GDAŃSK UNIVERSITY OF TECHNOLOGY
FACULTY OF ELECTRONICS, TELECOMMUNICATIONS AND INFORMATICS

Tomasz Gajger

Modeling parallel processing with GPU and verification using the MERPSYS platform

Modelowanie przetwarzania równoległego z wykorzystaniem GPU wraz z weryfikacją z wykorzystaniem platformy MERPSYS

Master's thesis
written under the supervision of
dr hab. inż. Paweł Czarnul

Gdańsk 2017

STRESZCZENIE

W niniejszej pracy dokonano oceny modelu wydajnościowego dla GPU, który bazując na prawie Little'a w analityczny sposób wyraża czas wykonania kernela jako zależny od ograniczeń wynikających z opóźnień i przepustowości, oraz osiągniętej zajętości zasobów. Następnie model ten połączono z rezultatami kilku innych prac naukowych z tejże dziedziny oraz dodano równania szacujące czas przesyłania danych, aby dalej włączyć go jako element do platformy MERPSYS, która jest symulatorem ogólnego przeznaczenia dla systemów równoległych i rozproszonych. Opracowane rozwiązanie pozwala użytkownikowi na utworzenie opisu aplikacji CUDA w edytorze MERPSYS z wykorzystaniem rozszerzonego języka Java, a następnie przeprowadzenie w wygodny sposób oceny wydajności dla różnorodnych konfiguracji uruchomieniowych z wykorzystaniem różnych urządzeń. Możliwe jest również przekroczenie rzeczywistych ograniczeń nakładanych przez sprzęt, przykładowo ilości pamięci dostępnej na hoście bądź urządzeniu, czy liczby połączonych ze sobą kart graficznych. Dodatkowo opisanych zostało wiele aspektów związanych z przetwarzaniem na GPU, takich jak architektura i model programistyczny CUDA, hierarchia pamięci, przebieg wykonania kernela, wytwarzanie i analiza aplikacji CUDA. Przedstawiono również systematyczną metodologię pozwalającą na wyznaczenie charakterystyk kernela, a następnie wykorzystanie ich jako parametry wejściowe modelu. Cały proces implementacyjny poczynając od napisania prostego kernela CUDA, jego kompilacji i dekompilacji, analizy kodu maszynowego, modyfikacji platformy MERPSYS i zdefiniowania w jej ramach modelu, został drobiazgowo opisany. Model został oceniony z wykorzystaniem kerneli wykazujących różne cechy i dla wielu konfiguracji uruchomieniowych. Autor ocenia go jako bardzo dokładny dla kerneli ograniczonych przez przepustowość obliczeniową i dla realistycznych zadań obliczeniowych. Natomiast dla kerneli ograniczonych przez przepustowość pamięci i zdegenerowanych scenariuszy wyniki wciąż mieściły się w zadowalających ramach. Dodatkowo dowiedziono przenośność modelu pomiędzy dwoma urządzeniami z tej samej architektury, ale o różniących się mocach obliczeniowych.

Słowa kluczowe: modelowanie wydajności GPU, GPGPU, MERPSYS, CUDA, przetwarzanie równoległe, obliczenia wysokiej wydajności

Dziedzina nauki i techniki, zgodnie z wymogami OECD: nauki inżynieryjne i techniczne, inżynieria informatyczna.

ABSTRACT

In this work, we evaluate an analytical GPU performance model based on Little's law, that expresses the kernel execution time in terms of latency bound, throughput bound, and achieved occupancy. We then combine it with the results of several other research papers from the field, introduce equations for data transfer time estimation, and finally incorporate it into the MERPSYS framework, which is a general-purpose simulator for parallel and distributed systems. The resulting solution enables the user to express a CUDA application in MERPSYS editor using an extended Java language and then conveniently evaluate its performance for various launch configurations using different hardware units. An additional benefit is a possibility to exceed actual limits imposed by the hardware, e.g. amount of operating memory available on the host or the device, or the number of interconnected graphic cards. We also explain numerous aspects related to the GPU computing such as CUDA architecture and programming model, memory hierarchy, kernel execution process, development and analysis of CUDA applications, and provide a systematic methodology for extracting kernel characteristics, that are used as input parameters of the model. The entire implementation process beginning with writing a simple CUDA kernel, its compilation and decompilation, assembly analysis, modifications of the MERPSYS framework and definition of the model within it, was meticulously documented. The model was evaluated using kernels representing different traits and for large variety of launch configurations. We found it to be very accurate for computation bound kernels and realistic workloads, whilst for memory throughput bound kernels and degenerated scenarios the results were not as good, but still within acceptable limits. We have also proven its portability between two devices of the same hardware architecture but different processing power.

