parallel make computations on different parts of a data structure. If e.g the arrays in listing 1.1 are of size 1000, we can launch 1000 parallel threads form the host to make the computation, and are able to operate on different elements in the array because the index is computed from the built in variables.

Making the hardware able to coalesce global memory access, certain requirements has to be met. These requirements are different depending on the compute capability of the GPU, and the requirements are lesser strict on newer generations of GPUs which makes it more easy for the programmer to achieve global memory coalescing.

Before the kernel in listing 1.1 can be executed on the GPU, the memory holding the data in the a, b and c arrays need to be allocated and the data needs to be transferred from the host to the device. The code listed in listing 2.2 shows how memory for the arrays, a b and c in the CUDA kernel can be allocated using the built in function cudaMalloc. The function takes two parameters, the address of a pointer, and the size of the memory which are going to be allocated in bytes.

When launching the cuda kernel from the host, first the data which the kernel needs for the computations have to copied from the host to the device. The copying of the data can be done using the built in function cudaMemcpy.

Listing 2.2: Memory allocation on the device.

```
int main()
{
int *d_a;
int *d_b;
int *d_c;

cudaMalloc(&d_a, 100*sizeof(int));
cudaMalloc(&d_b, 100*sizeof(int));
cudaMalloc(&d_c, 100*sizeof(int));
.
.
.
```

The function takes four parameters; the to address, the from address, the size of the data, and a constant defining if the data are going to be copied from the device to the host or from the host to the device. After the copy is finished, the kernel is launched from the host code with the special syntax «<a, b»> defining the number of blocks and the number of threads per block. In the example we launch 4 blocks with 25 threads per block, a total of 100 threads which are needed for adding the elements in the 100 element arrays. The copying of data and kernel launch is showed in listing 2.3

2.2.4 Thread hierarchy

The threadIdx variable is used inside a kernel to get the id of the thread executing the kernel. The threadIdx variable can have up to three dimensions and the values are accessible through either threadIdx.x, threadIdx.y or threadIdx.z. The dimensions of the threadIdx can be used to reflect the dimension of the data structure, and provides a natural way to access the data elements. If e.g the data structure is an array, we use one dimensional

Listing 2.3: Memory copy and kernel launch

```
int main()
{
.
.
.
 cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
 cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice);
 add<<<4, 25>>>(d_a, d_b, d_c);
 cudaMemcpy(h_c, d_c, size, cudaMemcpyDeviceToHost);

return 0
}
```

threadIdx, if it is a matrix, we use a two dimensional threadIdx and if it is a cube, we use a three dimensional threadIdx.

The threads are grouped together in blocks of threads, and a thread-block can be either one or two dimensional. The index of a block is retrieved by using the built in blockIdx variable similar as the threadIdx variable, and are accessible thorough blockIdx.x and blockIdx.y depending of whether a one or two dimensional block index is being used.

All the treads in a block are scheduled to the same multiprocessor, and all the resources needed to execute the thread block, such as registers and shared memory need to be available for the thread block to be scheduled. Because hardware resources are a limiting factor for scheduling of thread blocks, there is an upper limit of 1024 threads per thread block as they need to be scheduled to the same multiprocessor at the same time. The maximum number of thread blocks i 650xx, and is much higher than the max number of threads per block because the threads blocks does not need to be scheduled at the same time and can wait for resources to become available among the

SMs.

2.2.5 Memory model

The programmer needs to consider different memory spaces when programming using the CUDA framework. Each thread have access to register variables which are accessible only by each specific thread and provide low latency access for data private for each thread.

All threads in a block share a memory space named shared memory. The access latency of the shared memory is the same as registers, and thus provide fast access to data which are shared among threads and need to be accessed several times by many threads. The shared memory also makes it possible for threads within a block to communicate and transfer data to each other.

Global Memory

Data which are transferred from the host to the device are first copied to the Global memory. This is the memory space on the GPU with the biggest size, and the highest latency. It is accessible by all threads and thread blocks, and therefore useful for storing large amounts of data, and data which need to be accessed by threads in different thread blocks.

Shared Memory

Shared memory is on-chip fast memory with access latency which can be 100 times lower than the access latency of global memroy.[11] The shared memory is used for speeding up computations with fast memory access and for inter-thread communication between threads within a block.

Local Memory

Local memory is allocated in global memory and is private for each thread. The local memory is used for arrays and data structures which are to big for the registers, or other variables that uses more registerers than available on the GPU. Local memory has the same access latency and memory bandwidth as global memory, and requirements for memory coalescing also applies for local memory. As long as all threads in a warp access the same array index or struct member the local memory access are fully coalesced [12].

Texture Memory

Constant Memory

Constant memory are off chip memory which are cached by the constant cache. The caching decreases access latency for data-read, however constant memory is a read-only memory and can not be used by the threads for writing. Data in constant memory is also available to all thread blocks, thus performance can be increased by using constant memory for data which are read by many or all thread blocks.

Registers

Registers provide fast on chip memory with low latency for data which is private for each thread.

2.2.6 SIMT architecture

Threads are scheduled and executed in groups of 32 threads called a warp. All threads in a warp start at the same address in the CUDA program and are executed concurrently on the hardware. Each thread in a warp has separate

instruction counter and maintain their own register state. Thus, they are able to branch and execute independently of the other threads in the warp. However, only one instruction is executed for the threads in a warp at a time, so if branches in the code causes execution paths to diverge, each path need to be serially executed and parallel execution only occurs within each execution path.

2.2.7 Performance considerations

The multiprocessor maintains the state of the execution contexts such as program counters and registers for the hole lifetime of a warp. Context switches between different warps of threads can thus be executed very fast by the hardware with no overhead.

Occupancy

Occupancy is defined as the number of active warps per multiprocessor to the maximum number of possible active warps per multiprocessor. This is a way to view the percentage of the total possible processing ability of the hardware versus the ability being used. However, it is important to be aware of that a higher degree of occupancy does not always result in higher performance.[11]

Grid size and block size

The size of blocks per grid and the size of threads per block are important factors when programming in CUDA. The size of a block can be one, two ore three dimensional and the size of a grid can be one or two dimensional. The multidimensional aspects of the blocks and grids are only used for allowing more easy mapping of multidimensional data structures to CUDA and do not affect performance. To hide latency the hardware must be able to switch execution to an other block of threads if one block is waiting for memory

transfers or thread synchronization. Therefor, there should always be several blocks per SM so the hardware always can choose a different block to execute. The size of a block should also not be to large, as several block can be executed on the SM at the same time[11].

Data transfer between host and device

The bandwidth between host and device is much lower than the bandwidth between the GPU and the device memory. The NVIDIA GTX280 has a peak bandwidth of 141GBps. Compared to the bandwidth of the PCIe x16 Gen2, which is 8GBps, the bandwidth between the GPU and its global memory is over 17 times higher. Hence, data transfer between the host and the device should be minimized. Sometimes it may also be preferable to run kernels on the GPU which do not show better performance than running the same computations on the CPU, if it reduces the amount of data transfer between the host and the device[11].

Global memory coalescing

Coalescing global memory access is perhaps the single most important performance consideration when programming for the CUDA architecture. [11] When memory transfer is coalesced, loads and stores for several threads are combined into one transaction. For devices of compute capability 1.x the memory transfer of a half warp (16 threads) can be coalesced, and for devices of compute capability 2.x the memory coalescing is done for a full warp of threads[11].

