UPPSALA
UNIVERSITET

Performance and power consumption of the Fermi GPU

Tobias Skoglund

Project in Computational Science: Report

January 2012

PROJECT REPORT

"Life is getting tough means God is afraid of your progress" – Courage Wolf

Abstract

In the development of high-performance scientific computing we have recently witnessed improvements in performance by the use of graphics cards and the GPU co-processor. In many applications, the performance increase is very high. However, the underlaying complexity of the GPU architecture is a limiting factor in having full utilization. In this report, we study the architecture of the Nvidia Fermi GPU and test its performance and power consumption with different computational patterns and show that further investigation is needed over the general programming guidelines, to have full utilization.

Innehåll

1 Introduction

The use of graphics processing unit (GPU) co-processor in scientific computing has shown a substantial increase in performance for computational intensive tasks. Such solutions have become successfull by using the GPU to exploite data level-parallelism in a more efficient way. It is a fact that GPUs have a theoretical peak performance that is two orders of magnitude greater than modern CPUs and are only slightly more expensive per device. The immense computational power of the GPU makes it interesting for highly parallel algorithms.

The recent success of the GPU as a computational co-processor is due to its now more simple programming model. There are two main programming models which allow developers to program in a C/C++ simlar programming language called CUDA and OpenCL. OpenCL is a general high-performance framework and is intended for heterogenous computing. CUDA is a propretary framework developed by Nvidia which is only supported by their own grphics cards.

Before the advent of CUDA and OpenCL, developers used shader programs for non-rendering tasks that were inconvenient and difficult. With CUDA, the developer can access the processing hardware more explicitly than previously (far from completely though) and, with the latest updates, CUDA now integrates with C and C++ and has a "unified memory model". Debugging has always been less than trivial in CUDA, because of the additional hardware for administering debug traces. There is, however, a analysis tool called Parallel Nsight which can measure latency, bandwidth and utilization. Still, without a proper emulator, such as Ocelot[1], debugging only possible using two devices.

HPEC (High Performance Embedded Computing) [12] is a community challenge, founded in 2002 by MIT, that formulated eight problems inspired by frequently occuring problems in image- and signal processing.

The purpose of HPEC is to encourage the implementation of the problems on different (embedded) architectures and provide a media for recording the performance results and for compareing the different implementations and achievements. It has previously been shown in [13] that five of these problems gain performance using the GPU to accelerate the computations. In this report the performance of two additional problems is presented.

In addition to their massive compute capacity, GPUs are more efficient than CPUs at computing and at having a lower cost in power. Today, supercomputer clusters use GPUs to increase the computational power and reduce the power consumption. Currently, no proper online measurments of the GPUs power consumption for different problems have been performed. In particular, no measurments of the GPU only during execution have been made. It is however possible to measure the power consumption during execution by observing the power going through the PCI-express port and the supplementary power connectors (as used on powerful cards). In this report, a methodology for measuring the power consumption is developed and measurments are made for both HPEC problems.

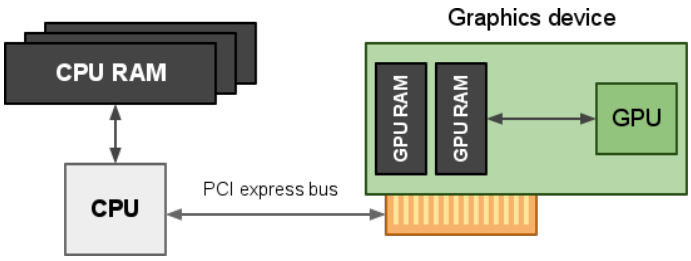
[1] http://code.google.com/p/gpuocelot/

2 The Fermi architecture

2.1 The GPU: Purpose and place

The GPU is a co-processor designed to accelerate system performance by offloading computations from the CPU. In graphics cards and integrated graphic devices the GPU is a specialized component streamlined for processing graphical data. For many years the development of GPUs focused on accelerating the graphics pipeline which typically involved calculations for moving, lighting and shading (colouring) geometric objects. To this end, the GPU was outfitted with a huge number of cores and other specialized circuitry which could process graphics primitives (points, lines and polygons) into coloured pixels.

In desktop computers the graphics device is an external device which interface with the rest of the system over a high speed Peripheral Component Interconnect (express) bus, or PCI-express. The device has an off chip RAM where the CPU stores data that is to be processed. Figure 1 illustrates the graphics card in a desktop computer setting.



Figur 1: The GPU has global memory on the device and may only access the CPU RAM via the PCIe bus.

CPU architecture development traditionally focuses on a small (two to eight) number of interconnected cores having a relatively high clock frequency. To compare, CPUs usually have a few very complex cores which work fast and have access to a lot of resources, while GPUs work at a slower rate but have a huge amount of cores which can simultaneously perform an instruction on lots of data simultaneously. This concept is refered to as Single Instruction Multiple Data (SIMD). The GPU is a heavy duty SIMD processor with a highly parallel structure capable of billions of floating-point operations per second (FLOPS). The sheer compute capacity and relatively low cost of the GPU have attracted attention from several research fields which, in general, require long running calculations on very expensive server clusters.

2.2 The Fermi GPU: History and future

In 2006 Nvidia brought out the first G80 based graphics card to the market. The G80 architecture introduced several changes to the previous architecture and kicked-off Nvidiaś new unified computing architecture called CUDA. The new architecture supported the C-like CUDA language, which appealed to the large amount of programmers, unfamiliar with shader programming languages. Because of its similarities with previous generations of graphics cards, the architecture only supported single floating point precision. The next generation of this architecture was the GT200 (launched in

2008) which included support for double floating point precision, more complex memory accessing and several other performance improvements. In 2010 Nvidia released their first graphics card which was based on their next GPU architecture codenamed Fermi.

Nvidia envisioned their next generation of GPUs to evolve into a more general purpose architecture by moving to a more CPU-like architecture. New features include a cache hierarchy, increased double precision performance, ECC (error-correcting code) and C++ compatibility. Nvidia claim [1] that these new features are an answer to the demands of the GPGPU (General-purpose computing on graphics processing units) community which, for many application, has been inclined to move into a CUDA environment but hesitant since there are still several key design features which first needed to be supported. Nvidia hopes that Fermi can overcome these issues by appealing to computing areas that require hardware support for ECC, such as finance, database- and mediacl applications [2]. The main new features beside increased compute power include:

- Improved double precision performance and extended support for the IEEE-754 standard.
- Error-correcting code (ECC).
- A cache hierarchy.
- Faster context switching and ability to run different programs concurrently.
- Full duplex transfers between the host and device.