Keywords: GPU performance modeling, GPGPU, MERPSYS, CUDA, parallel processing, high performance computing

TABLE OF CONTENTS

GLOSSARY

API – Application Programming Interface
ALU – Arithmetic and Logical Unit
CPI – Cycles Per Instruction
CPU – Central Processing Unit
CPW – Cycles Per Warp
CUDA – Compute Unified Device Architecture
DAC – Divide and Conquer
DRAM – Dynamic Random Access Memory
ETI – Faculty of Electronics, Telecommunication and Informatics
FLOPS – Floating Point Operations Per Second
FMAC – Fused Multiply-Accumulate
FP – Floating Point
FPU – Floating Point Unit
GDDR – Graphic DDR
GPGPU – General-Purpose computing on a GPU
GPU – Graphics Processing Unit
HPC – High Performance Computing
ILP – Instruction Level Parallelism
IPC – Instructions Per Cycle
ISA – Instruction Set Architecture
JSON – JavaScript Object Notation
MLP – Memory Level Parallelism
MPI – Message Passing Interface
NVCC – NVIDIA CUDA Compiler Driver
NVVP – NVIDIA Visual Profiler
PCIe – Peripheral Component Interconnect Express
PTX – Parallel Thread Execution
P2P – Peer to Peer
RAM – Random Access Memory
SASS – Shader Assembly
SFU – Special Function Unit
SIMD – Single Instruction Multiple Data
SIMT – Single Instruction Multiple Threads
SM – Streaming Multiprocessor
SPMD – Single Program Multiple Data
VC – Volunteer Computing
WLP – Warp Level Parallelism
WPC – Warps Per Cycle

This page intentionally left blank.

CHAPTER 1. INTRODUCTION

Graphics processing units (GPUs) are highly data-parallel devices that are nowadays ubiquitously used for a wide array of applications: graphics rendering, video processing, image analysis, etc. These are the most obvious use cases, but there exist many more areas in which GPUs excel, areas where the GPGPU paradigm is employed. GPGPU stands for general-purpose computing on a GPU and shortly speaking means that the GPU is used to execute tasks that it was not originally designed for, tasks which are not necessarily related to graphics and are similar to what the CPU does – general purpose tasks. Examples of such are physical simulations, numerical algorithms, neural networks training, data analysis, and many, many more.

To effectively use what the GPU offers, the programmer needs to write a dedicated application often combining several frameworks and libraries. Furthermore, the GPU differs greatly from the well-known CPUs both in terms of a hardware design and processing model. It would be of a great help to such programmer to have a tool that allows creation of a theoretical model of an application and provides means to assess it in terms of computational complexity, scalability and behavior on different hardware units. This becomes even more important when the GPUs are used in highly parallel and distributed environments such as HPC clusters, grids or volunteer computing networks.

What is more, according to the TOP500[1] ranking of the fastest supercomputers the accelerators (mostly NVIDIA GPUs) are an important component used in 18% of these systems as shown in Figure 1.1, which is an updated version of what we have reported in our previous work [1]. Considering NVIDIA's dominance in this field, size of the CUDA community and maturity of available software tools, adding to it the availability of the hardware for testing purposes on the ETI faculty, we find it well-justified to focus our efforts on the NVIDIA GPUs. Nevertheless, we want to develop a generic solution, which could be applied to other devices and GPU programming frameworks as well.

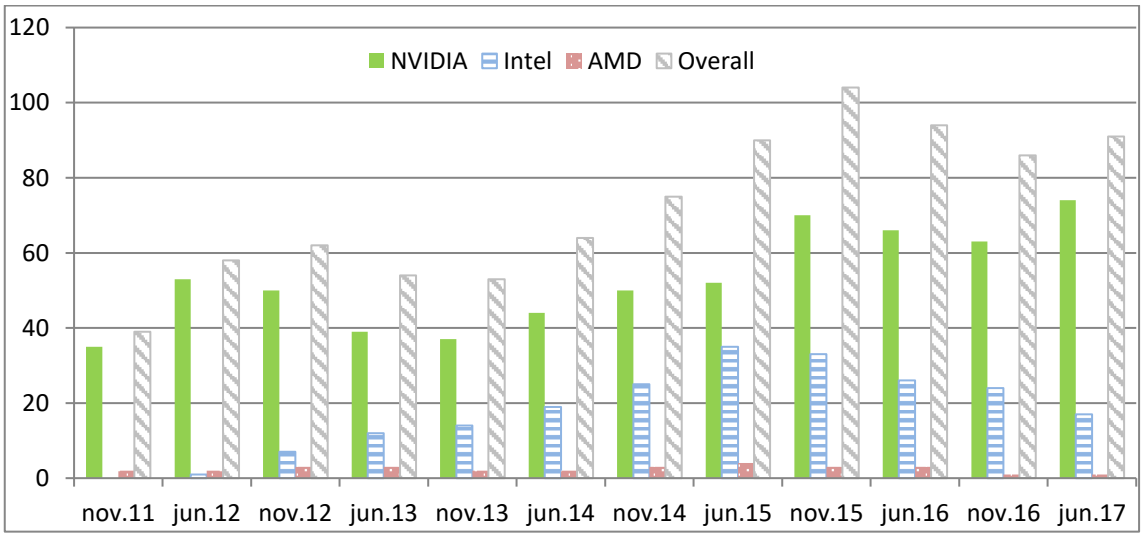

Figure 1.1. Accelerators use per TOP500 ranking

[1] https://www.top500.org/ (accessed 16 July 2017)

Using a simulator equipped with a proper theoretical model may be beneficial in many ways, one can use it to analyze the behavior of an application in hardware setups which may be hard to obtain or even unavailable for testing purposes. It also allows to exceed limits imposed by the hardware and for example assess the relation between execution time and data size for very large sets of the data. Despite providing estimations for application running time, a simulator may also compute the predicted energy consumption or failure chance. These values may be then used for multi-criteria optimization, including energy efficiency and reliability.