Shared memory bank conflicts

The shared memory is divided into memory banks. Each bank is 32 bits. If more than one thread of the same warp accesses the same memory bank, this

results in a bank conflict. The consequence of a bank conflict is the memory access being serialized and thus causing a decrease in performance. If three threads access the same bank at the same time, causing bank conflicts and serial memory access, the memory access could be done in parallel if each threads were accessing different memory banks[11].

Divergent warps

Threads are executed in groups of 32 threads. This group of threads are referred to as a warp and are executed at the same time on the hardware. To maximize performance each warp of threads should follow the same execution path. This is because the threads can only run i parallel on the hardware as long as the execution path of a warp does not diverge. Control flow instructions such as if, switch, do, for, while, can cause threads within a warp to follow different execution paths. In theses situations, the hardware can not execute the different execution paths in parallel and they need to be serialized. Therefore, branching in the code resulting in divergent warps should be avoided, however, this may not always be possible[11].

Chapter 3

Overview of the H.264 video encoding standard

H.264 is a standard for video compression which is widely used for distributing and compressing video data. Examples of usage are in terrestrial and satellite broadcast services and as one of the mandatory formats for Bly-ray. H.264 achieves a high compression efficiency which implies a high complexity of the encoding process.[21]

Features of the H.264 video encoding standard which enhances coding efficiency are small and variable block size motion compensation, quarter pixel motion compensation, motion compensation using multiple reference pictures, display order of frames are decoupled from referencing order, and small block size transform.

The standard defines the decoding of a bitstream, with syntax of the bitstream and methods for the decoding process. How to implement the encoder and create the encoded bitstream are thus left to the developers of the video-encoder, as long as the encoder produces a valid bitstream which conforms with the standard.

3.1 Key principles of H.264

3.1.1 Chroma subsampling

H.264 uses the YCbCr color space with the possibility to use chroma subsampling. One component is used for representing luma, the light intensity, and the two other components are used to represent blue color and red color, referred to as the chroma components. With subsampling, a reduced number for chroma values are used for representing the frame. This is done by using the same chroma component for several luma values.

3.1.2 Macroblocks

H.264 is a block-based video codec, where the frames are divided into blocks of size 16 x 16 pixels. The macroblocks can be further divided into sub-blocks of 4 x 4, 8 x 4 and 4 x 8. Motion compensation can be done for the hole macroblock, or it can be done for the sub-blocks, resulting in a finer level of granularity. Motion compensation is used in H.264 and other video coding standard to reduce the entropy in the data. This is achieved by using the residuals between the original block and the block used for prediction when doing the entropy encoding.

3.1.3 Integer transform

H.264 uses a DCT-based transform with some differences which makes it possible to do a lossless inverse transform, and all the operations in the transform is done using integer arithmetic. The normal DCT transform does not have the ability to fully restore the exact original values when doing the inverse transform, and H.264 are thus using this modified version.

3.1.4 Prediction

Much of the increase in compression efficiency of H.264 is due to the prediction methods where a prediction is created for every macroblock. The encoder attempts to predict the pixel value of a macroblock based on prediction from other previously encoded blocks. If the residuals, the difference in pixel values between a block an the one used for prediction, contains very little data, this in turn gives a good compression efficiency. This is because a block consisting of mostly zero-values can be represented by very few bits after entropy coding is applied.

I macroblocks, P macroblocks and B macroblocks are different types of macroblocks. An I macroblock is predicted using only prediction from neighbor blocks in the same frame as the macroblock to predict. No reference to other frames is used, and there are only dependencies within the current frame. Intra prediction is based on the relatively high correlation between pixels in the block and its neighbors. Thus, previously encoded blocks adjacent to the bock to be predicted are used for prediction.

In intra prediction the partition size of a macroblock can be 16x16, 8x8 or 4x4. Thus, there are not so many different partition sizes to chose form as there is for inter prediction, which also has 8x16, 16x8 and sub partitions of 8x8 blocks. The intra prediction can be performed using different modes. A 16x16 macroblock has four possible prediction modes. If the macroblock is partitioned into 8x8 or 4x4 blocks, these blocks have nine possible prediction modes.[15]

The prediction of a P macroblock is computed based on macroblocks in previously encoded frames. The frames used as a reference frame for prediction can be both before or after the current frame in display order. If a frame which is ahed in display order is used as a reference frame, this frame need to be encoded before the frame which are currently predicted and there is a decoupling of display order and encoding order. Macroblocks can be split

into macroblock partitions, and each macroblock partition can be predicted from different reference frames. However, when 8x8 macroblock partitions are further split into sub macroblock partitions, the prediction of the sub macroblocks in the same partition always uses the same reference frame.

The offset to a block used for prediction in a reference frame is stored in a motion vector, which has inter pixel, half pixel or quarter pixel precision. To use half pixel or quarter pixel precision in the motion vector, the pixel values in the reference frame need to be interpolated before the motion vector search for the best matching prediction block is performed.

The process of inter prediction can be summarized in the following steps. First the pixels in the possible reference frames are interpolated, if the encoder choose to use half pixel or quarter pixel motion vector precision. Then a reference frame is chosen among the frames available. Frames which can be used for prediction are stored in a Decoded Picture Buffer and can vary from one to several frames. The block size to predict is chosen among the different possible macroblock partition sizes or sub-macroblock partitions. Then the motion vectors for each block is determined, where the encoder chooses the motion vector using a motion vector search algorithm. The motion vectors are predicted from other motion vectors in the same frame to reduce the amount of data needed to represent the vectors. The prediction information and residuals in the frame are then coded and reconstructed for use as a reference frame for prediction in other frames. Finally a deblocking filter is applied to the reconstructed frame to remove blocking artifacts.[17][15]

3.1.5 Quantization

After the transform, the residuals are quantized, a process which also contributes to the compression of the bitstream. The quantization is the lossy step of the encoding process as it is not possible to regain the original values after the quantization step. The quantization, however, is an important

process for getting a good compression ratio and the choice of quantization parameter, determining the compression ratio of the video stream, is a trade-off between image quality and compression efficiency[16].

3.1.6 Reordering

After the quantization, the coefficients from the DCT transform are being reorder to organize the data in such a way that it improves the compression ratio. The quantize step in the encoding process produces many zero values and a few non zero values. The purpose of the reordering, doing a zig-zag scan of the data, is to organize the quantized coefficients with the non zero values in one group followed by the zero values. This reordering is done before the entropy coding, and the impact of the entropy coding on the compression ratio are thus enhanced[16].

3.1.7 Entropy coding

The entropy encoding makes the actual compressing of the data after the prediction process is done to reduce the entropy. The H.264 standard defines two types of entropy encoding. Context Adaptive Variable Length Coding (CAVLC) and Context Adaptive Binary Arithmetic Coding (CABAC).

3.1.8 Prediction of motion vectors

The motion vectors are transmitted using predictive coding to reduce the amount of data necessary to represent the motion vectors. Motion vector prediction increases coding efficiency by predicting motion vectors from previously encoded motion vectors. MVD (Motion Vector Difference) is the difference between the motion vector to be predicted and the motion vector used for predicted, and only this difference, MVD, is transmitted and used

in the decoder to reconstruct the original motion vector using an inverse prediction process[16].