Although Nvidia promised a lot in the Fermi architecture, some features were in practice unavailable for lower price range devices. The Fermi architecture was first introduced in the 400 series which had parts of the compute units and memory controllers disabled. ECC was unavailable for the entry- and mid-range cards also, instead it was limited to Nvidias high-end graphics card series Tesla for the High-performance computing market. The Tesla-series is also the only graphics card which have full duplex capabilities [3]. Its successor, the 500 series, was launched somewhat later and had a refreshed Fermi-GPU. However, ECC, full duplex and full double precision performance is to date still only limited to the Tesla series. The 500 series is at the time of writing for this report available for hundreds USD depending on what model while the Tesla series is no less than a thousand USD.

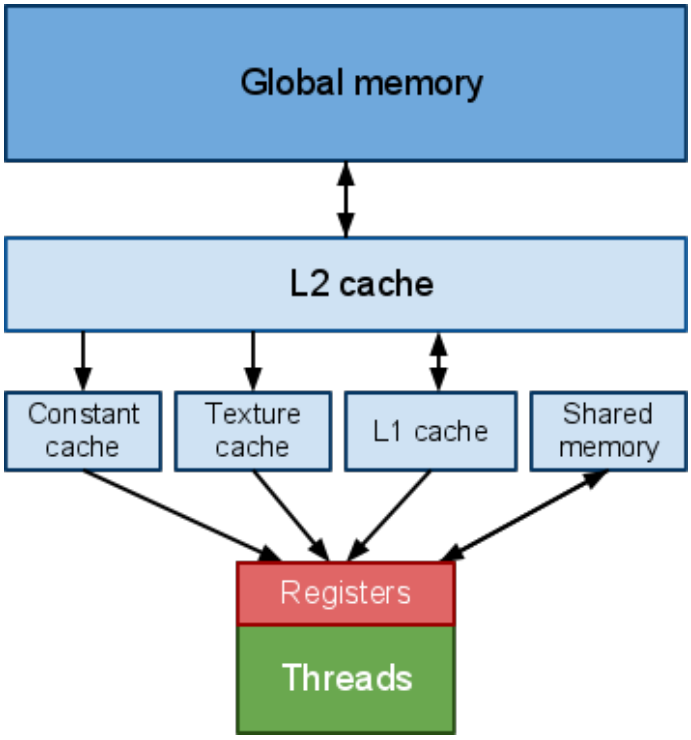
2.3 The Fermi architecture

In this chapter, the architecture of the Fermi-GPU is detailed. First the memory model is presented explaining how data is streamed to the execution units. Then the execution unit is presented in more detail and its memory architecture is unveiled. Then the execution unit is described and details about the arithmetic processors and how they achieve double precision are discussed. Lastly, the interface between the GPU and the CPU is explained and some performance metrics are presented.

2.3.1 Memory overview

The Fermi architecture has a three-leveled memory hierarchy with six different memory pools. The global memory serves as an off-chip semi-coherent memory which is

accessible by all threads, similar to the Random Access Memory (RAM) of the CPU. Since the GT200 architecture was established, a new feature is the introduction of an on-chip L2 cache which handles memory requests to the global memory from the execution units and cache them accordingly. Each execution unit has four different memory pools which are used in various ways to exploit memory locality and limit the number of memory requests to global memory. An overview of the memory model is displayed in Figure 2.



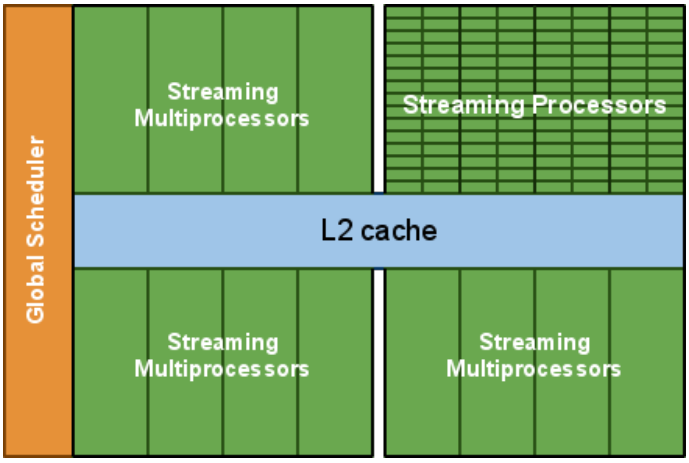
Figur 2: The memory hierarchy of the Fermi GPU.

2.3.2 Execution units

Execution within the GPU focus on the execution of threads. Threads in CUDA are much more lightweight than threads in a CPU. Since registers are allocated to each thread, swapping between concurrent threads is faster than context switching on the CPU where registers have to be spilled to another memory pool located far away. The Nvidia GPU computing model defines a block of 32 threads as a warp. A warp is the native computational unit in CUDA and the architecture and programming model is highly streamlined with respect to the size of the warp. For instance, scheduling only considers warps, not single threads, and dispatches entire warps to different execution units.

Fermi is equipped with 16 streaming multiprocessors (SMs) in banks of four units, each of which has 32 streaming processors (SPs). This gives each full Fermi GPU a maximum of 512 single precision floating point units. Each SM has a separate L1 cache space which is shared among all 32 SPs. Inside a streaming multiprocessor

we find the evolved vertex and pixel shader units (the streaming processors). The streaming processor is a unified processor (introduced in the G80 architecture) which has combined the vertex and pixel shader into a unified processor. Together, a single SM issues instructions from its current running program to single SPs which fetch one of these instructions and execute it in parallel for all running threads in the SP.



Figur 3: The execution unit hierarchy starts at the Streaming multiprocessor which issue work to its Streaming processors. Each SM is composed of 32 SPs and there are 16 SMs in groups of four.

The GPU fetches and stores data from and to a global memory pool. With Fermi, the size of the global memory is usually between hundreds of megabytes up to several gigabytes and communication with the host (CPU) is available through a PCI-E 2.0 x16 interface. The transfer rate between the host and the device (GPU) is thus bounded by the PCI-E x16 buss which is capable of 8 GB/s (or 2 byte at 4 giga transfers per second). This is unchanged since the last (GT200) architecture generation, however, Nvidia have improved the memory controller unit in Fermi so that the device can transfer memory from and to the CPU in both direction simultaneously (only available on professional devices).

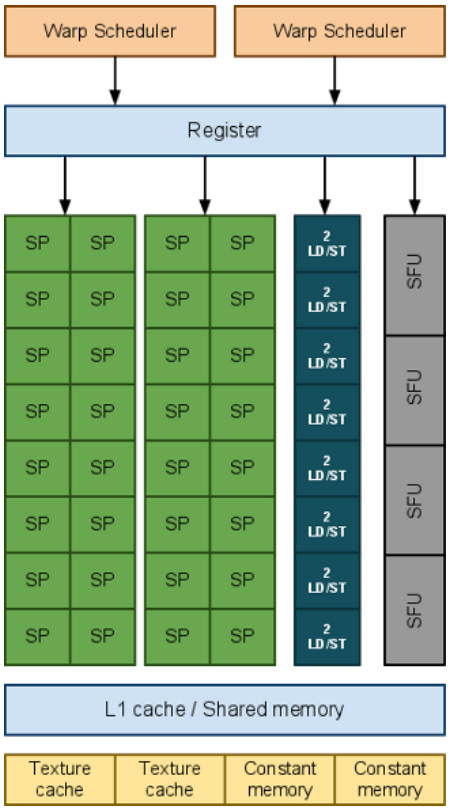
In Fermi, all transactions are cached to further exploit data locality and increase throughput. The application can help memory alignment by using data types that fit the alignment requirement. If a request is larger than 32 bytes it is divided into two separate requests. Each request is then further reduced to cache line requests which either is accessed at the speed of the L1 or L2 cache on a cache hit, otherwise it looks to global memory.