When searching for an appropriate theoretical model one should consider only these of acceptable (preferably as high as possible) accuracy, but this is not the only expected trait. Ease of use, extensibility and customizability are important as well and lastly, the closer the model resembles the actual processing that happens on the GPU, the better. The complexity of the GPU hardware and abundance of internal parallelism makes such a theoretical model harder to develop, but as we will further show, this is still a manageable task.

This thesis addresses the topic of modeling parallel processing on a GPU and furthermore aims to incorporate the selected model into an existing simulator – MERPSYS [2]. In the scope of this work we analyze existing solutions and based on them propose a performance model for a GPU. This model is then integrated into the MERPSYS framework and validated using a parallel GPU-enabled application with various launch configurations.

The paper is organized as follows, the first chapter is an introduction, which gives overview of the thesis topic and outlines important facts. The second chapter lists and explains all important aspects related to the GPU processing and briefly describes the MERPSYS framework. In the third chapter, we browse through related work in fields of general-purpose simulation and GPU performance models and then proceed to the fourth chapter, which is a proposition of our own solution based on the models described in preceding chapter. The fifth chapter is a complete walkthrough for the entire implementation process of the model, including CUDA application development and analysis techniques, it also contains a description of how the model was incorporated into MERPSYS. The model is validated against different input parameters in chapter 6, whilst chapter 7 summarizes what was achieved and suggests the direction of future research.

CHAPTER 2. THEORETICAL BACKGROUND

2.1. Introduction

To smoothly enter the realm of GPU computing we start off with a brief introduction that will allow us to build a required knowledge base and make ourselves familiar with the terminology and concepts used throughout this work. As it was already mentioned in previous chapter, this work focuses on NVIDIA GPUs, hence a very large part of this one was devoted to a description of the architecture and inner workings of these units. At the beginning, we concisely lay out the history of the GPGPU so that the reader knows why such solution emerged and what problem does it aim to solve. Next comes the description of the CUDA architecture and programming model, which is a practical implementation of the GPGPU paradigm.

In the part related to the GPU architecture the very design of the hardware is described, starting with high-level elements comprising the GPU, through memory hierarchy and internal structure of streaming multiprocessor, and as far as showing how the code is executed. We also introduce and explain the meaning of the important terms related to the GPU computing and CUDA, like: kernel, warp, streaming multiprocessor, compute capability, and many more. Additionally, we show the differences between the subsequent GPU architectures and outline how the compilation process works for CUDA code and what are the products.

In the next section, we show the similarities and differences between CUDA and OpenCL, another popular language used for parallel computing. Lastly, MERPSYS is described, which is a general-purpose simulator for parallel and distributed computing that was developed at the department of computer architecture of the ETI faculty, and to which the model for GPU processing must be added. In this chapter, we also show the important aspects related both to the CUDA architecture and MERPSYS, that need to be considered when designing a solution for modeling parallel processing on a GPU.

2.2. GPGPU paradigm

The focus of this work is modeling of the GPU so let us look briefly at the history and evolution of the graphics processing units. The need for an affordable accelerator targeted specifically at compute intensive rendering tasks came when first big game titles employing 3D graphics, very advanced and revolutionary at that point, reached big audience in the mid-1990s [3]. Graphics rendering is all about performing various transformations on input data in a well-defined order of steps that results in a final image being produced and displayed on the screen. At that point, graphic pipelines were very rigid, not customizable, and the only way of making an interaction with them was through the APIs designed for computer graphics like OpenGL[2] or DirectX[3]. The first major change came in the early 2000s when vertex and pixel shader engines

[2] https://www.opengl.org/ (accessed 12 August 2017)
[3] https://msdn.microsoft.com/en-us/library/windows/desktop/hh309467(v=vs.85).aspx (accessed 12 August 2017)

were exposed to the developers and it was now possible to program them using a shading language like HLSL or GLSL [4].

A pixel shader is a programmable arithmetic unit that uses pixel coordinates together with some context information (colors, etc.) to produce a final color of that pixel that will be displayed on the screen. It was soon discovered by the researchers that with the complete control over the shading unit behavior handed over to the programmers, the input and output of pixel shader can be any arbitrary data, not just a color, given that the constraints imposed by the underlying hardware are met. They key point here is that the GPU was used in a way it was not designed to, but nevertheless it performed very well in this task, however, there were several downfalls of such approach:

- the whole process was very complicated and most certainly not easy and appealing for the programmer,
- programs could receive input data only in a form of very well-defined structure of colors,
- access to arbitrary memory locations was not possible, there were serious constrains on how the result may be written to memory so whole class of algorithms that need to perform scatter operations were ruled out,
- unavailability of user-defined data types,
- instruction set targeted specifically at the graphic operations,
- the GPUs were using non-standardized floating point model. This is not an issue with image rendering where small discrepancies in the result will not even be noticed but crucial for scientific computations,
- debugging programs running on a GPU and finding problems in one's code in the event of application crashes or hangs was, mildly speaking, not an easy job,
- programmer must have gotten familiar with OpenGL or DirectX to make an interaction with the GPU rendering pipeline,
- actual computations must have been written using a shading language.

Still, the possibility of performing a general-purpose computation on a GPU was very appealing and the GPGPU paradigm was just about to rise. A breakthrough came in 2006 with the release of the first Tesla architecture based GPU – NVIDIA GeForce 8800 GTX, the G80 chipset embedded in this device was the first one built using the all-new CUDA[4] architecture. Not only the architecture was new, but also a whole programming environment: CUDA C++/C compiler, libraries, runtime, debugger.

