Institutionen för datavetenskap
Department of Computer and Information Science

Final thesis

GPGPU-Sim

by

Filip Andersson

LIU-IDA/LITH-EX-G--14/085--SE

2014-11-20

Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings universitet
581 83 Linköping

Final Thesis

GPGPU-Sim

by

Filip Andersson

LIU-IDA/LITH-EX-G--14/085--SE

2014-11-20

Supervisor: Arian Maghazeh

Examiner: Unmesh Bordoloi

Abstract

This thesis studies the impact of hardware features of graphics cards on performance of GPU computing using GPGPU-Sim simulation software tool. GPU computing is a growing topic in the world of computing, and could be an important milestone for computers. Therefore, such a study that seeks to identify the performance bottlenecks of the program with respect to hardware parameters of the device can be considered an important step towards tuning devices for higher efficiency.

In this work, we selected convolution algorithm – a typical GPGPU application – and conducted several tests to study different performance parameters. These tests were performed on two simulated graphics cards (NVIDIA GTX480, NVIDIA Tesla C2050), which are supported by GPGPU-Sim. By changing the hardware parameters of a graphics card such as memory cache sizes, frequency and the number of cores, we can make a fine-grained analysis on the effect of these parameters on the performance of the program.

A graphics card working on a picture convolution task relies on the L1 cache but has the worst performance with a small shared memory. Using this simulator to run performance tests on a theoretical GPU architecture could lead to better GPU design for embedded systems.

Acknowledgements

I would like to use this opportunity to thank Unmesh Bordoloi and Arian Maghazeh for giving me this thesis, supporting me and giving me empathy during this work, due to my current health situation, namely my back and leg injury. Secondly, I want to express my gratitude to the Adam Larsson, without our close cooperation in our university years together, I might never have reached this point, this early in time. Thank you for your patience with me.

Lastly, I would like to dedicate this work to Karvin “Marco” Tarantino and Goldblossoms “Andy” “Pandy ” Shalimar, both family members who sadly passed away during my work. Marco's age had started catch up to him since he could no longer walk any long distances. It struck me with great grief when Marco died, despite a complicated relationship between us. In the end, I still viewed and loved him like family and miss him dearly.

Andy soon passed away as well, shortly after Marco was gone. He had been struck with liver cancer and we did not notice until it was too late. We though he, like us, was in sorrow because of the disappearance of Marco. Soon, he was also gone and I felt devastated, partly because I still had not finished mourning Marco, but also since we were closer. I have never experienced our home being so empty and it will never be quite the same again.

Linköping, November 2014

Filip Andersson

Table of contents

Chapter 1

Introduction

Normally the GPU (graphics processing unit) is used for graphics. However, this project delves into GPU computing, which is a new topic that has been getting more attention in the last decade. A GPU has more cores than a CPU (central processing unit) and is superior in handling parallelism and heavy workloads with a lot of data such as compute intensive algorithms. GPU computing utilizes the GPU, instead of the CPU, to process the compute intensive part of a program and sends the result back to the CPU. This has allowed some programs to reach up to 200 times the speed, compared to when only the CPU is used [1].

GPU computing is done through a programming language. Among the most well known languages is CUDA (Compute Unified Device Architecture), from NVIDIA which is a proprietary framework. An alternative is OpenCL which was influenced by CUDA and is the dominant general-purpose language [2]. Both are extensions of C and C++ and use similar syntax's [3][4].

The main focus of this thesis is to help others set up, understand the basics and do their own continuation experimentation on GPGPU-Sim. The goal is to identify the performance bottlenecks in an architecture and help design better devices such as embedded systems.

In Chapter 2, we will introduce the software, GPGPU-Sim. Chapter 3 focuses on the GPU architecture. Chapter 4 lists the cards and their statistics. Chapter 5 explains the steps to install GPGPU-Sim. Chapter 6 lists the parameters used in the configuration file for changing the components of a card. Chapter 7 explains the tests and what purpose they have and Chapter 8 contains the results. Chapter 9 concludes this paper.

Chapter 2

GPGPU-Sim

GPGPU-Sim was created by Tor M. Aamodt and his graduate students from The University of British Columbia. GPGPU-Sim is a simulation software tool of a contemporary GPU, as stated by the developers [8]. It is an accurate simulation, it was tested for its instructions per clock cycles (IPC) correlation versus the GT200 and Fermi architectures. This software is a per cycle simulator which means it reports results for every cycle of a program. A program, for example, has three cycles. If this program was run on GPGPU-Sim, it would have three different sections, one for each cycle. Among the sections are several parameters such as the amount of instructions executed, memory accesses, memory hits, memory misses and much more. According to the developers it resulted in 98.37% and 97.35% correlation, respectively [5]. This thesis used the latest version, GPGPU-Sim 3.2.2 as of now, for testing. This simulator has the potential to explore different kinds of architectures that might be limited by hardware or size limitations. To illustrate, a computer using this tool does not need the hardware in order to test it. Another advantage is that it is possible to monitor all events for each cycle.

GPGPU-Sim is used by changing parameters in a configuration file, which holds all statistics and settings for a graphics card. By changing the parameters, it is possible to add a L2 cache to a card that by default it does not exist in the device, change the size of the memories, change the frequencies of the processors and many more options. By changing the configuration file, it is possible to build your own architecture. It is also possible to run various tests to measure the efficiency of the cards. The CUDA SDK provides many different programs to run on GPGPU-Sim [6]. Tests can also be manually created to suit different needs. That makes it possible to see which card component has the most impact on performance during various tasks. With this knowledge, designers would know which component is the most crucial to a compute intensive task and make better decisions in order to maximize the cost efficiency. This is important for embedded systems, which are limited by space and have expensive components. In theory, this would allow developers to design and produce more efficient cards for each kind of workloads.