3.1.9 Deblocking filter

The purpose of a deblocking filter is to smooth the block edges and reduce the blocking artifacts, which can be seen in frames of block-based video coding, caused by the partitioning of frames into blocks. The filter is applied in both the encoder and in the decoder after the inverse transform of the blocks in the frame. In the decoder this is before the reconstruction and displaying of the frame. In the encoder it is before the reconstruction and storing of the frame for predictions. It is important that the encoder computes the predictions based on reconstructed frames, as it has to be based on the same data used for motion compensation in the decoder[16].

3.1.10 Frames and slices

A frame can be partitioned into slices. The standard defines slices of specific types which determines how the slices is encoded. The frame does not have to be divided into slices and will then be a frame consisting of one slice.

The different slice types are I-slice, P-silce and B-slice. These can also be referred to as I-frame, P-frame an B-frame if division into slices are not used and they thus are frames consisting of only one slice, or only one slice-type is used within the frame

I (Intra predicted slice) - All macroblocks in the slice are intra predicted. Using intra prediction, the pixels in the macroblock are predicted using previously encoded data from the same frame where the macroblock resides. An intra predicted frame are only dependent on the frame itself and non of the the macroblocks in the frame are dependent on other frames for predicting its pixel-values.

P - Predicted slice: The slice contains one or more macroblocks which are predicted from previously encoded frames. A P-slice does not need to contain only macroblocks predicted from other frames, and can also contain intra-predicted macroblocks.

B - slice: Can be predicted by one or more slices and slices used for prediction can be in the past or in the future in display order[16].

3.2 The H.264 syntax

3.2.1 NAL Units

NAL Header

The header describes the type of the NAL Unit, and how important it is. E.g slices used to predict further slices have high priority because further decoding depends on these slices, and slices not referred by other slices have less importance because they are not needed to decode other slices.[17]

Sequence Parameter Set

The SPS contain parameters common an entire video sequence. Here is the information about the encoding profile, the video frame size and other constraints for decoding the video stream.

Picture Parameter Set

The PPS contain parameters that may be common to an entire video sequence or just a subset of the video sequence. The information in a PPS can be type of entropy coding used, the number of reference frames which

are currently active and parameters used for initializing the decoding of the stream.[17]

All slices slices uses the same parameter set until a new parameter set is activated. Thus, there is only a need to transmit a parameter set if it changes during the video sequence. If the same parameter set is used in the entire video sequence, the parameter set need only be transmitted once. The parameter sets can be transmitted before they are being used They stay inactive until they are referenced and are then set to active. A PPS is activated by being referred to in a slice header. An SPS is activated when a PPS which refers to it is activated.[17]

Video Coding Layer

The coded slices are the Video Coding Layer (VCL) NAL Units.

3.2.2 Slice layer

Each video frame consists of one or more slices. A slices consists of a slice header and a number of macroblocks. A video stream can have one slice per frame of several slices per frame. When multiple slices are used, the slices can contain a constant number of macroblocks or a varying number of macroblocks per frame. A reason to use a varying number of macroblocks per slice may be if the slices need to be of constant size for mapping to a network packet.

The slice header contains information common to all macroblocks in the slice. Slice type, frame number, reference frame settings and quantization parameter. After the slice header comes the slice data, which consists of several macroblocks. [17]

3.2.3 Macroblock layer

Macroblock coding type, and information about prediction and coding of the macroblock e.g I, P or B. An I macroblock is coded without reference to other frames and B macroblocks are coded using prediction from other frames. An mb_pred value defines the type of inter prediction or intra prediction used for this macroblock. How the prediction is performed is defined by several macroblock prediction syntax elements.

For an I macroblock the prediction mode for each block are signaled. This is only signaled explicitly for each block if it is an 4x4 block or an 8x8 block. If it is an 16x16 block, the prediction mode is already signaled in the mb_type and need not be signaled again.

P and B macroblocks need prediction information about reference frames and motion vectors. The reference frame for a macroblock is transmitted as an index to a frame in a list of reference frames. Only needed information is transmitted. If e.g only one reference frame is used in the encoding of a video sequence, the index to a reference frame need not be transmitted. The decoder knows then which frame to use without this stated explicitly as there is only one frame to choose as reference frame. The motion vectors are transmitted as motion vector differences, which are added to previously predicted motion vectors to restore the motion vectors for the current block.

If direct mode is used for a block, no information about reference frame of motion vectors are transmitted. This information is derived from previously decoded macroblocks in the decoder.

If the macroblock has an 8x8 partition size, the macroblock partition can be further divided into sub partitions of size 4x8, 8x4 or 4x4. For each 8x8 partition a sub_mb_type is sent which indicates either direct mode or size of the sub macroblock partitions and the reference frame. All blocks in a sub macroblock partition uses the same reference frame for prediction. Motion vector differences, however, are transmitted for each block.

All other macro blocks of partition size 16x16, 8x16 or 16x8 can not be further divided into sub macroblock partitions. For each of these partitions, reference frame index and motion vector differences are transmitted. [17]

Chapter 4

Previous work

4.1 Before CUDA

Research for utilizing the GPU for general purpose computations was made before NVIDIA introduced the CUDA framework. According to[10], which discuss implementation of motion estimation on a GPU, there had been no previous implementations of motion estimation on a GPU before the publication of this paper. In 2006, when this paper was published no framework had been introduced for programming the GPU as a device for general purpose computations. Therefore, the programming had to be done using OpenGL and the Cg computer language and runtime libraries from NVIDIA. The achievements in this research was compared to a Pentium IV 3 GHz CPU with 1 GB RAM. Compared to the CPU implementation, the results were two times speedup on the GPU for integer-pel and fourteen times speedup for half-pel. The high difference in speedup is achieved using the GPUs hardware support for interpolation.

4.2 Research using CUDA

In 2008, after NVIDIA had introduced the CUDA framework, implementation of motion estimation for H.264 using CUDA was presented in[4]. The paper discuss implementation motion estimation algorithm using the CUDA framework. The implementation proposed is a full search algorithm for image block size of 4x4 pixels with a motion estimation search range of 32x32 pixels.

The SAD calculation is done by dividing the SAD computations for an image block among 4 CUDA blocks with 256 threads in each block for a total of 1024 SADs. The 4x4 image block are put into shared memory so that each CUDA thread can access the image block data without reading from global memory. However, the data in the reference frame needs to be fetched from global memory for every thread.

To find the least sad a thread reduction process is applied. 128 threads are used to find the least SAD value among 256 SADs. For each iteration, one thread compares to SAD values and stores the least SAD. Then the number of active threads are halved and the process repeated until on least SAD value remains. To avoid shared memory bank conflicts and highly divergent warps, the implementation uses a sequential addressing strategy which avoids divergent branches. The thread reduction process are done in shared memory by first reading all the 256 SAD values from global memory before the reduction process begins. When the reduction is finished the least SAD value is written back to global memory.

Motion estimation on GPUs using the CUDA framework are explored in [7],and different approaches on how to implement motion estimation in CUDA are being discussed.

4.2.1 Using one thread to calculate one sad value between two pixels

Problems with this approach are block size has to be smaller than 22x22 because there is a limit of 512 threads per block. Further, an additional kernel for accumulating the SAD calculations is needed. Finally, and perhaps the biggest issue is because each thread only adds two values, an extensive amount of thread blocks is needed.