CUDA architecture introduced many significant improvements and changes into the GPU chip design, naming a few:

- memory load and store instructions with random byte addressing capability allowing to access arbitrary location of the device memory,
- more generic parallel programming model with a hierarchy of parallel threads,
- barrier synchronizations and atomic operations,

[4] https://www.nvidia.com/object/cuda_home_new.html (accessed 25 April 2017)

- a unified shader pipeline, allowing every arithmetic logic unit (ALU) on the chip to be marshaled by a program intending to perform general-purpose computations, in previous generations shader units were either pixel or vertex shader,
- compliance with IEEE requirements for single-precision floating-point arithmetic,
- instruction set fit for general computation rather than specifically for graphics,
- access to a software-managed cache known as shared memory.

Nowadays a CUDA capable GPUs are getting more-and-more widespread and the scope of their applications broadens, examples include: medical imaging, computational fluid dynamics, environmental science, image processing, automotive industries, structural dynamics, financial sector, national defense and many, many more.

There is also a major drive in the direction of GPU-powered deep learning AI, a recent study[5] by NVIDIA shows that the number of organizations engaged in a deep-learning solutions development increased from 1500 in 2014 to almost 20000 in 2016. It is no surprise that NVIDIA released a solution dedicated to deep-learning applications – DGX-1[6], consisting of eight Tesla P100 GPUs connected using NVLink and able to deliver performance of up to 170 teraflops for half-precision computations.

CUDA software stack coupled with NVIDIA GPUs, although being the most popular setup for GPGPU, is not the only one available. Alternatives in the form of OpenCL[7] and OpenACC[8] exist, both capable of targeting not only the NVIDIA GPUs but also AMD FirePro[9, 10] (especially the S-series dedicated to HPC).

2.3. CUDA GPU architecture and programming model

Unlike in the past, when single-core CPUs dominated the market, parallelism is now an essential part of the programming model for the still growing number of applications, this trend affects both the application and hardware designs. Two fundamental types of parallelism may be identified: task parallelism and data parallelism. We are encountering task parallelism when there are many tasks that are being executed in parallel by more than a single computational unit and so its focus is on distributing the tasks among available processors. Data parallelism on the other hand, takes place when many data items are distributed among available processors and processed in parallel. CUDA programming model is perfectly suited for data parallel applications because of the GPU design [5], which we will discuss in this chapter.

GPU is a many-core[11] device that exemplifies a SIMD (Single Instruction Multiple Data) architecture type per Flynn's taxonomy [6, p. 18]. In this type of architecture multiple cores execute the same instruction stream on a different input data, this type of processing is

[5] https://www.nvidia.com/en-us/deep-learning-ai/industries/ (accessed 25 April 2017)
[6] https://www.nvidia.com/object/deep-learning-system.html (accessed 25 April 2017)
[7] https://www.khronos.org/opencl/ (accessed 25 April 2017)
[8] http://www.openacc.org/ (accessed 25 April 2017)
[9] https://www.amd.com/en-us/solutions/professional/hpc (accessed 25 April 2017)
[10] http://developer.amd.com/tools-and-sdks/opencl-zone/opencl-resources/ (accessed 25 April 2017)
[11] The term many-core is usually used to describe multicore architectures with an especially high number of cores (tens or hundreds).

ubiquitous and can be found not only in the GPUs but also in the CPUs and co-processors like Intel Xeon Phi in a form of vector instructions (e.g. SSE / AVX instruction sets). Although the GPU itself does not contain vector units but rather simple cores capable of executing one instruction on a single operand at a time (or two instructions if we consider FMAC operations), it acts in a very similar manner. All the threads comprising a single work group (called warp) in a single clock cycle execute the same instruction on different operands, that's why NVIDIA describes this architecture as SIMT [7] (Single Instruction Multiple Threads). This term is obviously a direct derivative of SIMD but expresses the internal architecture of the device more clearly - multiple threads executing an instruction instead of a single thread executing it on a wide register containing multiple operands.

2.3.1. CUDA Programming model

A CUDA program is heterogeneous in its nature and consists of two parts: one that executes on the CPU and one that is offloaded to the GPU, upon the program launch the code is executed by the CPU until kernel invocation is encountered (see Figure 2.1). Kernel is a function written using CUDA extensions to the programming language that will be executed on the device, if not explicitly stated then the kernel is scheduled to launch using the default stream. A stream is a sort of work queue for the GPU and all operations from the same stream execute sequentially (memory copies, kernel launches), multiple streams may be present but this is an advanced topic not important in scope of this work and will be omitted. When launching a kernel the number of threads that will execute the code in parallel must be specified, threads are grouped in thread blocks and blocks are collectively called a grid. A grid of threads is executed on the device in a highly parallel manner until every single of them finishes its work, what indicates the completion of the kernel launch and allows the next instruction from the current stream to be processed. Kernel launch is an asynchronous operation and execution on the host continues until an explicit[12] or implicit[13] synchronization point is encountered. Host and the device maintain and manage their own memory spaces and so the data must be moved between them either by performing explicit transfers or using mechanisms like Unified Memory[14,15], in which case the runtime environment manages the data copies transparently to the programmer.