Chapter 3

GPU Architecture

GPGPU-Sim can currently model few actual GPU architectures. With time, more architectures will be available for experimentation. A GPU being modelled consists of Single Instruction Multiple Thread (SIMT) cores, connected via an on-chip connection network, and memory partitions that connect graphics GDDR DRAM [5]. A SIMT core is a multi-threaded pipelined Single Instruction Multiple Data (SIMD) processor. NVIDIA calls this a Streaming Multiprocessor (SM).

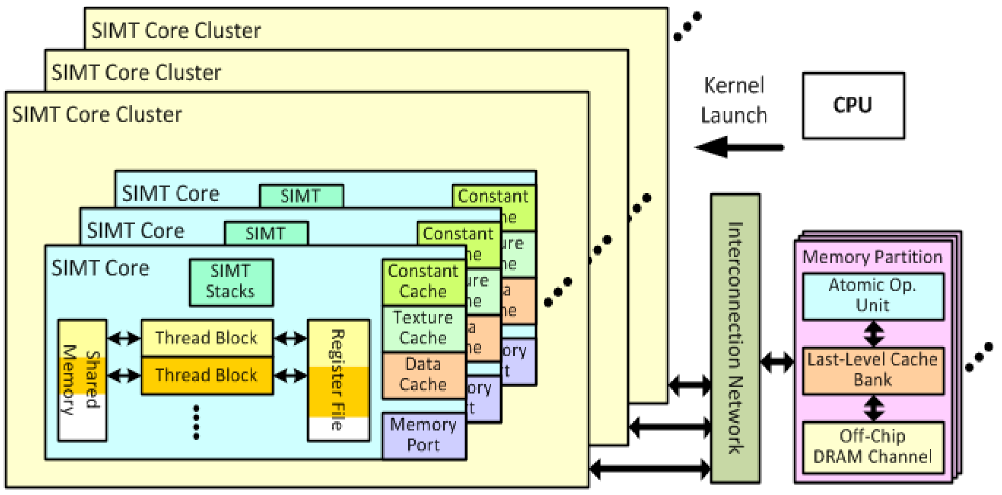

Figure 1-1. Overview of the architecture. Taken from GPGPU-Sim Manual [5]

As illustrated in figure 1-1, A single SIMT cores are grouped together in a SIMT Core Cluster. Every SIMT core Cluster has its own FIFO pipeline, which takes packets from the Interconnection Network.

In this architecture, several kinds of memories exist [5].

- Constant cache is used to store constants but it is also used to store parameters. It is a read only cache.
- Texture memory is used to cache textures but it is also possible to use the L2 cache (if available and enabled) to store textures. This is also a read only cache.
- In the data cache, private local memory and global memory is stored and accessed. For local memory, the L1 data cache acts as a write-back cache with write no-allocate. For global memory, write hits cause eviction of the block.
- Shared memory is handled by the programmer, and its size is changeable. It is used by its own set of threads. All of what are only accessible to its SIMT Core.

In addition to the constant cache, texture cache, data cache and shared memory, each SIMT Core also has an instruction cache and a LDST Unit which is shown in Figure 1-2. The LDST Unit serves as a memory pipeline. Each SIMT core clusters LDST unit is connected to an Injection Port Buffer which handles the Interconnection Network communication.

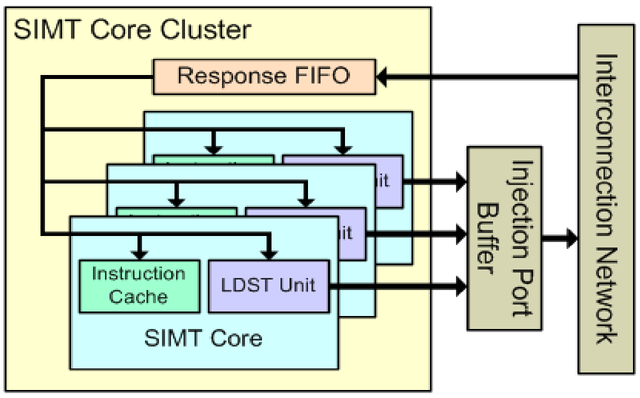

Figure 1-2. SIMT Core Cluster pipelines. Taken from GPGPU-Sim Manual [5]

Chapter 4

Card Specifications

This version of GPGPU-Sim has four different GPU architectures pre-packaged in the installation [7]. The following architectures are included GTX480, TeslaC2050, QuadroFX5600 and QuadroFX5800. In this study, only the GTX480 and TeslaC2050 are experimented on. In the following are the default components and statistics for each used architecture. Many of the numbers and components are changeable, for example, it is possible to change the L1 cache size or the clock frequency.

Tesla-C2050

- 14 Streaming Multiprocessors (SM)
- GPU clock: 575 MHz
- Shader clock: 1150 MHz
- Memory size: 3072 MB
- Shader units / CUDA cores: 448
- 64 KB L1 cache / Shared memory per 32 cores. 32 cores is one SM.
- 768 KB L2 cache
- Memory type: GDDR5

GTX480 Specifications

- 15 Streaming Multiprocessors (SM)
- GPU clock: 700 MHz
- Shader clock: 1401 MHz
- Memory size: 1536 MB
- Shader units / CUDA cores: 480
- 64 KB L1 cache / Shared memory per 32 cores.
- 768 KB L2 cache
- Memory type: GDDR5

Chapter 5

GPGPU-Sim Installation

The platform setting used in this case were:

- OS: Ubuntu 12.04 LTS 64-bit
- RAM: 3.8 GB
- CPU: Intel Core 2 CPU Q9550 2.83 GHz x 4 (Quad core)

Note that the hardware used will have an impact on the execution time of the simulation. Weaker hardware will result in the tests taking longer time to finish, which can be a problem.