The streaming multiprocessor arrays share 768KiB of L2 cache which handle reading and writing operations to global memory. It is also responsible for handling texture operations (see texture cache below).

Programs on the GPU, referred to as kernels, are run by a global scheduler which schedules work to different SMs. A new feature in Fermi is the ability to have concurrent kernels running, one for each SM. Furthermore, context switching between kernels have improved by up to ten times. This is particularly important for small kernels which can be scheduled more efficiently than in previous architectures.

2.3.3 Streaming multiprocessor (SM)

A streaming multiprocessor, SM, is to a Fermi GPU what a core is to a multi-core CPU, and is a composition of fast memory, co-processors and execution units. Nvidia names the execution units CUDA cores but in this document they are referred to as streaming processors. The streaming processor uses the co-processors for calculating memory addresses and for calculating transcendental functions, such as sine, cosine, square root and logarithm. The memory address is calculated in load-store units and the transcendental functions are calculated in Special Function Units (SFUs). The fast memory is available as registers, shared memory and L1 cache. The register is a storage unit for values that are used in executing the instructions and the shared memory and L1 cache are accessed last. Figure 4 shows the components of an SM.



Figur 4: The first Fermi generation SM. These SMs execute 16 threads (one group of 32 threads is named a *warp*) each cycle which can use either 16 SPs, 16 load/store units for address calculation or 4 Special function units. Note: the refurnished Fermi chip carries an additional SP lane.

Shared memory is a fast and local type of memory. It is only accessible within an SM and is used for inter-thread communication. In Fermi, the shared memory is shares memory with the L1 cache. The size is controlled by the programmer who can choose to configurable it to either 16 kb or 48 kb.

The L1 cache is a data cache for recently used and prefetched data. The cache is of

write-back design and its secondary memory pool is the L2 cache.

2.4 Memory details

2.4.1 Global memory

The global memory (GPU-RAM) is an off-chip memory pool which is sometimes referred to as device memory. It is the largest memory available to the GPU and typically ranges from 512 MB to 2GB. Before data is available to the GPU, it has to be explicitly allocated and transferred to the device memory by the CPU over the PCIe-bus. Because it is an off-chip resource, the latency is very high in comparison to any other memory resources on the GPU.

2.4.2 L2 cache

At 768 KiB, the L2 cache is the largest on-chip memory available. All caches exploit a type of data dependence called spacial and temporal locality by buffering (caching) regularly used data. The cache is automated and the programmer needs to consider cache locality to benefit from it. By serving the execution units with cached memory, this memory can efficiently reduce the cost of reading and writing to global memory.

The data in L2 cache is semi-coherent, which means that it is not necessarily mirrored to global memory immediately. Instead, the data is marked as dirty until it is evicted from cache, in which case the cache must first flush dirty data to global memory before synchronous access is allowed. This concept is called write-back. L2 cache is the fastest memory for inter-SM communication. L1 cache is restricted to communication within the SM only, so any modifications to data within an SM has to pass through the L2 cache before it is available to other SMs.

Coalescing memory accesses
The global memory is only accessible in 32, 64 or 128 byte aligned transactions. Accesses from threads within the same warp are coalesced into one or more transaction. Consequently, the remaining free space in any transaction results in a loss of throughput.

2.4.3 SM memory structure

L1 cache

L1 cache is a per SM type of memory that is smaller in size but faster than L2 cache. It has the same roll as an ordinary L1 CPU cache. On a cache miss the cache has to to request that data from the L2 cache which in worst case refers to global memory causing a severe drop in data availability (see table 1).

Shared memory

Shared memory is a, per SM, user controlled cache located on chip with low latency. Executing threads are working in blocks which can have an arbitrary size. The only

way for threads to communicate within the block is to use shared memory. The shared memory is a fast small sized memory were the threads can store data. Usually shared memory is used to hide the latency to global memory. A shared memory resource is declared with the CUDA qualifier `__shared__` .

Bank conflicts in shared memory

Shared memory is structured in 32 32-bit (a word) wide memory banks. Consecutive 32-bit words are stored in successive memory banks. When two or more threads require access to bytes in different words within the same bank, a certain type of conflicts occur, called bank conflicts. This happens for instance when there are two elements in a bank which are accessed by two or more threads. If all threads access the same element within the bank, then the shared memory can perform a broadcasting operation to all accessing threads.

Texture memory

The texture cache is mainly used by the texture units, which can interpolate between data. The texture cache is read–only memory that has greater importance for graphics applications. Texture reading operations from global memory is cached in order to decrease global memory bandwidth. The texture memory is designed for optimal performance using two–dimensional data sets, since this is a frequent format in computer graphics.

Constant memory

Constant memory resides in global memory and is cached in constant caches inside each SM. The constant memory is read–only and used to store values that are declared with the C-style qualifier `const`.

2.4.4 Latency table

In Table 1 the access times of the different memory pools in the SM is listed.

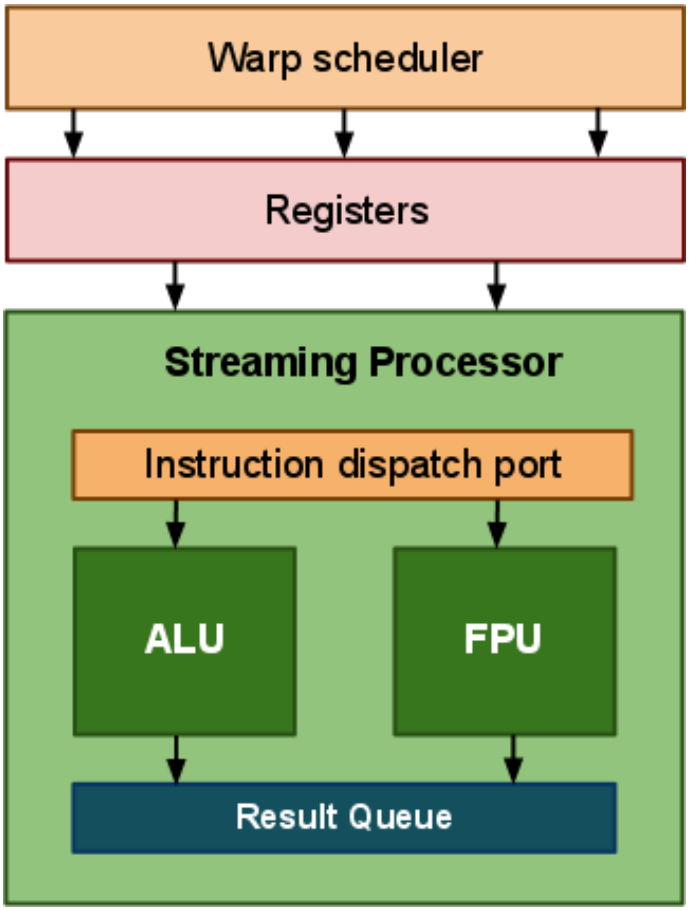
Memory type	**Latency (cycles)**	**Notes**
Register	0	
Shared	0	
Constant	~ 0	First touch high cost, then low.
Texture	> 100	
Global	> 100	

Tabell 1: Access time to different memory resources within the chip is very different. In this table the approximate latency for an executing thread is listed.