4.2.2 One thread per SAD calculation for one block

Every thread calculates the SAD value for one block within the search range, and one thread block computes all the SAD value for one image block. The advantage is that all SADs for one image block is computed within the same thread block. There is no longer a problem with big number of image blocks as it is with the previous approach. An issue with this approach is that the maximum threads per CUDA block is 512 which limits the maximum search range in the motion estimation because one thread is needed per SAD candidate.

4.2.3 Several CUDA blocks per search frame

The paper also discuss using multiple CUDA blocks to calculate SAD values for a search frame, where the search frame are divided among four thread blocks. This it the same approach used in [4] where four blocks with 256 threads each are used to compute SAD values for a block which have a search frame of 32 x 32 SAD candidates. This approach increases the maximum search range compared to the approach using only one CUDA block per image block.

4.2.4 A mixed approach

After experimenting, the paper concludes a mix of the above approaches is the best solution as it removes some of the limitations mentioned.

4.2.5 Other publications related to video encoding on GPUs

Vide coding on GPUs are discussed in [6] and [9] [5]

4.3 NUDT

The National University of Defense Technology in China has made an implementation of an H.264 encoder on GPUs using the CUDA framework.

4.4 NVIDIA H.264 encoder CUDA library

NVIDA has provided CUDA libraries for implementing H.264 on GPUs.

Chapter 5

Design

5.1 Motion Estimation

Motion estimation is the process of finding an offset in a reference frame which gives the best match for pixels in a block in the current frame. Block based motion estimation divides the current frame into blocks and for each block searches the reference frame for a block of pixels which has pixel-values differing the least from the block in the current frame. Motion estimation is implemented in the encoder and how this is implemented is not specified by the H.264 standard.

A full search algorithm matches the block against all other possible blocks in the search window. This is an operation which requires many computations and many reads and writes to memory.

A full search algorithm requires too many computations to be suited for many practical applications as it takes to long time to finish the motion vector search. In applications with computation limited constraints, a fast search algorithm is preferable. These algorithms computes the best mach for the block in only a subset of the search frame. The pros of these algorithms are increased speed as it requires less computations. The cons are it may

not find the best mach for the block within the search frame as it does not compute all possible matches. The full search always finds the minimum difference between a block in the current frame and a block in the search frame while a fast search algorithm may not always find this minimum value.

5.2 Motion vector prediction

When video frames are encoded, the data for motion vectors can be a significant amount of the total data needed to encode the frames. This is specially true when small block sizes are used in the encoding. A frame block size of 4x4 pixels requires up to 129 600 motion vectors for per frames for resolutions of 1080 x 1920 pixels. If two bytes are used per motion vector, this gives a total of 259200 bytes of motion vector data per image frame prior to entropy coding. To decrease the size of the motion vector data to be transmitted, H.264 uses predictive coding of motion vectors where the difference between the current vector and the predicted vector is transmitted. To predict a motion vector, the median of the motion vector to the left, en the top and on the top right are first computed. The median vector, of these three vectors are then used to predict the current motion vector.

5.3 Sum of absolute differences

The Sum of Absolute Differences (SAD) can be used as a measurement to determine which area in the reference frame is the best match for predicting an area of the current frame. It is decided by the implementors of the encoder which algorithm to be used for determining the best prediction block in a frame.

5.4 Thread reduction

A reduction algorithm can be used to parallelize the extraction of one data element from multiple data. After all the SAD values for the search frame is computed, thread reduction can be applied to parallelize the process of finding the least SAD value.

Chapter 6

Implementation

6.1 CUDA implementation using only global memory for pixel data

We implement a CUDA based sad function without using shared memory. On the GT280 architecture, all the data reside in global memory and on the GF100 architecture, the data can be cached when doing computations. Our first concern is dividing the search frame among the CUDA blocks and threads. We need to be able to compare the SAD values within the search frame to each other. If there are more SAD values to be computed within a search frame than there is threads available within a block we can either split the computations in one search frame among several blocks or make the SAD computation in several phases within one block. An other reason for dividing the work among several blocks or several phases is performance issues. Maximizing or near maximizing the number of threads within a block could give a lesser performance than using a lesser amount of threads and instead using several phases or blocks.

The GT200 architecture has a maximum of 512 CUDA threads per block and the GF100 architecture has a maximum of 1024 CUDA threads per block.

We want to test the same implementation on both architectures and thus we need to make an implementation of max 512 CUDA threads per block.

6.1.1 MV search with 29 x 29 block size

Listing 6.1: Computing the indexes

```
int cur_block_frame_index = blockIdx.y*WIDTH*BLOCKDIM +
    blockIdx.x*4;
int top_left_x = blockIdx.x * BLOCKDIM - SEARCHRANGE;
int top_left_y = blockIdx.y * BLOCKDIM - SEARCHRANGE;

struct mv this_mv;
this_mv.x = threadIdx.x - 14;
this_mv.y = threadIdx.y - 14;

if(top_left_x < 0)
{
  this_mv.x += (top_left_x * -1);
  top_left_x = 0;
}

if(top_left_y < 0)
{
  this_mv.y += (top_left_y * -1);
  top_left_y = 0;
}
.
.
.
```

The first consideration is how to map the pixel data. We create one CUDA block for each image block in the frame. That is 480 x 270, a total of 129600 blocks. With a search frame of 32 x 32 pixels, there are 841 SAD values to

be computed (29 x 29). Each CUDA block is responsible for computing all the SAD values within the search frame for one image block. We assign one thread per SAD computation, which gives 841 threads per CUDA block. The total amount of SAD values to be computed for one frame is 108 993 600.

Listing 6.1 shows how each CUDA thread can compute the index to the 4 x 4 pixel area in the search frame it is going to use for SAD computation. The pixel data are stored in a one dimensional array, so an index can be computed by index in y direction times the total number of pixels in x direction plus the index in x direction. First, each thread computes the cur_block_frame_index, which is the index to the top left pixel in the current frame. This is done by using the built in CUDA variables blockIdx.x and blockIdx.x which uniquely identifies a thread block. The value of cur_block_frame_index is the same for all the threads in a thread block. In addition, the threads needs an index in the reference frame for the 4 x 4 block it is going to use for SAD computation. It also computes the value of this_mv which is the motion vector, the offset from the block in the current frame, to the block in the reference frame where this thread computes a SAD value. Each thread computes the x and y index of the search frame which is the top_left_x and top_left_y. This variables are adjusted accordingly, together with the motion vector value, if the search frame is outside the border of the frame. Finally, the index in the reference frame can be computed based on the indexes of the search frame and built in CUDA variables.

Listing 6.2 shows how each SAD value is computed using the CUDA built in SAD function. After the computation is complete, the value is stored in an array in shared memory. The motion vector which corresponds to the SAD value, is also stored in shared memory. The one of these motion vectors which corresponds to the least SAD value is going to be used as the motion vector for the image block.

After the SAD values are computed, we need to find the least SAD value and store its corresponding MV. To find the least SAD value we use a thread

Listing 6.2: SAD computation

```
int sad = 0;
for(int i=0; i<4; i++)
{
  for(int j=0; j<4; j++)
  {
    sad = __usad(cur_frame[cur_block_frame_index + i*WIDTH +
        j], ref_frame[read_index + i*WIDTH + j], sad);
  }
}

shared_sads[thread_index] = sad;
shared_mvs[thread_index] = this_mv;
```

reduction algorithm discussed in [8] and [1].