Dependencies and Preparations

In this section, terminal commands will differ depending on the user and therefore, replace the obvious changeable parts with a suitable counterpart of your own. Installation should be possible on all Linux platforms if all dependencies are fulfilled [9]. In this case, it was done on the Ubuntu Long Term Support version 12.04. In that case, the following commands should be typed in the terminal.

GPGPU-Sim dependencies:

```
sudo apt-get install build-essential xutils-dev bison zlib1g-dev
flex libglu1-mesa-dev
```

GPGPU-Sim documents:

```
sudo apt-get install doxygen graphviz
```

AerialVision dependencies:

```
sudo apt-get install python-pmw python-ply python-numpy libpng12-
dev python-matplotlib
```

Cuda SDK dependencies:

```
sudo apt-get install libxi-dev libxmu-dev libglut3-dev
```

There is a problem with Ubuntu. Additional dependencies needs to be installed as libglut3-dev is outdated from Ubuntu 9.10 [10]. Add these dependencies:

```
sudo apt-get install mesa-common-dev freeglut3-dev freeglut3
```

The SDK needs to be installed manually, from NVIDIAs website [10]. This thesis used version 3.1 and is also recommended by the developers of GPGPU-Sim [9]. The SDK contains the most important tests, namely the convolutions. You need to separately get the SDK samples which contains the tests. To install the SDK, download a SDK corresponding to the operative system you are using. Then make the .run file executable, by using the following command:

```
sudo chmod -x filename_of_SDK.run
```

Then run it with:

```
sudo ./filename_of_SDK.run
```

It is necessary to use the sudo command, or create the folder ahead of the installation otherwise it will fail. The program will ask you where you want to install the SDK, the default location is `"/user/local/cuda"`. Change it to something else or leave it to default, and press enter. The SDK is installed, but set the install path of it to the correct place. To do that, run these commands. Change the `"/usr/local/cuda"` part to if you did not use the default path. The developers also suggest copying the following commands into your ".bashchr" file so it is not necessary to do this every time you wish to use the simulator. To find the ".bashchr" file, press CTRL+h in the file manager and scroll down to it.

```
export CUDA_INSTALL_PATH=/usr/local/cuda
export PATH=$PATH:$CUDA_INSTALL_PATH/bin
```

After that, the SDK samples needs to be installed in a similar way. Use the "chmod" and "./file.run" command previously used, and change its name to the corresponding file. It is important you install the SDK core before installing the samples as the samples depends on it. Finally, make sure the LD_LIBRARY_PATH is set correctly with this command:

```
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/cuda/lib64
```

Compiling the Tests

Now, navigate to the following folder in the terminal,

```
/home/yourusername/NVIDIA_GPU_Computing_SDK/C/src /foldername
```

There should be a lot of folders containing different tests, as stated before, this thesis used the convolutionSeparable and convolutionTexture tests. Enter a folder and type

```
make
```

That should compile the test and can be found in an other folder:

```
/home/yourusername/NVIDIA_GPU_Computing_SDK/C/bin/linux/release
```

For now, ignore this file and continue to install GPGPU-Sim.

GPGPU-Sim

To install the simulator, it must first be fetched from the developers site using a tool called git [8]. Make sure the program git is installed before installing GPGPU-Sim. Run the following command in the terminal to install git followed by the second command for installing the simulator:

```
sudo apt-get install git
git clone git://dev.ece.ubc.ca/gpgpu-sim
```

However, as of the 30th of September, the developers has changed the way of installing the program. It is now manually fetched in a zip file from github [7]. Fetch the file and extract it anywhere you want it. Then go into the terminal and navigate to the installation folder. Once inside, run these commands:

```
source setup_environment <current build>
```

I and the developers recommend using the ”release” build. After that, it should say it was successfully setup. Then run a simple command:

```
make
```

This will finally install the simulator and it is ready to be used. Optionally, you can install the doxygen documentation with:

```
make docs
```

Now should everything be installed and ready for use. In case of further studying the source code of the simulator, the files should be located in the folder “`/installation_path/src/gpgpu-sim/`”.

Navigate to the folder where you installed it, further into the “configs” folder. There are some folders with different graphics cards names and copy the folder that you desire to use for testing. Place the copy preferably in a place where it is easy to access, for example in your “HOME” folder. For simplicities sake, we call it the *working directory*. In this folder, there are some configuration files for the graphics card you want to simulate on. The most important file is “gpgpusim.config” which is the core file for the graphics card you simulate. Everything from frequency to memory controllers are configurable from that file. Enter the working directory with the terminal and type the following:

```
source                  /home/your-username/GPGPU-Sim-installation-
path/setup_environment name-of-the-folder-you-copied
```

An example of this would be:

```
source /home/filan750/gpgpu-sim/v3.x/setup_environment GTX480
```

This will setup the simulator to run tests on a GTX480 card. Note that this command has to be typed every time you open a new terminal, before you continue with the tests. Now, copy the .run file from the folder:

```
/home/your-username/NVIDIA_GPU_Computing_SDK/C/bin/linux/release
```

Past it in your *working directory*, containing the configuration files. GPGPU-Sim is now finally ready to run a simulation test, on a graphics card of your choice. Type this command in the terminal:

```
./name-of-the-program
```

To get a .txt file with all the output from the program and see the output in real time, use the following command instead:

```
script -c ./nameofprogram yourfile.txt
```

Chapter 6

Configuration File

As stated before, the most important file is the “gpgpusim.config” file, which contains all parameters for the graphics card. Many, but not all, parameters in the configuration file have an explanation on the projects Wikipedia manual [5]. The most important parameters that are used in our will be listed along with a brief explanation.