2.5 Streaming processor and precision details

2.5.1 The streaming processor (CUDA core)

A SM has two sets of SP arrays and each SP is equipped with a dedicated 32-bit integer and floating point Arithmetic Logic Units (ALUs). The ALUs share instruction scheduler which means that each thread can execute on either the ALU or the FPU but not simultaneously. The operations within the ALU and FPU are nowadays 32-bit to support the IEEE 754-2008 double precision convention properly. The appliance is due to a new set of rounding modes for higher precision.



Figur 5: Inside the SP the scheduler has two separate data paths which may not be used simultaneously. Operation is hence either on the ALU or the FPU.

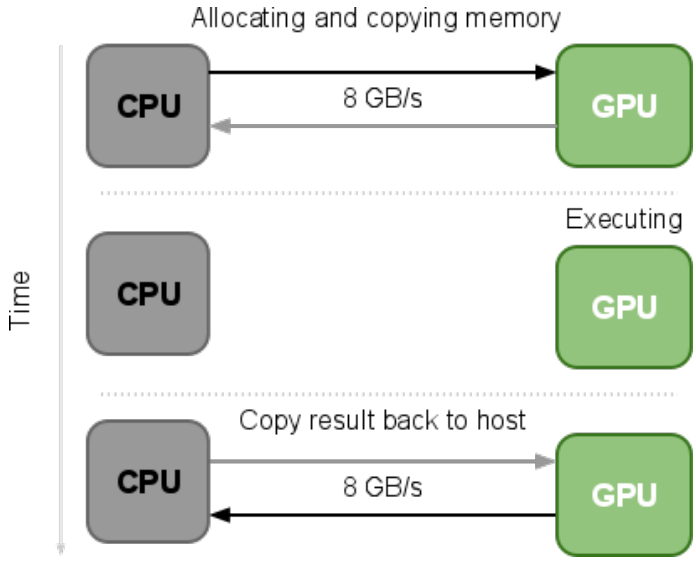
In previous architectures, only IEEE 754-1985 double precision was available but has now been upgraded to support single precision denormalized numbers, as well as the required four rounding modes: nearest, zero, positive infinity (up), negative infinity (down). The main difference between the IEEE 754-2008 and 754-1985 is that subnormal numbers were truncated to zero, which incurred a loss of accuracy. Denormalized numbers are supported by hardware in Fermi and improve denormalized number performance drastically in comparison with CPUs which do not have native hardware

support. FMAD is a floating-point multiply-add operation which, in accordance with IEEE 754-2008, is performed in one single cycle, usually one during rising edge clock signal and the other on falling edge.

When executing in a double precision mode, the SM schedules half the computations to each warp scheduler. Every clock cycle the available execution paths are thus ALU/ALU, ALU/memory, ALU/SFU and memory/ALU. See Figure 4.

2.6 The PCI-express interface

The memory bandwidth between the host and the device is bounded by the PCI-Express 2.0 x16 buss which runs at approximately 8 GB per second. As mentioned, the memory controllers now support full duplex which means that the transfer rate between the device and then back to the host again have doubled. Previously, programmers had to hide the latency by interleaving transfers to and from the device by computations.

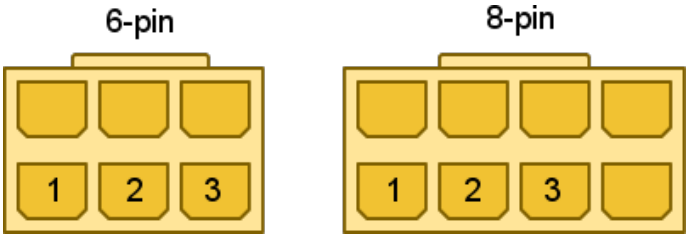


Figur 6: Highlighting the host and device relation. In this particular example, there is no data being transferred during execution.

Transfering data between the host and the device is usually the bottleneck in any GPGPU application that require a high memory bandwidth. The full duplex addresses some aspects of this problem, however, for these problems the key to high performance is still to minimize the transfers with low bandwidth. This also implies maximizing on-chip memory utilization.

Graphics cards connect to the mother board over the PCI-express X16 socket which has a +3.3V and +12V power rail. The PCI-express standard ([10]) specifies the maximum current over these rails as 3A and 5.5A respectivly, which gives a theoretical power supply of 75 Watts.

The standard also specifies an additional 6- and 8-pin connectors which delivers up to 75 Watts and 150 Watts respectively over 3 +12 Volt rails.



Figur 7: Slots 1, 2 and 3 are +12V connectors which can be connected to the side of the graphcis card.

3 Investigating HPEC algorithms

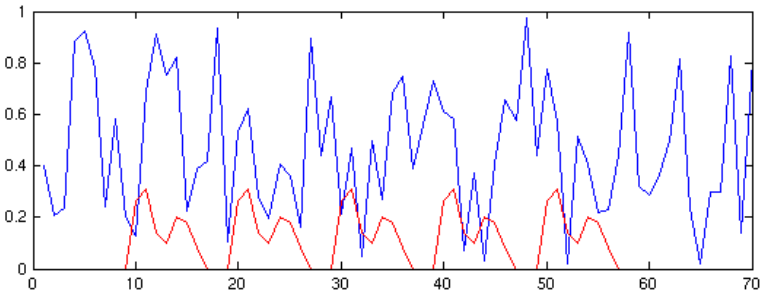
3.1 Pattern matching

In pattern matching, a test vector is matched with a set of library patterns. The objective is to find the most similar library pattern using the mean square error (MSE) as an error metric. The library pattern that gives the least value with respect to this metric is considered to be the most similar. The MSE is computed as:

$$\text{MSE} = \frac{\sum_{k=1}^{N}(w_k(a_k - t_k)^2)}{\sum_{k=1}^{N} w_k} \tag{1}$$

where w_k are a some prescribed weights.