6.3 shows how this can be implemented. For each iteration, one CUDA thread compares to SAD values, one SAD value with index in the shared array corresponding to the thread index, and one SAD value which has index equals thread index plus half of the number of remaining SAD which still needs to be compared. The least of the two SAD values are stored in the shared array with index equals the thread index, together with the motion vector in a corresponding array. That is the lowest index of the two. Each iteration, the number of active threads are halved, and the remaining threads compute the least of two more SAD values. When the reduction finishes, the least SAD value and the best motion vector is in index 0 in the shared arrays. Because there 841 SAD values, an odd number, one more comparison has to be made. That is comparing the SAD value in index 840 (SAD number 841) to the SAD value in index 0. One CUDA thread, the thread with index 0, compares the two remaining SAD values and writes the corresponding motion vector to global memory.

Listing 6.3: Thread reduction

```
int total_sads = 840;
  while(total_sads > 1)
  {
    int half_sads = total_sads >> 1;
    if(thread_index < half_sads)
    {
      int tmp_sad = shared_sads[thread_index + half_sads];
      struct mv tmp_mv = shared_mvs[thread_index + half_sads];

      if(tmp_sad < shared_sads[thread_index])
      {
        shared_sads[thread_index] = tmp_sad;
        shared_mvs[thread_index] = tmp_mv;
      }
    }
  __syncthreads();
  total_sads = total_sads >> 1;
  }
```

6.1.2 Reducing the block size to 29 x 15

When reducing the CUDA block size to 29 x 15, most of the threads now must compute two SAD values instead of one. Because the number of threads is reduced, each thread must do more work. In the first iteration, each thread computes one SAD value, and stores it in shared memory together with its corresponding motion vector in a separate array. Then the threads with with threadIdx.y < 14 computes the remaining SADs. This is a branch causing divergent warps. Sometimes this is necessary.

6.2 CUDA implementation using shared memory for storing pixel data while doing SAD computations

We want to test the benefits of shared memory, and look at differences in increase in performance between the GF100 architecture and the GT200 architecture.

The second decision is how to choose the number of threads, if the number per block should be chosen based on the number of SADs to compute or the amount data to read.

When making an implementation using shared memory we have to decide if the number of CUDA threads per block is to be chosen based on the SAD values to be computed or the number of pixel values to be read from global memory. If the number of threads are based on the amount of pixel values to be read from the global memory into the shared memory, this could give a better memory read performance because of coalesced reads and writes to global memory. However, assigning more threads for the memory read operations into sheared memory, requires more threads to be inactive during the SAD computations as the number of SADs to be computed in a row is equal to the number of pixels in a row - 3. If we decide to set the amount of CUDA threads based on the amount of SADs we do not need to have inactive threads during the SAD computations, however, theres is instead a need for doing reads into global memory in more iterations.

When we allocate shared memory in a CUDA block for the computations, the data type can be a performance issue. Because the pixels are of byte size, the data structure in shared memory can be a two dimensional char array. This results in a higher amount of shared memory bank conflicts, causing the memory access to be serialized when multiple threads access the same bank at the same time. An argument for doing the implementation

with a data structure using chars is reduced use of shared memory, as an int implementation use uses four times the amount of memory. When the CUDA blocks use less shared memory, more CUDA blocks may be executed in parallel on chip and this give a give a higher performance even if the implementation causes more bank conflicts.

6.2.1 Storing search frame in shared memory

Listing 6.4: Writing search frame to shared memory

```
shared_search_frame[threadIdx.y][threadIdx.x] = (int)
    ref_frame[read_index];
read_index += WIDTH * 16;
shared_search_frame[threadIdx.y + 16][threadIdx.x] = (int)
    ref_frame[read_index];
__syncthreads();
```

When storing the search frame in shared memory, we use a block size of 32 x 16 threads instead of 29 x 29. This is because the global memory access of one warp of 32 threads can be coalesced to increase performance. Still using 29 X 29 threads would decrease performance as more accesses to global memory would be required. Listing 6.4 show how the data can be read in to shared memory. It is important to have a barrier synchronization after the data is written to shared memory using the CUDA built in function ___syncthreads(). This is to ensure that all the data are available before the SAD computation starts. After the data are written to shared memory, the SAD values can be computed in the same procedure as in 6.2, only with reference to shared memory instead of global memory.

6.3 Motion vector prediction on GPU

Motion vector prediction can be a complicated task to implement on a GPU using CUDA because of the dependencies between the Motion vectors. We implement a diagonal approach for computing the motion vector predictions. The algorithm starts in the top left corner of the frame and computes the motion vector predictions diagonally across the frame to the bottom right corner.

To achieve this, we need an implementation of a CUDA kernel with total number of threads equal to the maximal number of motion vector predictions which can be computed in parallel. The maximal number which can be computed in parallel is equal to the maximal number of image blocks where the motion vector predictions do not depend on each other. It is also necessary to be able to make synchronizations between all the threads. Therefor, all the threads must be in the same CUDA block, and we use a kernel function with one CUDA block having a total of max mvps in parallel number of threads. Each of the threads also need to know the index for where to compute the MVp. To achieve this, we pre-compute the all the indexes.

Motion vector prediction is not an easy task to parallelize, and it is not possible to fully utilize the parallel processing capabilities of the GPU hardware because of the dependencies between data. In these situations it is important to decide if the computations still should be made on the GPU, or if it is better to transfer the data back to the host, and make the computations on the CPU. How to find the best solution here is affected by the how the video encoding computations are implemented after the motion vector prediction. If we continue to make computations on the GPU the cost of memory transfer back to the device, after the CPU has made the motion vector predictions, have to be taken into account.

We have implemented two different approaches for managing the threads. One approach is to manage the number of active threads on the host, with

several calls to the same kernel increasing the number of threads for each call. Every time the kernel is called, each CUDA thread computes two MVPÕs. Then the host increases the number of active threads by one an makes a new call to the kernel. This process continues until the maximum number of active threads is reached. This is 239 threads, which is equal to the maximum number of blocks which can be motion vector predicted in parallel. Maximum parallelization can be done for 62 iterations, then the host has to reduce the number of active threads for each kernel call.

The other approach is to is to make only one call to a CUDA kernel from the host, and have the kernel mange the number of active threads in all the iterations through the blocks in the frame. In each iteration, the active CUDA threads have to be able to access the correct index of the frame block for which the thread is going to compute the MVp. The indexes are pre-computed and stored in an array.

6.3.1 Managing active threads on the GPU kernel

6.5 shows how the motion vector predictions can be computed using several iterations trough a diagonal pattern in the search frame with active threads managed in the kernel and pre-computed indexes to the frame blocks. Each iteration, one thread computes two MV predictions. The index to the image block, which the thread is going to compute a motion vector prediction to, is stored in the indexes array. The block for each CUDA thread changes each iteration, and the index to the block can be retrieved based on thread index and iteration number. Each thread computes two MV predictions. Then the degree of parallelism increases by one and the number of active threads are increased accordingly. It is important to use the barrier synchronization_syncthreads() to ensure all threads have finished computation before moving to the next iteration. Because of the need for synchronizing the threads, all threads must be in the same CUDA block. Therefore, the kernel only has a grid size of one block, and 239 threads per block, which

is the maximum number of threads which can run in parallel during this computation. When the number of active threads reaches 238, the loop ends and 239 motion vector predictions are processed in parallel 62 times. Then the loop in 6.5 is repeated, only now the number of active threads decreases as the processing moves towards the bottom right of the frame.