Table 6-1. Parameters for the configuration file

-gpgpu_ptx_sim_mode	It runs the simulator but does not print any statistics. It is useful for quick testing if a CUDA program gets a deadlock or errors.
-gpgpu_clock_domains <Core Clock>:<Interconnect Clock>:<L2 Clock>:<DRAM Clock>	In Fermi, each pipeline has 16 execution units, so the Core clock needs to be divided by 2. (GPGPU-Sim simulates a warp (32 threads) in a single cycle). 1400/2 = 700
-gpgpu_num_sched_per_core	Number of warp schedulers per core
-gpgpu_n_clusters	Number of processing clusters
-gpgpu_n_cores_per_cluster	Number of SIMD cores per cluster
-gpgpu_perfect_mem <0=off (default), 1=on>	This makes the memory always have a hit on a memory access.
-gpgpu_tex_cache:l1 <nsets>:<bsize>:<assoc>:<rep>:<wr>:<alloc>,<mshr>:<N>:<merge>,<mq>	Texture cache (Read-Only) configuration. Evict policy: L = LRU, F = FIFO, R = Random. The nset x bsize x assoc = memory size.
-gpgpu_const_cache:l1 <nsets>:<bsize>:<assoc>:<rep>:<wr>:<alloc>,<mshr>:<N>:<merge>,<mq>	Constant cache (Read-Only) configuration. Evict policy: L = LRU, F = FIFO, R = Random
-gpgpu_cache:il1 <nsets>:<bsize>:<assoc>:<rep>:<wr>:<alloc>,<mshr>:<N>:<merge>,<mq>	Shader L1 instruction cache (for global and local memory) configuration. Evict policy: L = LRU, F = FIFO, R = Random
-gpgpu_cache:dl1 <nsets>:<bsize>:<assoc>:<rep>:<wr>:<alloc>,<mshr>:<N>:<merge>,<mq> -- set to "none" for no DL1 --	L1 data cache (for global and local memory) configuration. Evict policy: L = LRU, F = FIFO, R = Random
-gpgpu_cache:dl2 <nsets>:<bsize>:<assoc>:<rep>:<wr>:<alloc>,<mshr>:<N>:<merge>,<mq>	Unified banked L2 data cache configuration. This specifies the configuration for the L2 cache bank in one of the memory partitions. The total L2 cache capacity = <nsets> x <bsize> x <assoc> x <# memory controller>.
-gpgpu_shmem_size	Size of shared memory per SIMT core (aka shader core) (default=16kB)
-gpgpu_cache:dl2_texture_only	L2 cache is used texture only (default=0)

Chapter 7

Tests

The tests simulated on GPGPU-Sim were four in total:

- “Hello World”
- ConvolutionTexture
- ConvolutionSeparable-V1
- ConvolutionSeparable-V2

The “Hello World!” was a simple program that only printed the phrase and was mostly used for initial testing. “Hello World” was run on both GTX480 and TeslaC2050, outputs were then compared against each other. Due to the small scale of the “Hello World” program, its comparison will neither be shown, nor evaluated.

The two convolution tests, ConvolutionTexture and ConvolutionSeparable, were provided by the CUDA SDK 3.1 [6]. ConvolutionTexture was also used on both TeslaC2050 and GTX480 then evaluated. The first point investigated was how accurately the simulator could simulate two different cards. This is important, to make sure the theory matches with the reality. The card with more resources should perform better than the other. ConvolutionTexture simulates a picture 1536 pixels high and 3072 pixels wide, iterated 10 times which explains the high amount of cycles and instructions.

ConvolutionSeparable was used on the GTX480 for viewing the impact of the L1 cache and shared memory. ConvolutionTexture would have been used but it does not need to use the shared memory, and therefore would not yield any results, and for that reason ConvolutionSeparable was used instead. The ConvolutionSeparable test that came with the CUDA SDK 3.1 was slightly modified. The reason for this, it simulated a picture convolution and iterated 16 times on a 4096 pixel picture which resulted in a long simulation time, over 24 hours. Due to this time consuming task, the scale of the test was limited to a more reasonable scale for running the test many times. The ConvolutionSeparable-V1 is configured with just two iterations, not counting the warm up iteration, and a 1024*1024 pixel picture.

Another variation, ConvolutionSeparable-V2, was configured with three iterations, not counting the warm up round again, and 2048*2048 pixel picture. This was to make sure the patterns are correct with a second, larger scale test.

- Table 8-1 lists the results from ConvolutionTexture
- Table 8-2 lists the results from ConvolutionSeparable-V1
- Table 8-3 lists the remaining results from ConvolutionSeparable-V1
- Table 8-4 lists the results from ConvolutionSeparable-V2
- Table 8-5 lists the remaining results from ConvolutionSeparable-V2

ConvolutionSeparable were only run on the GTX480 architecture. To change the ConvolutionSeparable test, the main-file for it needs editing. The file can be found in:

```
/home/*username*/NVIDIA_GPU_Computing_SDK/C/src/convolutionSeparable
```

Find and change the parameters:

```
const int imageW
const int imageH
const int iterations
```

Then, build the tests and move them to your workplace.