[12] API call like *cudaDeviceSynchronize().*

[13] For example, *cudaMemcpy*. Operations from a single stream are executed sequentially, hence a memory copy will only be able to complete after the kernel launch that has been scheduled earlier.

[14] https://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/ (accessed 2 July 2017)

[15] https://devblogs.nvidia.com/parallelforall/unified-memory-cuda-beginners/ (accessed 2 July 2017)

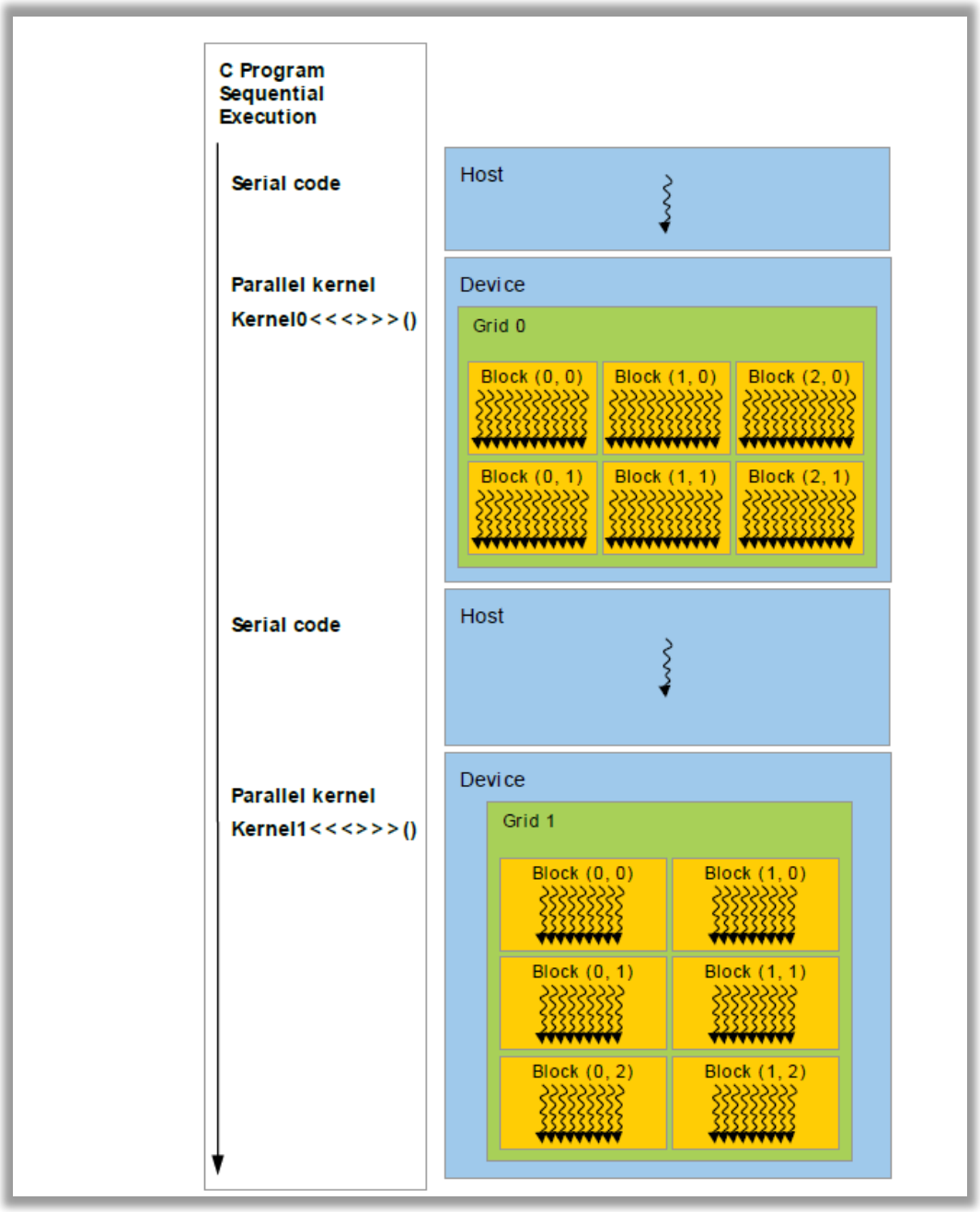

Figure 2.1. Heterogeneous execution model of CUDA (source: *CUDA C Programming Guide* [8])

2.3.2. Architectural differences between GPU and CPU

Even though the many-core term can be used to describe both the GPU and contemporary CPUs[16], the core of a GPU differs a lot when compared to a CPU core. The core of a CPU is equipped with complex circuitry allowing it to effectively handle complex control logic and execute programs exhibiting low degree of parallelism and substantial divergence in control flow as efficiently as possible. This is achieved by employing branch target buffers, dynamic instruction reordering and out of order execution, complex pipelines, higher core clock

[16] 48 logical cores for Intel Xeon E7-8890 v4, 248 logical cores for Intel Xeon Phi 7230F.

frequencies and hierarchical caches. CPU excels in executing tasks with complicated and highly unpredictable control flow logic but small input sizes, when compared to the GPU.

GPU core on the other hand is simple in its design with ALUs taking most of the die area (see Figure 2.2), does not handle control flow divergence well due to lack of branch prediction, relies on software managed caches (shared memory) instead of hardware cache and does not reorder instructions as aggressively although some ILP (Instruction Level Parallelism) exists. For this reasons, the GPU is perfectly suited for handling tasks that are highly data parallel and compute intensive, yet simple in terms of the control logic.