6.3.2 Managing active threads on the CPU (host)

6.6 shows an example of managing the threads on the host, in the CPU code. Instead of launching all 239 CUDA threads at once, and have inactive threads on the kernel, there is made several calls to the CUDA kernel where the number of active threads is increased each time. The processing continues the same way as the kernel managed threads, with computing 239 MV predictions in parallel 62 times, and then decrease the parallelism towards the bottom right corner for the frame.

Listing 6.5: MV prediction with kernel managed threads

```
for(i=0; i<238; i++)
{
  if(thread_index < active_threads)
  {
    index = indexes[thread_index * MAXTHREADS + iteration_nr];

    int predmvx = compute_median(mvdiffs[index-1].mv_x,
        mvdiffs[index-NUMBLOCKSX].mv_x, mvdiffs[index -
        (NUMBLOCKSX + 1)].mv_x);
    int predmvy = compute_median(mvdiffs[index-1].mv_y,
        mvdiffs[index-NUMBLOCKSX].mv_y, mvdiffs[index -
        (NUMBLOCKSX + 1)].mv_y);

    mvdiffs[index].mv_x = mvs[index].mv_x - predmvx;
    mvdiffs[index].mv_y = mvs[index].mv_y - predmvy;

    index = indexes[thread_index * MAXTHREADS + iteration_nr
       + 1];

    __syncthreads();

    predmvx = compute_median(mvdiffs[index-1].mv_x,
        mvdiffs[index-NUMBLOCKSX].mv_x, mvdiffs[index -
        (NUMBLOCKSX + 1)].mv_x);
    predmvy = compute_median(mvdiffs[index-1].mv_y,
        mvdiffs[index-NUMBLOCKSX].mv_y, mvdiffs[index -
        (NUMBLOCKSX + 1)].mv_y);
    mvdiffs[index].mv_x = mvs[index].mv_x - predmvx;
    mvdiffs[index].mv_y = mvs[index].mv_y - predmvy;
  }
  active_threads ++;
  iteration_nr += 2;
  __syncthreads();
}
.
.
.
```

Listing 6.6: MV prediction with host managed threads

```
for(i=1; i<239; i++)
  {
    mv_pred_kernel<<<1, i>>>(d_mvs, d_mvdiffs, (i-1)*2,
        d_indexes);
    cudaThreadSynchronize();
  }
```

Chapter 7

Experiments and results

All results in CUDA tests are running time on the GPU for the specified kernel function in isolation. In these experiments we have not considered the cost of memory copy to and from the GPU. This cost must also be taken into consideration in a real application. The CUDA kernels are tested on the GT280 and the GTX480 GPUs from NVIDIA. All the experiments are made doing a computation on 1000 test frames to measure the performance. The results are compared to a serial CPU implementation. The purpose of the comparison, is not to decide if the CPU or GPU is the best computing device, as the CPU implementation could be further optimized using e.g multithreading and SSE instructions, but rather give a guidance for how well suited the computations are for offloading to GPU.

7.1 MV search

In this experiment we test SAD computation for a full search algorithm with a search range of 32 x 32 pixels, where 29 x 29 SAD values has to be computed. In this experiment the computations are made without reading image data to shared memory, and the SAD values are computed using indexing to global

memory. Shared memory are used for storing computed SAD values and motion vectors. The results on both GPUs show that a small block size is preferable for achieving the highest performance.

We start by a SAD computation with a search frame of 32 x 32 pixels and block size of 4x4 pixels. That gives 29 x 29 SAD values, a total of 841.

To make the computations, the CUDA threads need to be mapped to the data structure. A frame of 1080 x 1920 pixels gives 270 x 480 blocks which are going to have theirs SAD values for all pixels within the search frame computed. The CPU implementation has a performance of approximately 0.5 Frames Per Second (FPS).

7.1.1 Block size 29 x 29

To map the CUDA threads to the data structure, we create one CUDA block for each block in the frame. That is 270 X 480 CUDA blocks. Each CUDA block, corresponding to one image block, is then responsible for computing all the SAD values within the search frame for that block. One thread computes one SAD. We then have 29 x 29 threads, equals 841, which corresponds to the SAD values which needs to be computed a search frame.

After the SAD values are computed, the least sad value is computed by performing a thread reduction. Compared to a thread reduction algorithm, having one CUDA thread going through the computed SADs to find the least value would not perform well. A reduction algorithm is therefor used to utilize the threads running in parallel within a CUDA block.

This implementation has a performance of 1000 frames in 57.5 seconds, or 17.4 Frames Per Second (FPS). Although, this is not an optimized CUDA implementation for the GPU, the performance increase is still large compared to the CPU implementation. Thus, we can conclude, the MV full search is an algorithm which conforms good to the parallel processing hardware of the

GPU. We use the CUDA Visual profiler to further examine how the kernel utilizes the hardware.

Listing 7.1: Kernel profiling

```
Kernel details : Grid size: 480 x 270, Block size: 29 x 29 x 1
Register Ratio              = 0.625  ( 20480 / 32768 ) [22
    registers per thread]
Shared Memory Ratio = 0.145833 ( 7168 / 49152 ) [6728 bytes per
    Block]
Active Blocks per SM  = 1 : 8
Active threads per SM = 841 : 1536
Occupancy             = 0.5625  ( 27 / 48 )
Achieved occupancy  = 0.5625  (on 15 SMs)
Occupancy limiting factor = Block-Size
```

On the current hardware, there can be 1536 active threads divided among 8 thread blocks. To fully occupy the hardware, there can e.g be 3 thread blocks with 512 threads in each block or 8 thread blocks with 192 threads in each block. The kernel we now tested had 841 threads in one thread block, which makes it impossible for the hardware to execute more than one block at a time because adding one extra block gives 1682 threads, which would exceed the max thread limit of 1536. To further utilize more of the processing capabilities, the kernel must have fewer threads per CUDA block.

7.1.2 Block size 29 x 15

We reduce the number of threads in a CUDA block to 29 x 15, a total of 435 threads. To compute the SADs and MVs with fewer threads in the kernel, each thread now has to compute two SADs, except for one last row of threads. First all threads compute one SAD value each. Then the threads with threadIdx.y value < 15, compute the remaining SADs.

29 x 15 on GTX480

This implementation has a performance of 1000 frames in 29 seconds, or 34 FPS. This is a doubling in performance compared to the previous kernel with block size of 29 x 29 threads.

Listing 7.2: Kernel profiling

```
Kernel details : Grid size: 480 x 270, Block size: 29 x 15 x 1
Register Ratio    = 0.875  ( 28672 / 32768 ) [31 registers per
    thread]
Shared Memory Ratio = 0.291667 ( 14336 / 49152 ) [6728 bytes
    per Block]
Active Blocks per SM  = 2 : 8
Active threads per SM = 870 : 1536
Occupancy   = 0.583333  ( 28 / 48 )
Achieved occupancy  = 0.583333  (on 15 SMs)
Occupancy limiting factor = Registers
```

The profiling of the new kernel shows a small increase in threads per SM from 841 in the kernel with 29 x 29 block size to 870 in the kernel with 29 x 15 block size. There are more utilization of the processing resources in this kernel. Most of the increase in performance is probably due to the SMs ability to switch execution among the thread blocks, when threads in one block are waiting for memory operations. Further use of hardware resources is limited by register usage. Each thread consume registers from the register file of an SM. When all the registers of an SM are used, no more threads may be scheduled to that SM. It is also important to notice that threads are scheduled to SMs in hole thread blocks. There has to be available registers for all threads in a thread block for more threads to be scheduled to a specific SM.