The input test vector consists of N elements. The library patterns, here on called templates, are of size K elements, where $0 < K \leq N$. A template pattern may match any subsequence of elements along the test and in any scaled range of values. Therefore, the pattern matcher first computes the minimum MSE with respect to a shift and then with respect to a range of scaled values.



Figur 8: An example of shifting a template (repeating low curves) along a test (greater curve)

The MSE value for all shifts and scalings are stored in a local table. The shift with the minimum MSE value gives the best fit along the test and the MSE value for the best scaled template at that shift gives the final MSE score for the template. This score is

stored in a global table of MSE results. The global solution to the best template fit is found by finding the minimum global MSE score. The procedure is outlined in the following kernel benchmark description (Listing 1):

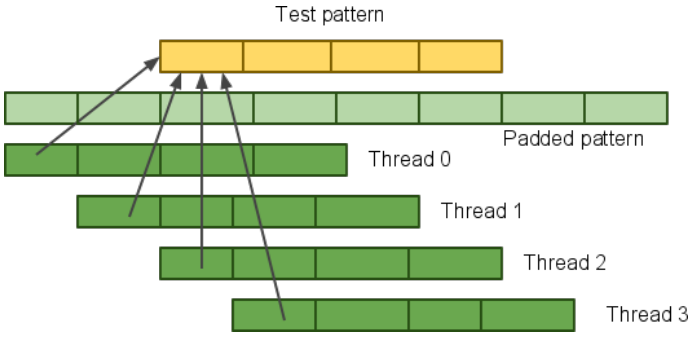
Listing 1: The implementation scheme

```
1:      for each of K patterns:
2:              for each of Sr shfit values
3:                      calculate MSE value with shifted pattern
4:              Choose shift value with smallest MSE
5:              for each of Sm magnitude values
6:                      calculate MSE value with scaled pattern
7:              Choose gain value with the smallest MSE
8:              Score-board the MSE value
9:      Choose the pattern with the smallest MSE value
```

3.2 Parallel algorithm for the pattern matching problemem

In addition to the HPEC kernel benchmark, the implementation extends to support multiple test vectors. Each test is mapped to a parallel CUDA block. All blocks have the same set of L library patterns. The number of threads per block is equal to the number of shifts or scalings plus the length of the padding which is explained below.

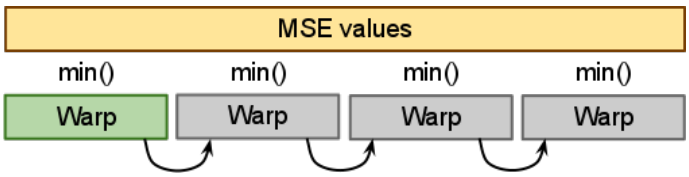
Execution within the block follows the algorithm in Listing 1. First, threads fetch the test and template library from global memory and store them in the shared memory. The test vector is padded with zeroes at both ends to allow the template to match any subsequence of the first and last elements. A thread starts calculating the MSE at its thread number position. This way, a thread calculates the MSE for a certain shift and stores the value in shared memory table. All threads synchronize after having calculated their MSE value.



Figur 9: Shifted patterns are defined by N consecutive elements starting from the offset index. Similarly, all threads receive a scaling value based on their thread id, which is used in calculating the optimal scaling for the best shift.

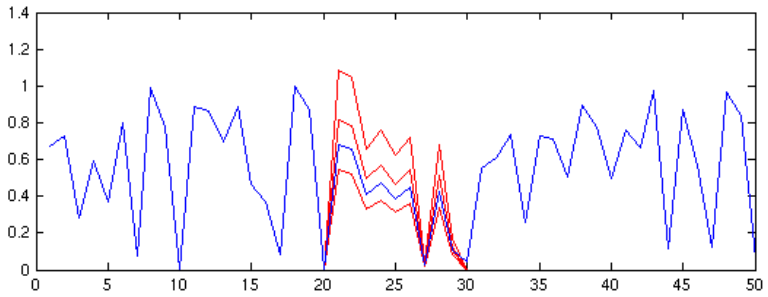
The best shift is determined by the smallest MSE value. To find this value, a minimum operation has to be performed on the local MSE table. The procedure is similar a reduction operation. If properly configured a fast way of finding the minimum value is to let a single warp loop over the table and compare the values. Preferably, the

number of threads is a multiple of 32, the size of a warp. The last step is to compare the values in the warp. This can be done in constant time by recurrsively comparing the lower half threads with the upper.



Figur 10: In finding the minimum value, a single warp loops over all table values and intermediately stores the smallest value.

Finding the optimum scale is similar to computing the shifts. First, if the number of threads is larger than the number of scalings, the supurflous threads branch of, causing some loss in performace. The other threads compute their unique scale value in some range, $[a, b]$, and compute the MSE, starting at the optimum shift offset.



Figur 11: An example of scaling where the templates (curve repeating vertically) are tested for the optimal scale value in comparison with the test (the greater curve)

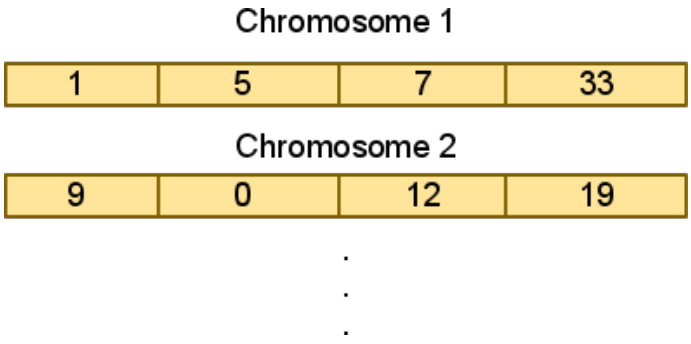
The MSE value is stored in a local table and all threads are synchronized as a last step. The minimum MSE value of the scaled MSEs is the optimal solution for the particular test.

3.3 Genetic Algorithm

Genetic algorithms utilize processes inspired by evolution, namely, to evolve from one solution state to another. Optimization schemes traditionally implement the convergent transition invariance, which states that when transcending from one iteration to the next the error must not grow. However, by use of mutation and crossover, genetic algorithms feature randomness which may produce state transitions that are either very good or very poor.

Genetic algorithms use candidate solutions, called chromosomes. The chromosomes have genes, which can be values or some other quantities of interest, and the set of genes represents the chromosome's solution. In this problem, a gene can carry an

integer number called a code (a gene). The number of possible codes is used as a program parameter.



Figur 12: Two chromosomes with four genes, where each gene represent a different number.

By randomly mutating a single gene and performing crossover between two chromosomes the values of the genes are changed from state to state. Crossover involves randomly selecting two chromosomes and a site along the chromosome, where the genes are exchanged and possibly mutated.

Furthermore, the kernel algorithm implements elitism, which states that the chromosome with the best fitness score is left unmodified from the current state to the next state. The fitness score of a particular chromosome is its evaluated score in relation to the optimal goal. Hence, letting the elite chromosome survive between states will increase the probability of convergence to the optimal goal and ensure that the error never grows between generations.