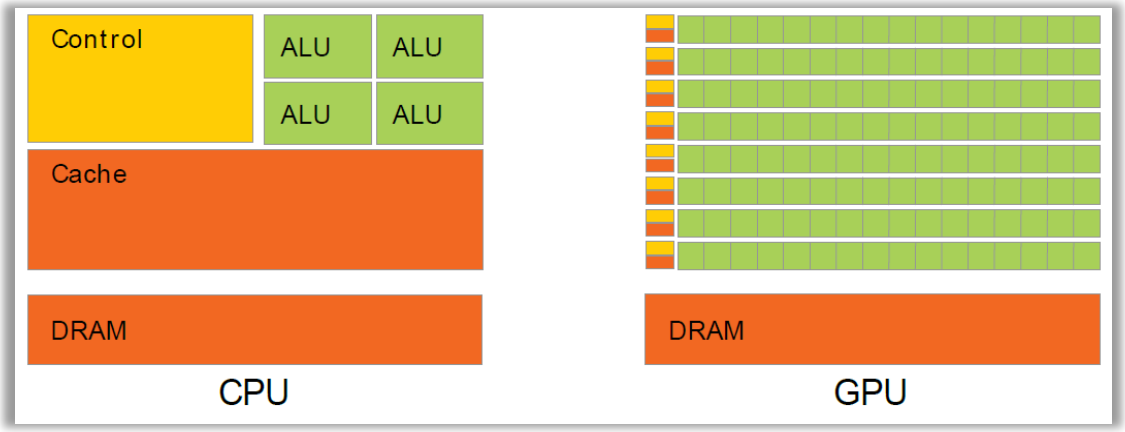

Figure 2.2. Architectural differences between GPU and CPU cores (source: CUDA C Programming Guide [8])

Significant differences are also present when we look at both kinds of devices at the thread level. CPU threads are generally heavyweight with complex execution contexts that need to be swapped when the operating system decides to switch from executing one thread to another. Context switches are the key mechanism that provides multithreading and multitasking in contemporary systems, and as already mentioned every switch requires a very large number of clock cycles and incurs a significant performance penalty. Threads on GPUs are extremely lightweight and typically thousands of them are active at the same time. Context switches do not incur any performance penalty because the register file on each of the SMs is large enough to store the state of every single queued thread for its entire lifespan. If one warp is stalled for some reasons, then another one will begin executing after zero-overhead context switch. Furthermore, CPU cores are designed to minimize the latency of the threads being executed while GPU cores aim to maximize the overall throughput and hide the latency mostly with warp-level parallelism [5, p. 13].

Bringing up some numbers, contemporary CPUs like Intel Xeon E7-8890 v4 will be able to execute tens of threads concurrently (48 in this case to be specific) while the Tesla P100 GPU supports up to 114688[17] resident (loaded to SM and queued for execution) threads [9].

[17] 2048 (max number of resident threads on the SM) × 56 (number of SMs).

2.3.3. GPU hardware architecture

Several major elements ought to be identified in the GPU architecture [10], these are:

- host interface,
- copy engines,
- DRAM Interface,
- streaming multiprocessors (SMs).

Host interface reads commands (for example memcpy operations and kernel launches) issued by the host and dispatches them to appropriate hardware units on the GPU, it also facilitates synchronization of the device with rest of the devices present on the system, be it CPUs or GPUs.

Copy engines can perform memory transfers between the host and the device while the SMs are doing computations. They also provide means for DMA (Direct Memory Access) and are responsible for saturating the PCIe bus to use it as efficiently as possible. At most two of the copy engines are required since they will be able to saturate both directions of PCIe bus [10, p. 42].

DRAM interface consists of device's global memory, hardware controller capable of coalescing memory accesses and L2 write-through cache shared among all SMs. It provides very high bandwidth reaching hundreds of gigabits per second

A single GPU may contain up to several dozens of SMs, and depending on the device architecture significant changes are present in their internal design. Some of the differences in the architectures are important from the point of view of this work and will be covered in section 2.3.6. Streaming multiprocessor is the key component of a GPU that performs all compute tasks, it is comprised of:

- basic execution units (CUDA cores) that perform 32-bit integer as well as single, and half (since Pascal architecture) precision floating-point arithmetic,
- double precision units that perform double precision floating-point arithmetic,
- special function units (SFUs) to compute single-precision approximations of transcendental functions, reciprocals, etc.,
- warp schedulers to coordinate instruction dispatch to the execution units,
- caches – constant, L1, texture (or unified – depending on architecture),
- shared memory for data interchange between threads in a thread block,
- dedicated hardware for texture mapping,
- load / store units that process memory access instructions and generate necessary memory transactions,
- register file from which registers are allocated to the threads.

2.3.4. Kernel execution model

When a kernel launch command is invoked two important parameters are specified: grid size and block size. Grid is a three-dimensional mesh of thread blocks and a thread block is a three-dimensional mesh of threads. A linear or two-dimensional ordering may be achieved when proper dimensions are set to 1, a single thread block may consist of up to 1024 threads and the number of thread blocks in a mesh is virtually unlimited[18], Figure 2.3 shows the hierarchy.