Chapter 8

Results

This section will list the most important result parameters with an explanation. A comparison between the GTX480 and TeslaC2050, as well the convolution tests that were run on the GTX480. Unfortunately, there is currently a parameter that is glitched, "`gpgpu_stall_shd_mem[gl_mem][coal_stall]`" [11][12]. It does not report what it is supposed to report and therefore, renders some results worthless and at best, doubtable. The developers know this and will hopefully fix this in time.

GTX480 vs TeslaC2050

Table 8-1. ConvolutionTexture results.

Parameter	**Explanation**	**TeslaC2050 result**	**GTX480 result**
gpu_sim_cycle	Current iteration's cycles	1077403	1006505
gpu_sim_insn	Current iteration's instructions	877658112	877658112
gpu_ipc	Avg. instructions per cycle	814.6052	871.9858
gpu_tot_sim_cycle	Total number of cycles (in Core clock) simulated for all the kernels launched so far	21107073	19714489
gpu_tot_sim_insn	Total number of instructions executed for all the kernels launched so far	17553162240	17553162240
gpu_tot_ipc	Total avg. instructions per cycles	831.6247	890.3687

The gpu_sim_cycle parameter is the amount of cycles for one iteration. In the test there is usually a loop with the main workload, the algorithm, and each loop finished is called an iteration. The gpu_sim_insn, is similar but is counting the amount of instruction in the loop for the iteration. The result of gpu_sim_cycle for Tesla C2050 shows that it took 1077403 cycles to finish, for one iteraton. The gpu_ipc parameter is the gpu_sim_insn divided by the gpu_sim_cycle which translates into a ratio called instructions per cycle. This ratio is used to measure performance. The last parameters with a "tot" in the name, such as the gpu_tot_sim_insn, shows the amount of instructions needed to execute for all iterations in the program.

The parameter gpu_ipc results, 814.6052 and 871.9858, show that the GTX480 card has more instructions per cycles, due to superior hardware. This means a higher output which is positive for performance results. It is also important to measure other statistics such as the hit and miss rates for the different caches. The test was run on the default GTX480 settings, therefore, it would have other results with its other shared memory and L1 cache settings. Also, since the stalled parameters are not entirely reliable other parameters could be affected as well, which might be an explanation for results in the other test.

GTX480 ConvolutionSeparable-V1

As previously stated, the caches and shared memory has an impact on the performance. In the next test, the sizes of both are changed to measure the effects on the card. The experiment used fixed scaling for simplicities sake. With that condition, we are able to see which component has the most impact on the graphics card. The GTX480 has a default 16 kB shared memory and a L1 cache with a size of 48 kB. A built in alternative is to switch the two, to 48 kB shared memory and 16 kB L1 cache. The ConvolutionSeparable test takes further study how the restrictions and availability influence the results, both theoretical architecture mixes along with the default ones. In the following evaluation, the notations are like this, 32:32, which means 32 kB L1 cache:32 kB shared memory.

Table 8-2. Data table for ConvolutionSeparable-V1

Iteration	L1 size (KB)	Shd mem. size (KB)	Nr. Cycles	Nr. Instructions	Frequency (MHz)	Cycles / Frequency (us)
1	16	16	212043	62914560	700	302.92
2	16	16	205212	62914560	700	293.16
Total:			417255	125829120	700	596.08
1	32	16	215775	62914560	700	308.25
2	32	16	208286	62914560	700	297.55
Total			424061	125829120	700	605.80
1	48	16	225685	62914560	700	322.41
2	48	16	212326	62914560	700	303.32
Total			438011	125829120	700	625.73
1	16	32	171701	62914560	700	245.29
2	16	32	164412	62914560	700	234.87
Total			336113	125829120	700	480.16
1	32	32	166791	62914560	700	238.27
2	32	32	160917	62914560	700	229.88
Total			327708	125829120	700	468.15
1	48	32	166751	62914560	700	238.22
2	48	32	160572	62914560	700	229.39
Total			327323	125829120	700	467.60
1	16	48	171701	62914560	700	245.29
2	16	48	164412	62914560	700	234.87
Total			336113	125829120	700	480.16
1	32	48	166791	62914560	700	238.27
2	32	48	160917	62914560	700	229.39
Total			327708	125829120	700	468.15
1	48	48	166751	62914560	700	238.22
2	48	48	160572	62914560	700	229.39
Total			327323	125829120	700	467.60

In Table 8-2, all the configurations have the same amount of instructions and frequency to make sure the workload is equal across them. From the results in the cycles per frequency column, the worst performance is the 48:16 with the lowest time of 322.41 µs, 303.32 µs and 625.73 µs in the cycles per frequency column. The best performance is a tie with both the 48:32 and 48:48 setting having the same time of 238.27 and 229.39. The last result is different with 468.15 µs and 467.60 µs meaning that the 48:48 setting is technically faster but is not efficient. For a 0.55 µs difference, 16 extra kB memory did not add much benefit. The 48:48 has almost the same results as the 48:32, however, the purpose of running the ConvolutionSeparable test was to find out which configuration has the better performance based on the hardware. Since the 48:48 has more resources but cannot utilize it efficiently which is could be an economical waste, if this set-up has workloads consisting of convolution or similar tasks. However, speed could be a requirement for a system, and is common, so the 48:48 should not be completely overlooked in that case.