29 x 15 on GT280

This implementation has a performance of 1000 frames in 257 seconds, or 3.9 FPS. This performance is much lower than the GTX480 which has 35.4 FPS for the 29 x 15 thread block kernel. This experiment shows a big difference in performance between the two generations of GPUs, where the GTX480 is over 9 times faster than the GT280. The difference in performance is first and foremost due to the lack of global memory caching on the GT280.

7.1.3 Block size 29 x 15 with search frame in shared memory on GT280

Search frame in shared memory on GT280

This implementation has a performance of 1000 frames in 53.5 seconds, or 18.5 FPS. This experiment shows the importance of using shared memory for computations on the previous generations of GPUs, which do not have global memory caching. The implementation which uses shared memory for caching is about 4.7 times faster compared to the implementation which does not store in shared memory before the SAD computations begin. The reason use of shared memory gives the increase in performance is because each data element in the search frame needs to be accessed several times by different threads. Using one thread per SAD value with block size of 4x4 requires each pixel value to be read 16 times per search frame. Without use of shared memory, the pixel values must be read from the high latency global memory each time. Using shared memory, the threads in one block can cooperate using coalesced access to global memory, which further increases performance, when reading the search frame in to shared memory. Then the low latency shared memory is used when accessing the pixel values during SAD computations.

Search frame in shared memory on GTX480

This implementation has a performance of 1000 frames in 31 seconds, or 32 FPS. This is a reduced performance compared to the implementation which does not store the search frame in shared memory. In this experiment, the GTX480 does not benefit from increased use of shared memory because of its caching hierarchy. It is more beneficial to rely on the cache for increased performance, than to store the data in shared memory before the computations begin.

7.2 MV prediction

We test the performance of MV prediction on the GPU using different approaches. Because this is a computation with data dependencies, a first and most simple approach is to make the computation of MV predictions on the GPU using only one CUDA thread, and thus make a serial implementation for the GPU. This is also useful for testing the performance of the GPU for serial processing with one thread. Further, we parallelize the processing by using a diagonal pattern from top left of the search frame to the bottom right. The diagonal pattern is used because parallel processing can only be done in a specific order because the dependencies between predictions of motion vectors requires the processing to be made in a specific order. Because the degree of parallelism changes during the processing of the frame from the top left to the bottom right, the threads which can run in parallel also changes. The number of active threads can be managed from the CPU part of the code, or within the CUDA kernel. We test the to different approaches.

7.2.1 One CUDA thread

In this experiment we test the performance of doing MV prediction with one CUDA thread. This is a serial implementation which does not utilize the parallel capabilities of the GPU. A first approach for general computations on a GPU where parts where parts of the computations have data dependencies, could be to do the processing in serial in one CUDA thread. This can be beneficial if the cost of copying data back to the CPU for doing the serial processing here and then copy the data back to the GPU for further processing, is higher than doing the processing in serial with one thread on the GPU. This scenario assumes further processing on the GPU after the serial processing of the data dependent parts. An other argument for serial processing on the GPU, can be if the CPU is already occupied doing some other processing and the total performance of the application decreases by moving more computations to the CPU.

One thread on GTX480

The experiment shows a performance of 46 seconds for 1000 frames, or 21.7 frames per second. Compared to the CPU implementation, which finishes computation of motion vector predictions for 1000 frames in 0.68 seconds, or approximately 1470 FPS. This shows that the CPU is 67 - 68 times faster in this experiment where only one CUDA thread is used. It shows that the GPU is not well suited for serial execution. This is because of the reduced processing powers of one core in favor of increased parallel processing powers and with only one thread, the GPU can not utilize the bandwidth and coalesced access to global memory and suffers from increased latency for memory access compared to the CPU.

One thread on GT280

This implementation has a performance of approximately 4.4 FPS. 1000 frames in 228 seconds. Compared to the execution time on the GT280, the GTX480 is almost five times faster.

7.2.2 Parallel implementation with host managed threads

The experiment uses a diagonal pattern with precomputed indexes for each thread. The array of indexes maps and the iteration number to the MV prediction the specific thread is going to compute. The number which can be active are varying during the computation across the frame from the top left corner to the bottom right. In this experiment we mange the threads on the host, using multiple calls to the same CUDA kernel with different number of active threads in the kernel launches.

Host managed threads on GTX480

This implementation has a performance of 71 FPS with 1000 frames in 14 seconds. The kernel profiling in 7.3 shows that only one SM is being used. This gives a potential for higher performance by better use of the parallel capabilities of the CPU. The low register ratio shows that more state can be stored in each thread without necessarily having negative effect on performance. The active threads per SM and occupancy does not need to be higher because this is the highest degree of parallelism in this computation. To further utilize parallel processing powers, the processing must be distributed among more SMs.

Listing 7.3: Kernel profiling

```
Kernel details : Grid size: 1 x 1, Block size: 239 x 1 x 1
Register Ratio    = 0.125  ( 4096 / 32768 ) [13 registers per
    thread]
Shared Memory Ratio = 0 ( 0 / 49152 ) [0 bytes per Block]
Active Blocks per SM  = 1 : 8
Active threads per SM = 239 : 1536
Occupancy    = 0.166667  ( 8 / 48 )
Max achieved occupancy  = 0.166667  (on 1 SMs)
Min achieved occupancy  = 0  (on 14 SMs)
Warning: Grid Size (1) is less than number of available SMs
    (15).
```

Host managed threads on GTX280

This implementation has a performance of 90 FPS with 1000 frames in 11.1 seconds. This is actually faster than the GTX480, and it may seem as if the GT280 can have higher performance than GTX480 in some cases where only one SM is used and the same data elements does not need to be accessed many times.

7.2.3 Parallel MV prediction with kernel managed threads

An other way to mange the threads, than to make several calls to the same kernel with different block size is to manage the threads in the kernel, have a loop in the kernel iterating through the diagonal pattern in the frame, and only make one call to the kernel for each frame on the host. The kernel as a variable which holds the number of currently active threads, and for each iteration in the loop has an if test which only allows threads with thread-index to enter the computation of MV predictions.

Kernel managed threads on GTX480

This implementation has a performance of 153 FPS. 1000 frames in 6.5 seconds.

Listing 7.4: Kernel profiling

```
Occupancy analysis for kernel 'mv_pred_kernel' for context
    'Session1 : Device_0 : Context_0' :
Kernel details : Grid size: 1 x 1, Block size: 256 x 1 x 1
Register Ratio    = 0.15625  ( 5120 / 32768 ) [17 registers per
    thread]
Shared Memory Ratio = 0 ( 0 / 49152 ) [0 bytes per Block]
Active Blocks per SM  = 1 : 8
Active threads per SM = 256 : 1536
Occupancy    = 0.166667  ( 8 / 48 )
Max achieved occupancy  = 0.166667  (on 1 SMs)
Min achieved occupancy  = 0  (on 14 SMs)
Warning: Grid Size (1) is less than number of available SMs
    (15).
```

The kernel only runs on one SM, and therefore far from utilizes the potential parallelism of the GPU. The CUDA kernel only has one block because the we need to be able to synchronize the execution among the threads, and threads need to be in the same thread block for the execution to be synchronized. A potential for increasing parallelism is to run the computation on multiple frames or multiple slices in parallel, and have one slice or one frame per SM computed by one CUDA block.