The program converges either when the optimal goal is found or when the maximum number of iterations has been reached. The complete procedure is outlined in Listing 2.

Listing 2: The implementaion scheme

```
1. Simple Genetic Algorithm ()
2. {
3.       Initialization;
4.       Evaluation;
5.       while termination criterion has not been reached
6.       {
7.                Selection_and_Reproduction;
8.                Crossover;
9.                Mutation;
10               Evaluation;
11       }
12 }
```

The program is governed by a set of parameters:

- The number of chromosome solutions in the population.
- The number of genes inside each chromosome.

- The number of codes which can be coded in a gene.
- The maximum number of iterations.
- The number of subsequent iterations a chromosome must be selected as the elite chromosome for the optimal goal to be considered found.
- The probability of a mutation event to occur.
- The probability of a crossover event to occur.

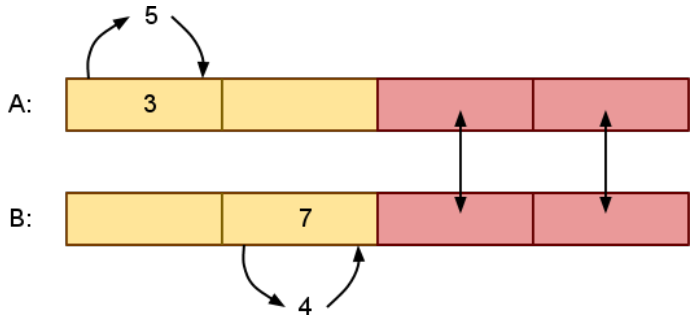
The algorithm is implemented as a two-dimensional scoreboard and an array of chromosomes. The genes inside the chromosome map to values in the scoreboard. When a chromosome is subject to mutation or crossover the map to the scoreboard is changed and the fitness of the chromosome is changed too. The fitness is stored as a one-dimensional array and needs to be passed between generations.

3.4 Parallel algorithm for the genetic algorithm problem

The treatment of each chromosome features a high level of parallelism. The bottleneck in this problem is in randomly selecting pairs of chromosomes to crossover and to compute the elite chromosome. The selection phase must be synchronized. This is hard to implement in one single kernel launch (remeber the layout of blocks and threads cannot be changed within a kernel). Instead, the problem might gain from splitting it up into smaller kernels.

In this implementation only the crossover and mutation phase are computed on the GPU. Inbetween, data needs to be transferred between the host and device back and forth. This introduces a huge performance penalty, which can not be amortized by interleaving or increasing the workload.

The kernel launch is configured with one thread per gene and the number of blocks equal to half the number of chromosomes. Each chromosome is responsible for crossing over with another chromsome. If no crossover occurs, then each thread attempts to mutate its gene for both chromosome. If crossover occurs, then each thread switches gene values and attempts to mutate them. The block with index zero perform elitism by copying a possibly mutated copy of itself into the next state. If there is an odd number of chromosomes, the last block copies a possibly mutated copy of itself into the next state. All other blocks perform selection, reproduction and also possibly crossover.



Figur 13: Mutation events on sites A(1) and B(2) and a crossover event at site three and four.

The fitness of the chromosomes is evaluated by summing the values in the scoreboard and normalizing the sum using the number of genes. Furthermore, the program needs to track the total score of the current state. This implies performing a non-synchronous reduction operation using an atomic operation.

It is better to split up the problem into several kernels such that data is not send back and forth between the host and the device. The kernels can then be scheduled efficiently on the GPU with the benefit of being able to run in different configurations. Synchronization could be implemented using an array which maps together chromosomes that can be used in the selection, crossover and mutation phase.

4 Performance

In this section the performance of the implementation of the pattern matching and genetic algorithm problem on the GPU is benchmarked. The power consumption of each algorithm is analyzed using equipment for measuring the power consumption of three different graphics cards. The cards span a range of low-cost power efficient devices (GT 240), a mid-ranged consumer card (GTX 560 Ti) and a high-end high performance card (GTX 480).

Performance is measured using a test suite, designed to vary the paramaters in both problems. In some cases, the choice of parameters also governs the number of blocks and threads and their configuration.

Power consumption (Watt) is measured using a PCI-express slot extender and a clamp meter for the supplementary power connectors. For verification, the Heaven-benchmark [4] by Unigine Corp was used to push the cards to maximum use and verify that the power consumption during execution reach near peak consumption.

4.1 Pattern matching

The pattern matching problem is fairly well-suited for the GPU. It features a high level of parallelism which can be exploited by the CUDA programming model using many threads. The magnitude of computational work performed should therefore be high.

In this problem the metrics of interest is GFLOP/s and the degree of utilization in GFLOP/s in comparison with the peak theoretical value. Furthermore, the power consumption is measured and the metric GFLOP/s per Watt is derived from these results.

4.1.1 Test suit

The test suit uses a number of parameters, some of which determine the launch configuration for the kernel. Table 2 shows the parameter values, used in the tests. T determines the number of blocks and the number of threads is determined by $N - K - 1$ i.e. the number of shifts.

Test	N	K	L	T	SCALE
1	128	32	16	5000	0
2	128	32	32	5000	0
3	256	32	32	5000	0
4	512	32	16	5000	0
5	512	32	32	5000	0
6	512	32	16	5000	1
7	512	32	16	5000	32
8	512	64	16	5000	0
9	512	128	16	5000	0
10	512	256	16	5000	0
11	512	256	1	5000	0
12	512	512	2	50000	0
13	256	256	1	50000	0
14	512	512	6	50000	32

Tabell 2: The test suit for the pattern matching kernel

N is the length of the test vector and K is the length of the template patterns. L is the number of templates to test and T is the number of test vectors to compute. SCALE is the number of scales to calculate for the best shift.

Time is measured using CUDA's own timing functions. Once the wall time is determined, the GFLOP/s is calculated as follows,

$$\text{GFLOP/s} = \frac{\times T \times L \times K \times (3 \cdot N + SCALE \cdot 4)}{\text{time}} \tag{2}$$

The peak GFLOP/s rate can be calculated a priori to act as an upper bound on the compute capacity. For the Fermi GPU this is calculated by multiplying the maximum number of operation per cycle with the number of cycles per second. The number of operations per second depends on the number of SMs and SP lanes in each SM. The GT 200 and GTX 480 have two lanes (32 ALUs in total) while the GTX 560 Ti is fitted with three SP lanes (48 ALUs in total). The peak GFLOP/s for each card is listed in the following table:

Device	Peak GFLOP/s
GT 240	280
GTX 560 Ti	1228.8
GTX 480	1345

Tabell 3: Peak GFLOP/s of the graphics cards

The peak GFLOP/s in Table 3 is an independent measure of problem, algorithm and implementation. It is a measurment of the computational capacity of the GPU as a computing unit.