Table 8-3. Data table for ConvolutionSeparable-V1's L1 cache hits and misses

Iteration	L1 size (KB)	Shd mem. size (KB)	Inst. Hits	Inst. Misses	Data hits	Data Misses	Const. Hits	Const. Misses
1	16	16	1113153	118337	145408	106586	573440	466
2	16	16	1114331	119515	145048	104731	573440	-
Total:			2227484	237852	290816	211317	1146880	466
1	32	16	1115246	120430	145408	107503	573440	480
2	32	16	1114674	119858	145048	107352	573440	-
Total			2229920	240288	290816	214855	1146880	480
1	48	16	1115414	120598	145408	109009	573440	480
2	48	16	1114817	100001	145048	107358	573440	-
Total			2230231	220599	290816	216367	1146880	480
1	16	32	1097879	100063	145408	107259	573440	612
2	16	32	1093117	98301	145048	106857	573440	-
Total			2190966	198364	290816	214116	1146880	612
1	32	32	1095456	100640	145408	106125	573440	607
2	32	32	1094048	99232	145048	105446	573440	-
Total			2189504	199872	290816	211571	1146880	607
1	48	32	1095202	100386	145408	106192	573440	607
2	48	32	1092914	98098	145048	105256	573440	-
Total			2188116	198484	290816	211448	1146880	607
1	16	48	1094879	100063	145408	107259	573440	612
2	16	48	1093117	98301	145048	106857	573440	-
Total			2187996	198364	290816	214116	1146880	612
1	32	48	1095456	100640	145408	106125	573440	607
2	32	48	1094048	99232	145048	105446	573440	-
Total			2189504	199872	290816	211571	1146880	607
1	48	48	1095202	100386	145408	106192	573440	607
2	48	48	1092914	98098	145048	105256	573440	-
Total			2188116	198484	290816	211448	1146880	607

In Table 8-3, there is no column for the data texture cache hits and misses, as the test did not use the texture cache. The columns “Inst. Hits” and “Inst. Misses” shows the amount of instruction misses decreases with the instruction hits. A very likely possibility is that since a smaller L1 cache has to clear and fetch instructions more often, resulting in a higher number. This could also explain the data misses, and why the 16:16 set-up has the least amount of misses. Data hits and Constant hits should be the same for all the set-ups, otherwise it is most likely not the same convolution simulation.

The constant cache misses columns does miss some data in the second iterations. This is due to how the C++ data fetching program works, which was used for quickly gathering results from the simulation output file. The C++ program subtracts the first run results from the total result and presents that as the second iteration result. It is unknown why it does not appear to change on the second iteration. One theory is that the parameter is glitched. Another theory could be that the constant cache only need to store the parameters the simulator need in the beginning of the simulation, which would be a more understandable reason for the results in the “Const. Miss” column.

The column for data cache misses, shown in the “Data Misses” column, shows the 48:16 once again has the worst performance with a total of 216367 misses. The ones with the least amount of misses are 16:48 and 16:32 with a total of 214116 misses. The conclusion from the data cache results and the previous results instruction cache results, a pattern exists. The less amount of shared memory, the worse performance. Although the L1 cache also has an impact, it is not as important as the shared memory. The conclusion is to prioritize the shared memory over L1 if no other options are available when designing a system that works with convolution or a similar task.

Lastly, a second version of ConvolutionSeparable was created and tested. As previously stated, the test was only to confirm consistency for the results. The test gave similar results, with the 32:32 setting being the fastest at 899.87 µs, 892.03 µs. 892.67 µs and 2684.57 µs in the cycles per frequency column from the Table 8-4.

GTX480 ConvolutionSeparable-V2

Table 8-4. Data table for ConvolutionSeparable-V2

Iteration	L1 size (KB)	Shd mem (KB)	Nr. Cycles	Nr. Instructions	Frequency (MHz)	Cycles/Frequenzy (us)
1	16	16	805009	251658240	700	1150.01
2	16	16	795841	251658240	700	1136.92
3	16	16	795841	251658240	700	1136.92
Total			2396691	754974720	700	3423.84
1	32	16	802624	251658240	700	1146.61
2	32	16	790225	251658240	700	1128.89
3	32	16	788142	251658240	700	1125.92
Total			2380991	754974720	700	3401.42
1	48	16	810275	251658240	700	1157.54
2	48	16	794337	251658240	700	1134.77
3	48	16	831380	251658240	700	1187.69
Total			2435992	754974720	700	3479.99
1	16	32	634326	251658240	700	906.18
2	16	32	628465	251658240	700	897.81
3	16	32	629495	251658240	700	899.28
Total			1892286	754974720	700	2703.27
1	32	32	629906	251658240	700	899.87
2	32	32	624420	251658240	700	892.03
3	32	32	624872	251658240	700	892.67
Total			1879198	754974720	700	2684.57
1	48	32	639374	251658240	700	913.39
2	48	32	630117	251658240	700	900.17
3	48	32	623674	251658240	700	890.96
Total			1893165	754974720	700	2704.52
1	16	48	634326	251658240	700	906.18
2	16	48	628465	251658240	700	897.81
3	16	48	629495	251658240	700	899.28
Total			1892286	754974720	700	2703.27
1	32	48	629906	251658240	700	899.86
2	32	48	624420	251658240	700	892.03
3	32	48	624872	251658240	700	892.67
Total			1879198	754974720	700	2684.57
1	48	48	639374	251658240	700	913.39
2	48	48	630117	251658240	700	900.17
3	48	48	623674	251658240	700	890.96
Total			1893165	754974720	700	2704.52