As the grid may comprise billions of threads it is logical that only a limited number will be loaded to the SMs at a given time, allocation is done at the block level and once a block finishes its execution a new one will be scheduled, this process continues until all of the blocks are processed [11, pp. 5-7]. Without going into deeper details, from our perspective important are the following facts:

- SM may hold a limited number of blocks at a time,
- blocks have their requirements for shared memory and registers which need to be allocated from the pool available on the SM,
- threads are executed in groups of 32, called warps,
- there are constraints regarding maximum number of active warps as well as rules affecting warps execution,
- SM can perform zero-overhead context switches between warps.

Above directly affect the capability of the SM to execute the instructions effectively, if the number of active warps is not large enough to cover the stalls (caused by memory accesses, conditional branches causing warp divergence, etc.) then the performance will be suboptimal [12]. The process of calculating the number of threads that may be loaded to a single SM considering shared memory and register constraints is called occupancy calculation[19]. Shortly speaking, if the occupancy is poor then likely the performance of the application will decrease[20]. It should however be noted that increasing occupancy may not lead to any performance gains or may even result in a performance degradation [13] because of an increased contention for shared resources as was experimentally proven in section 4.7 of paper by Bakhoda, et al. [14]. It is also entirely possible to achieve optimal performance with a very low occupancy if thread granularity (number of instructions executed per thread) is sufficient [15]. Given all these, it is clear that the effectiveness of warps execution depends on the characteristics of the specific kernel.

[18] There are constraints regarding the width of each dimensions but these are large enough to omit them for practical purposes, for the detailed information see appendix G from *CUDA C Programming Guide* [8].

[19] Precisely speaking, occupancy is the ratio between the number of active warps per multiprocessor and the maximum number of possible active warps, see chapter 10 of *CUDA C Best Practices Guide* [17] for details.

[20] This is true if for given occupancy the kernel is limited not by throughput but by the latency, a most common latency bound is the one resulting from memory accesses, it occurs when the level of parallelism is insufficient to hide memory access latencies. For more information see section 3.3.7.

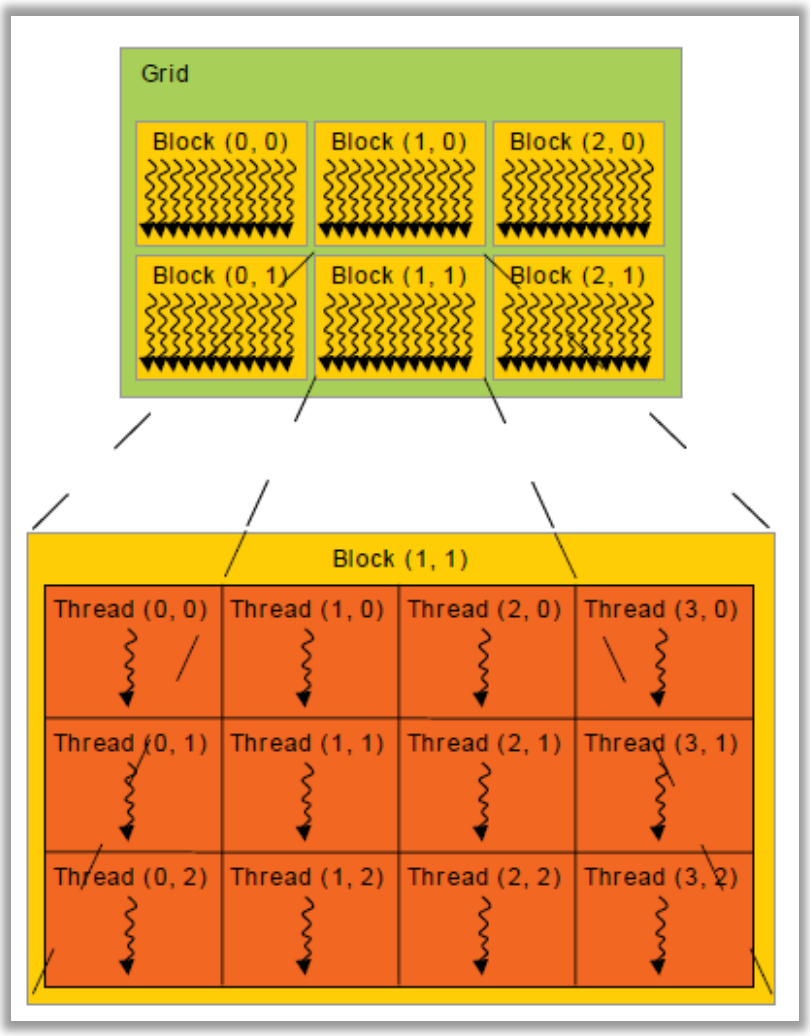

Figure 2.3. Two-dimensional ordering of threads in a block and blocks in a grid (source: *CUDA C Programming Guide* [8])

2.3.5. GPU memory hierarchy

The big differentiator of the GPU architecture is its memory hierarchy. We can distinguish two kinds of memories: software and hardware managed. When we consider CPUs, there is a programmatically controlled DRAM shared amongst all threads backed up by a multi-level, hardware managed cache and the registers. In turn, GPU offers a much wider variety of memories and grants much bigger control to the programmer, programmatically controlled are: global, local, constant, texture, shared and register memories (see Figure 2.4), also there are L2, L1, constant and texture caches managed by the hardware. A very deep insight into the GPU memory hierarchy and caching policies can be found in work by Mei and Chu [16]. We will now briefly discuss each type of the memory available on the GPU.