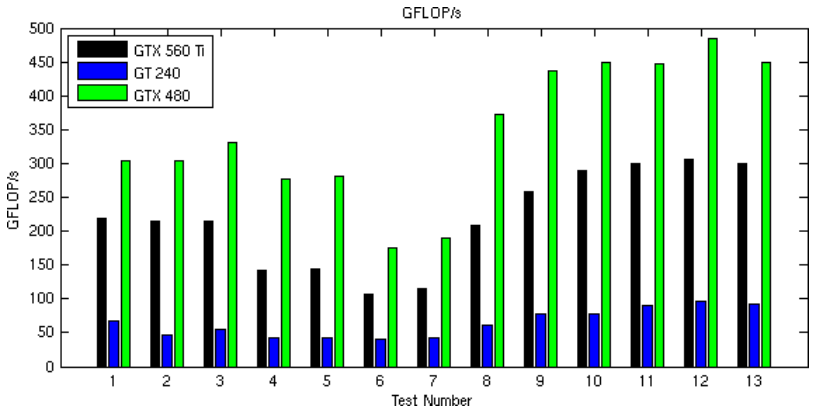
4.1.2 Performance

It is possible to calculate an upper limit on the GFLOP/s count for this particular problem, which makes it possilbe to accurately measure the performance of the code. Looking at the PTX assembler code it is observed that during the execution of the shifting and scaling operation the following code is executed in the loop body:

Listing 3: The PTX assembler result from the CUDA code

```
ld.shared.f32    %f27, [%rd15+12];
ld.shared.f32    %f28, [%rd4+12];
sub.f32          %f29, %f27, %f28;
mad.f32          %f30, %f29, %f29, %f26;
```

For each iteration the program must perform two load-instruction to fetch data from shared memory into registers, execute a FMA- and subtract-instruction, and then perform a store-instruction to save the result. This gives an approximate utilization of 3/4 of the maximum GFLOP/s count. The figure below shows the results achieved in GFLOP/s:



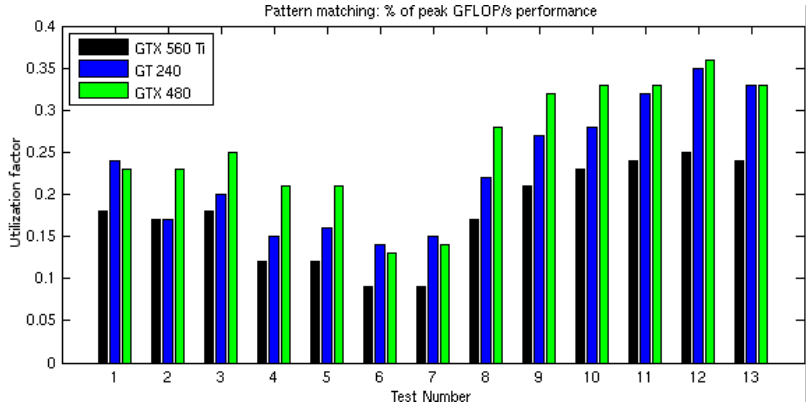
Figur 14: GFLOP/s count in the pattern matching problem showing the GTX 560 Ti (black), GT 240 (blue) and the GTX 480 (green).

The maximum and minimum GFLOP/s value is ~ 485 for test 12 and ~ 175 for test 6. The highest GLFOP/s is achieved when the number of threads is high and the amount of work per thread is high. The lowest value is achieved when there is a lot of threads and the amount of work per thread is low and there is warp divergence in the scaling-operation.

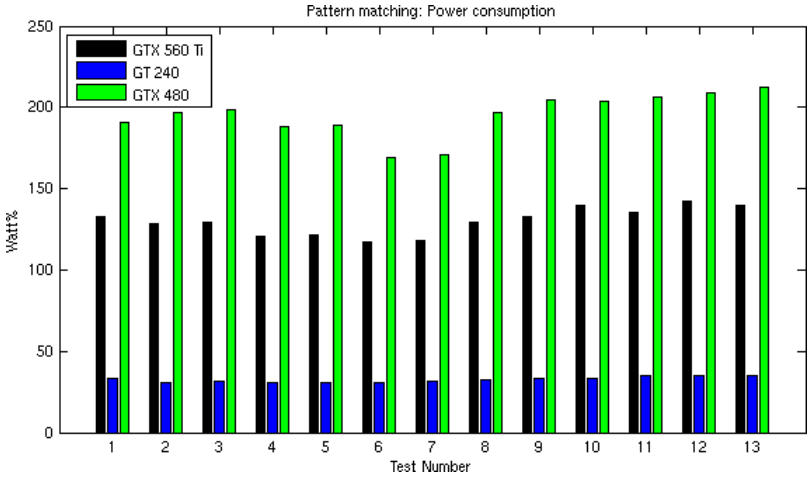
In comparison with the peak GFLOP/s performance the result in percentage is displayed in Figure 15.

4.1.3 Power consumption

The power consumption for the test suit is displayed in the graph below:



Figur 15: % of the peak GFLOP/s performance



Figur 16: The power consumption is considerably lower for the passively cooled GT 240 in comparison with the fan-cooled GTX 560 Ti and GTX 480.

4.2 Genetic algorithm

When implemented in parallel, the genetic algorithm poses many difficulties that are hard to solve in a good way for the GPU. Performance is then expected to be considerably lower for this problem than for the pattern matching problem.

The interesting metric is power consumption. GFLOP/s is not discussed because it is not straightforward how to calculate the GFLOP/s for this problem.

4.2.1 Test suit

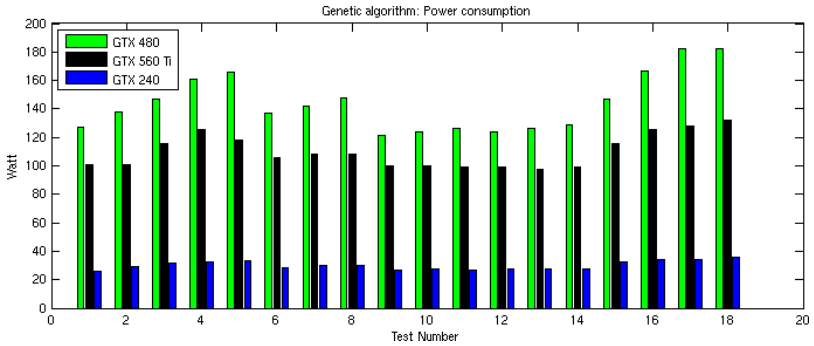
The test suit is constructed to test the program for different kinds of problem settings. Table 4 below shows the different launch configurations:

Test	Codes	GenesC	Chromo	PrMut	PrCros	MaxGen	MaxElite
1	4	8	50	0.6	0.01	500	500
2	4	50	50	0.6	0.01	500	500
3	4	100	50	0.6	0.01	500	500
4	4	200	50	0.6	0.01	500	500
5	4	500	50	0.6	0.01	500	500
6	4	8	100	0.6	0.01	500	500
7	4	8	150	0.6	0.01	500	500
8	4	8	200	0.6	0.01	500	500
9	8	8	50	0.6	0.01	500	500
10	20	8	50	0.6	0.01	500	500
11	40	8	50	0.6	0.01	500	500
12	100	8	50	0.6	0.01	500	500
13	150	8	50	0.6	0.01	500	500
14	200	8	50	0.6	0.01	500	500
15	8	100	50	0.6	0.01	500	500
16	8	100	100	0.6	0.01	500	500
17	8	100	150	0.6	0.01	500	500
18	8	100	200	0.6	0.01	500	500

Tabell 4: Genetic algorithm test suit.