Table 8-5. Data table for ConvolutionSeparable-V2's L1 cache hits and misses

Iteration	L1 size (KB)	Shd mem. Size (KB)	Inst. Hits	Inst. Misses	Data Hits	Data Misses	Const. Hits	Const Misses
1	16	16	4459363	479075	585728	416900	2293760	441
2	16	16	4459619	331479	585728	414150	2293760	-
3	16	16	4460804	480516	585728	412458	2293760	-
Total			13379786	1438922	1757184	1243508	6881280	441
1	32	16	4461745	481457	585728	416447	2293760	441
2	32	16	4463129	482841	585728	412051	2293760	-
3	32	16	4463785	483497	585728	410763	2293760	-
Total			13388659	1447795	1757184	1239261	6881280	441
1	48	16	4461685	481397	585728	421946	2293760	496
2	48	16	4462058	481770	585728	416229	2293760	-
3	48	16	4460752	480464	585728	430576	2293760	-
Total			13384495	1443631	1757184	1268751	6881280	496
1	16	32	4378087	397799	585728	422484	2293760	538
2	16	32	4371965	391677	585728	422002	2293760	-
3	16	32	4376128	395840	585728	423045	2293760	-
Total			13126180	1185316	1757184	1267531	6881280	538
1	32	32	4378295	398007	585728	420703	2293760	538
2	32	32	4379711	399423	585728	420491	2293760	-
3	32	32	4376854	396566	585728	420481	2293760	-
Total			13134860	1193996	1757184	1261675	6881280	538
1	48	32	4385140	404852	585728	425251	2293760	615
2	48	32	4381151	400863	585728	423078	2293760	-
3	48	32	4378922	398634	585728	421743	2293760	-
Total			13145213	1204349	1757184	1270072	6881280	615
1	16	48	4378087	397799	585728	422484	2293760	538
2	16	48	4371965	391677	585728	422002	2293760	-
3	16	48	4376128	395840	585728	423045	2293760	-
Total			13126180	1185316	1757184	1267531	6881280	538
1	32	48	4378295	398007	585728	420703	2293760	538
2	32	48	4379711	399423	585728	420491	2293760	-
3	32	48	4376854	396566	585728	420481	2293760	-
Total			13134860	1193996	1757184	1261675	6881280	538
1	48	48	4385140	404852	585728	425251	2293760	615
2	48	48	4381151	400863	585728	423078	2293760	-
3	48	48	4378922	398634	585728	421743	2293760	-
Total			13145213	1204349	1757184	1270072	6881280	615

Chapter 9

Conclusion

This thesis has presented GPGPU-Sim, a simulator to exercise GPU computing on a simulated card. It shows how to install, use GPGPU-Sim for future projects. Finally, we presented a portion of the results from an example. In the experiments two different cards has been simulated and compared against each other for an accuracy test of the simulator. Two other tests have compared the importance of having a L1 cache and shared memory available for convolution tasks. In the second convolution test similar patterns, like the ones in the first one, are present.

Future works on GPGPU-Sim could be a similar study but with a focus on power usage. Green IT and power conservation are important issues nowadays. This study could also be beneficial for embedded systems. There is a program called GPUWattch integrated in GPGPU-Sim in version 3.20 and later that could be further studied [13].

Another possibility would be to use the results to help develop better compilers that take advantage of the hardware characteristics of the GPU into consideration and produce more efficient executables.

It is also possible to use the experience and knowledge gained in this work to build a similar tool for embedded GPUs based on GPGPU-Sim. GPGPU-Sim is still use-able, but not optimal for smaller architectures.

Chapter 10

Sources

1. Ali Bakhoda, George Yuan, Wilson W. L. Fung, Henry Wong, Tor M. Aamodt, Analyzing CUDA Workloads Using a Detailed GPU Simulator, in IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Boston, MA, April 19-21, 2009.

2."General-purpose Computing on Graphics Processing Units." Wikipedia. Wikimedia Foundation. [www] <http://en.wikipedia.org/wiki/General-purpose_computing_on_graphics_processing_units>. Accessed 29 Sept. 2014.

3. "CUDA." Wikipedia. Wikimedia Foundation. [www] <http://en.wikipedia.org/wiki/CUDA>. Accessed 10 Oct. 2014.

4. "OpenCL." Wikipedia. Wikimedia Foundation. [www] <http://en.wikipedia.org/wiki/OpenCL>. Accessed 10 Oct. 2014.

5. Aadmodt, Tor M., and Wilson W.L. Fung. "GPGPU-Sim 3.x Manual." GPGPU-Sim. The University of British Columbia. [www] <http://gpgpu-sim.org/manual/index.php/GPGPU-Sim_3.x_Manual>. Accessed 25 Oct. 2014.

6. "CUDA Toolkit 3.1 Downloads." CUDA Toolkit 3.1 Downloads. NVIDIA. [www] <https://developer.nvidia.com/cuda-toolkit-31-downloads>. Accessed 21 May 2014.

7. "Gpgpu-sim/gpgpu-sim_distribution." GitHub. [www] <https://github.com/gpgpu-sim/gpgpu-sim_distribution>. Accessed 30 Sept. 2014.

8. "GPGPU-Sim." GPGPU-Sim. The University of British Columbia. [www] <http://www.gpgpu-sim.org/>. Accessed 25 Oct. 2014.

9. "Root/v3.x/README." /v3.x/README – GPGPU-Sim. N.p. [www] <https://dev.ece.ubc.ca/projects/gpgpu-sim/browser/v3.x/README>. Accessed 23 Sept. 2014.