Kernel managed threads on GT280

This implementation has a performance of 270 FPS. 1000 frames in 3.7 seconds. This experiment also shows a higher performance on GT280 than GTX480.

7.2.4 Second parallel implementation with host managed threads using several CUDA blocks

When managing the threads on the host, it is possible to distribute the processing among several SMs. This is done by also adjusting the blocks per grid as the active threads increases, instead of having a block size of one and only increase the threads per block as the active threads increase and decrease across the frame.

host managed threads using several CUDA blocks on GTX480

This implementation has a performance of 120 FPS. 1000 frames in 8.3 seconds. This kernel has higher performance than the other kernel with host managed threads, but has still lower performance than the kernel which manages threads on the GPU. We can conclude from this, that there is a high cost of kernel launch if it has to be done many times, and many kernel launches should therefore be avoided.

host managed threads using several CUDA blocks on GTX280

This implementation has a performance of 103 FPS. 1000 frames in 9.7 seconds. Now, when processing is distributed among several SMs, the performance is higher on the GTX480 than the GT280.

Chapter 8

Conclusion and further work

8.0.5 Conclusion

The motion vector search algorithm conforms good to the GPU architecture because it has many computations which can be executed in random order on independent data. Computation of the best motion vector of one image block does not affect the computation of the best motion vector of an other image block. In addition, the algorithm has a high computational intensity which also makes it suitable for offloading to the GPU, compared to execution on the CPU.

Independent data, high computational intensity and the uncomplicated mapping of CUDA threads to data elements, makes a full search motion estimation algorithm a good example of computations which are very suitable for parallel processing on a GPU. We also see that, when programming in CUDA, there probably is a need for first to do an initial implementation on the GPU which then has to be improved after profiling to find the best algorithm for GPU processing.

We have seen that data dependent problems are harder to implement on the GPU, and may not utilize the computational capabilities of the GPU in

the same way as a data independent problem. In both problems finding a good way to map threads to data is important for GPU computing. With data dependent computations, it may also be more challenging to map the execution on the GPU, with the CUDA threads to the correct data. Instead of computing the mapping of threads to data in each thread, pre-computing indexes can be a solution.

We see that the caching in the newer GPU generation increases computational speed of the GPU. It also makes the development more easy as it removes the challenge of getting a good utilization of shared memory. However, in some cases, the previous architecture may still show better performance.

8.0.6 Further work

Further work can be to further explore the optimization of the MV search and MV prediction algorithms. As the experiments shows there may be possibilities of increased performance as the hardware is not fully utilized. The differences in performance when using different search ranges should also be examined.

Implementing motion estimation which uses Rate Distortion (RD) algorithms should be further examined for GPU offloading. This can be a challenging task to offload to the GPU with satisfactory performance.

MV search with half pixel and quarter pixel accuracy should also be further examined as interpolation capabilities of GPUs may make this kind of computations beneficial for GPU processing.

Finally, the GPU algorithms should also be implemented in a video encoder to test the computations in a real application.

Bibliography

[1] The supercomputing blog. http://supercomputingblog.com/cuda/cuda-tutorial-3-thread-communication/.

[2] Canter, D. Nvidia's gt200: Inside a parallel processor. http://realworldtech.com/page.cfm?ArticleID=RWT090808195242, 2008.

[3] Canter, D. Inside fermi: Nvidia's hpc push. http://realworldtech.com/page.cfm?ArticleID=RWT093009110932, 2009.

[4] Chen, W., and Hang, H. H. 264/avc motion estimation implementation on compute unified device architecture (cuda). In *Multimedia and Expo, 2008 IEEE International Conference on* (2008), Ieee, pp. 697–700.

[5] Cheung, N., Au, O., Kung, M., Wong, P., and Liu, C. Highly parallel rate-distortion optimized intra-mode decision on multicore graphics processors. *Circuits and Systems for Video Technology, IEEE Transactions on 19*, 11 (2009), 1692–1703.

[6] Cheung, N., Fan, X., Au, O., and Kung, M. Video coding on multicore graphics processors. *Signal Processing Magazine, IEEE 27*, 2 (2010), 79–89.

[7] Colic, A., Kalva, H., and Furht, B. Exploring nvidia-cuda for video coding. In *Proceedings of the first annual ACM SIGMM conference on Multimedia systems* (2010), ACM, pp. 13–22.

[8] Kirk, D., Wen-mei, W., and Hwu, W. *Programming massively parallel processors: a hands-on approach*. Morgan Kaufmann, 2010.

[9] Kung, M., Au, O., Wong, P., and Liu, C. Intra frame encoding using programmable graphics hardware. In *Proceedings of the multimedia 8th Pacific Rim conference on Advances in multimedia information processing* (2007), Springer-Verlag, pp. 609–618.

[10] Lin, Y., Li, P., Chang, C., Wu, C., Tsao, Y., and Chien, S. Multi-pass algorithm of motion estimation in video encoding for generic gpu. In *Circuits and Systems, 2006. ISCAS 2006. Proceedings. 2006 IEEE International Symposium on* (2006), IEEE, pp. 4–pp.

[11] NVIDIA. *NVIDIA CUDA C Best Practices Guide*. 2011.

[12] NVIDIA. *NVIDIA CUDA C Programming Guide*. 2011.

[13] NVIDIA. *NVIDIA GF100 Whitepaper*. 2011.

[14] Olukotun, K., and Hammond, L. The future of microprocessors. *Queue 3*, 7 (2005), 26–29.

[15] Ostermann, J., Bormans, J., List, P., Marpe, D., Narroschke, M., Pereira, F., Stockhammer, T., and Wedi, T. Video coding with h. 264/avc: tools, performance, and complexity. *Circuits and Systems magazine, IEEE 4*, 1 (2004), 7–28.

[16] Richardson, I. *H. 264 and MPEG-4 video compression*, vol. 20. Wiley Online Library, 2003.

[17] Richardson, I. *The H. 264 advanced video compression standard*. John Wiley & Sons Inc, 2010.

[18] Sanders, J., and Kandrot, E. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[19] Stensland, H., Espeland, H., Griwodz, C., and Halvorsen, P. Tips, tricks and troubles: optimizing for cell and gpu. In *Proceedings of the 20th international workshop on Network and operating systems support for digital audio and video* (2010), ACM, pp. 75–80.

[20] Sutter, H., and Larus, J. Software and the concurrency revolution. *Queue 3*, 7 (2005), 54–62.

[21] Wiegand, T., Sullivan, G., Bjontegaard, G., and Luthra, A. Overview of the h. 264/avc video coding standard. *Circuits and Systems for Video Technology, IEEE Transactions on 13*, 7 (2003), 560–576.

Appendix

The code for the experiments is available at http://heim.ifi.uio.no/magnusfh/t/tests/