4.2.2 Power consumption

The results from the power measurements are displayed in Figure 17.



Figur 17: The Power consumption in the genetic algorithm problem can vary greatly depending on the choice of parameters and is more sensitive to the size of data

5 Discussion

In this section, a summary of the results is presented and conclusions are drawn on basis of the figures in the previous section. First, we consider the degree of utilization in the pattern matching problem. Then, we discuss the genetic algorithm and discuss a possible improvement. The power consumption of different types of problems on the

Fermi GPU is discussed next and its performance in comparison with green supercomputer clusters is compared. Lastly, a reflection on the implications of writing efficient code presented and how this relates to the future GPU architectures.

5.1 Pattern matching and high-computational problems for GPUs

The pattern matching problem has a high computational complexity. A large portion of the runtime constitutes of arithmetic computation and logic comparison. This problem should therefore suit the GPU well.

As seen in Figure 14, the degree of utilization is satisfactory in relation to the peak theoretical GFLOP/s. For most well-posed problems the achieved efficiency should be in the interval of 70-80% utilization. However, the achieved efficiency is considerably lower for this probem. The pattern matching problem suffers from a lack of computational work per loaded element. The PTX assembly code in Listing 3 illustrates this problem. The table shows the computational section in each loop iteration, and it is clear that the section is bound by loading data from memory. The GTX 560 card suffers more than the GTX 480 since it has 48 SPs in comparison with 32. Efficiency is therefore higher on the GTX 480 because fewer SPs are idle.

5.2 Genetic algorithms and other diverse problems for GPUs

The nature of the genetic algorithm problem poses many complications for the GPU. In this work only selection, mutation and crossover is performed on the GPU and the rest of the algorithm is executed on the host.

In the pattern matching problem the power consumption is fairly regular for similar tests, while in the genetic algorithm the power consumption varies more. This could be explained by the large number of synchronization points in each iteration and between each iteration. Also, between iterations the fitness of each chromosome needs to be sent to the CPU which then determines the elite chromosome and transfers the next generation back to the GPU.

The implementation of the genetic algorithm is immature. A more refined solution was developed where the fitness could be globally evaluated on the graphics card. This involved complex synchronization between blocks using CUDA's set of atomic instructions. But getting blocks to synchronize without using an less efficient API call is a tricky task and the scheduling of blocks and threads can cause deadlocks in the application. This ultimately lead to falling back on the less mature solution which is far from optimal. An interesting experiment would be to incorporate spin-locks with exponential back-off which have been proven very efficient in [2]. Such techniques could benefit the implementation greatly since each block takes random time to complete.

5.3 Power consumption in high-performance computaion

An important observation in both problems is that neither achieve the specified peak consumption during the tests. During the verification proces the consumption of the GTX 480 and GTX 560 reached very close to the peak consumption value. This indicates that computing tasks on the Fermi GPU is not as power consuming as rendering tasks. Lower power consumption is very important for industry applications and in particular as an competitor to Field-programmable gate arrays (FGPAs). A trend in supercomputing clusters recent years has been to introduce GPUs to increase the cluster GFLOP/s per Watt performance. The GFLOP/s per Watt is alternative benchmark for green supercomputers and a trend in modern cluster construction. A list of the world's top green clusters can be found at [11]. Note, how in most cases the results from our implementation is close to or higher than the top listed super clusters.

Also, with Nvidia having set a goal of much higher double precision performance per Watt in the upcoming Kepler-architecture [5], they address the power consumption as a key architectural design goal for future GPUs.

5.4 Importance of programming efficiently

The trend in efficiency and power consumption indicates a linear relation between higher utilization and GFLOP/s per Watt. Our results support the argument that aiming for good utilization is a primary goal for implementations running on clusters of GPUs. Consequently, a problem that can be implemented to near maximum utilization is more efficient than others which are less efficient. This is important for GPU frameworks such as for example cuBLAS and cuSPARSE which is becoming more and more efficient with every release of the CUDA toolkit.

6 Conclusion

In high-perofmance applications GPUs are lately attracting much interest from the academia and industry. The shear computational power and low power consumption means GPUs are becoming more general than they were before the introduction of CUDA and OpenCL.

In this report, we study the Fermi GPU using two algorithms frequently appearing in high-performance and embedded applications; pattern matching and genetic algorithms. It is shown that the power consumption is lower for general computation tasks than in rendering tasks (such as video games) and that we achieve lower than specified peak consumption in all our tests. It is also shown that the efficiency of the implementation is proportional to the amounts of GFLOP/s per Watt which is an increasingly popular metric for cluster computers. We also show that the particular architecture of the graphics card has implications on the performance and the degree of utilization.

7 Acknowledgements

I am deeply in gratidute to Ian Wainwright, whose supervision have been inspirational and considerate.

I am also very grateful to the support and assistance from the course co-ordinartor Maya Neytcheva who has made the administration of this course very easy.

Lastly, I would like to mention Jimmy Pettersson who has contributed by discussing possible directions to take in the project and which areas to explore further.

10cm

Referenser

[1] Nvidia Fermi compute architecture whitepaper, page 4-5.

[2] Efi¬$cient Synchronization Primitives for GPUs, Stuart A. Jeff and Owens D. John, arxiv.org October 2011

[3] GPU computing in medical physics: A review. Medical Physics, Guillem Pratxa and Lei Xing, Vol. 38, No. 5, May 2011.

[4] http://unigine.com/products/heaven/

[5] http://www.anandtech.com/show/4572/kepler-gpus-shipping-this-year-nvidia-says-yes

[6] http://www.nvidia.com/object/product-geforce-gtx-560ti-us.html

[7] http://www.nvidia.com/object/product-geforce-gtx-560ti-us.html

[8] http://www.nvidia.com/object/product_geforce_gtx_480_us.html

[9] http://www.nvidia.com/object/product_geforce_gt_240_us.html

[10] http://www.pcisig.com/

[11] http://www.green500.org/

[12] http://www.ll.mit.edu/HPECchallenge/

[13] Radar Signal Processing with Graphics Processors (GPUs), Pettersson, Jimmy and Wainwright Ian, uu.diva-portal.org