10. "GPU Computing SDK." NVIDIA. [www] <http://developer.download.nvidia.com/compute/cuda/3_1/sdk/gpucomputingsdk_3.1_linux.run>. Accessed 21 May 2014.

11. Minseok, Lee, and Wilson W.L. Fung. "GPGPU-Sim Google Group." Google Groups. Google. [www] <https://groups.google.com/forum/#!searchin/gpgpu-sim/gpgpu_stall_shd_mem/gpgpu-sim/8L1tuwudnYw/HRVh17mQf18J>. Accessed 18 Sept. 2014.

12. Fung, Wilson W.L. "GPGPU-Sim Bug Tracking System." Bugzilla. N.p. [www] <http://www.gpgpu-sim.org/bugs/show_bug.cgi?id=25>. Accessed 18 Sept. 2014.

13. "GPU Wattch Manual." [www] <http://www.gpgpu-sim.org/gpuwattch/>. Accessed 18 Nov. 2014.

Appendix

C++ program used to fetch the parameters and present them, for a three iteration run in a simulation.

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
#include <iterator>

using namespace std;

void fetcher(string& data,vector<string>& res);
void get_value_and_store(vector<string>& res, vector<int>& summary);
void print(const vector<int> & summary);
void print_hit(const vector<int>& v, int n );
void print_miss(const vector<int>& v, int n );

int main()
{
  vector<string> res;
  vector<int> summary;
  string data;

  // fetches and stores all parameters we are interested in
  fetcher(data,res);

  // optional extra print
  //copy(res.begin(),res.end(),ostream_iterator<int>(cout, "\n"));

  /*
   * Since the results are stored like following example:
   * L2_total_cache_accesses = 117233
   * we must find the position of '=' and then we can extract
   * the number behind it, by subtracting a part of the string from
   * the position to the end of the string. There we got the number
   * and push it into the vector for storage and later use
   * both misses and hits are handled.
   */
  get_value_and_store(res,summary);

  // even more debug outprint
  //copy(summary.begin(),summary.end(),ostream_iterator<int>(cout, "\n"));

  print(summary); // giant print function

  return 0;
}

void fetcher(string& data,vector<string>& res)
{
  while(getline(cin,data))
    {
      if(data.find("L1I_total_cache_accesses") != std::string::npos)
      res.push_back(data);

      if(data.find("L1I_total_cache_misses") != std::string::npos)
      res.push_back(data);
      if(data.find("L1D_total_cache_accesses") != std::string::npos)
      res.push_back(data);

      if(data.find("L1D_total_cache_misses") != std::string::npos)
      res.push_back(data);

      if(data.find("L1C_total_cache_accesses") != std::string::npos)
      res.push_back(data);

      if(data.find("L1C_total_cache_misses") != std::string::npos)
      res.push_back(data);

      if(data.find("L1T_total_cache_accesses") != std::string::npos)
      res.push_back(data);

      if(data.find("L1T_total_cache_misses") != std::string::npos)
      res.push_back(data);

      if(data.find("L2_total_cache_accesses") != std::string::npos)
      res.push_back(data);

      if(data.find("L2_total_cache_misses") != std::string::npos)
      res.push_back(data);
    }
}

void get_value_and_store(vector<string>& res, vector<int>& summary)
{
  int iter1, iter2, iter3, eq;

  for(int i = 0; i < 10; i++)
    {
      eq = res.at(i).find("="); //first iteration results (both hits & misses)
      string d;
      d = res.at(i).substr(eq+2, res.at(i).length());
      iter1 = atoi(d.c_str());
      summary.push_back(iter1);

      eq = res.at(i+10).find("="); //second iteration results
      string d2;
      d2 = res.at(i+10).substr(eq+2, res.at(i+10).length());
      iter2 = atoi(d2.c_str());
      summary.push_back(iter2);

      eq = res.at(i+20).find("="); //third iteration results
      string d3;

      d3 = res.at(i+20).substr(eq+2, res.at(i+20).length());
      iter3 = atoi(d3.c_str());
      summary.push_back(iter3);
    }
}

void print(const vector<int>& summary)
{
  cout << "Now printing the results, in order, Inst, Data, Const, Text, L2:\n";
  cout << "L1 Instruction cache results: " << endl;
  print_hit(summary, 0);
  print_miss(summary, 3);

  cout << "L1 Data cache results:" << endl;
  print_hit(summary, 6);
  print_miss(summary, 9);

  cout << "L1 Constant cache results:" << endl;
  print_hit(summary, 12);
  print_miss(summary, 15);

  cout << "L1 texture cache results: " << endl << endl;
  print_hit(summary, 18);
  print_miss(summary, 21);

  cout << "L2cache results:" << endl;
  print_hit(summary, 24);
  print_miss(summary, 27);
}

void print_hit(const vector<int>& v, int n )
{
  cout << "iteration 1 hits: " << v.at(n) << endl;
  cout << "iteration 2 hits: " << v.at(n+1)-v.at(n) << endl;
  cout << "iteration 3 hits: " << v.at(n+2)-v.at(n+1) << endl;
  cout << "iteration total hits: " << v.at(n+2) << endl;
}

void print_miss(const vector<int>& v, int n )
{
  cout << "iteration 1 misses: " << v.at(n) << endl;
  cout << "iteration 2 misses: " << v.at(n+1)-v.at(n) << endl;
  cout << "iteration 3 misses: " << v.at(n+2)-v.at(n+1) << endl;
  cout << "iteration total misses: " << v.at(n+2) << endl;
}
```