Global memory has the largest amount of space available, but its drawbacks are highest access latency and lowest throughput when compared to other memories. It is located off-chip, shared between all thread blocks and takes an important part in data copying between host and the device. What is important from the performance point of view it that the accesses to it may be performed in 32, 64 or 128 bytes wide transactions. Coalesced (referencing

consecutive locations of memory) accesses may be realized as a single, performance-effective transaction, uncoalesced accesses are split into separate transactions and thus hamper the performance. Global memory is cached in L1 and L2 caches depending on the device's compute capability.

Local memory is located in off-chip DRAM and is cached in L1/L2, same as global memory, the difference is that it is exclusive to a single thread and that the compiler ensures that the accesses are always coalesced. It is used transparently to the programmer, what this means is that there are several rules stating when the local memory will be used by the compiler, i.e.: the registers are unable to hold all local variables used by the thread (this is called register spilling) or arrays are indexed with variables that are not compile-time constants.

Constant / texture memories are read-only, located in DRAM, cached respectively in specialized constant / texture cache, with the latter being optimized for the 2D spatial locality. Both are not important from our perspective so we will restrain from describing them more accurately.

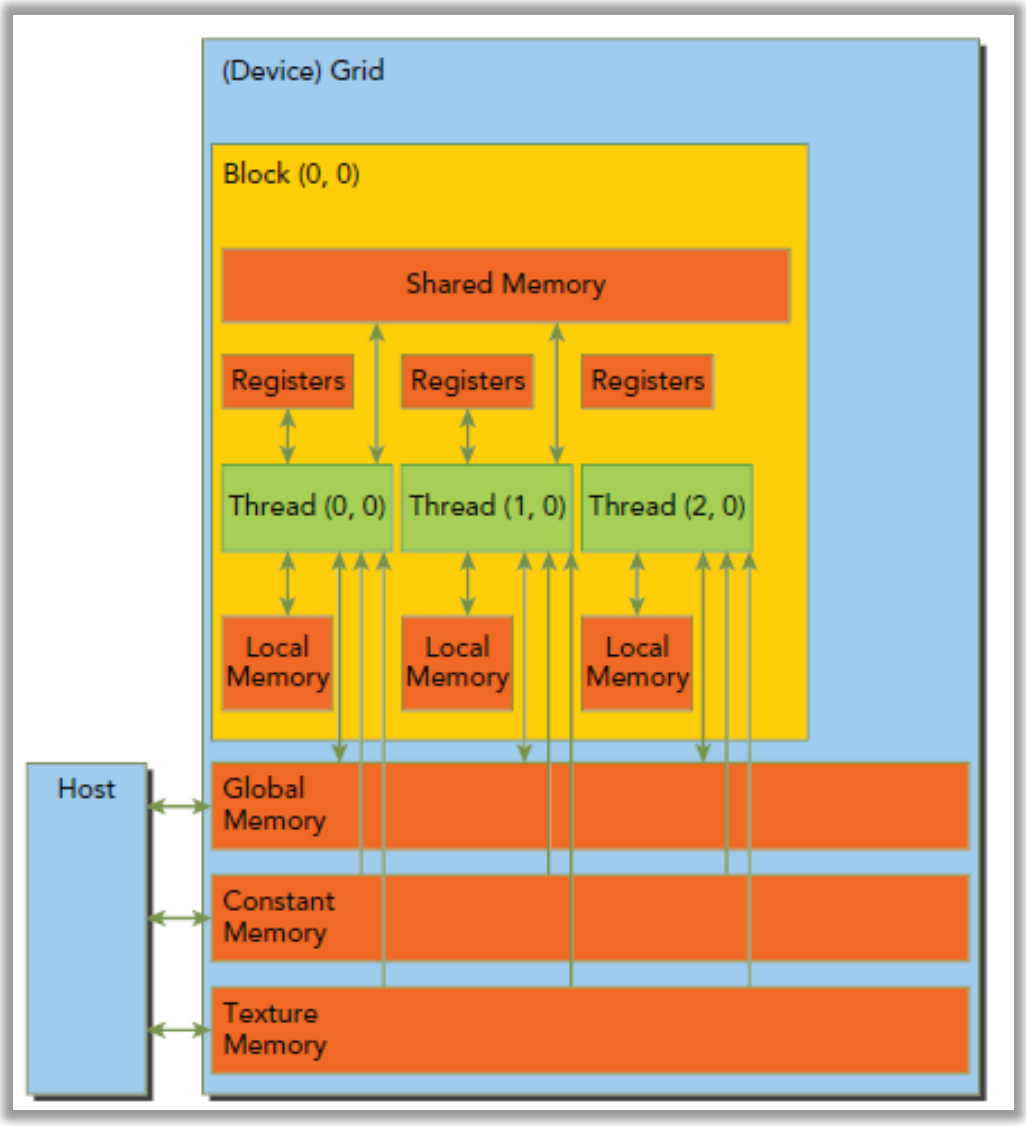

Figure 2.4. GPU memory hierarchy (source: *CUDA C Programming Guide* [8])

Shared memory is located on-chip, exclusive to one thread block and offers significantly better throughput and latency when compared to aforementioned memories. It is divided into separate banks which allow to parallelize accesses to it as long as the threads are accessing separate (in most cases consecutive) banks. When two threads access one bank at the same time then the accesses are serialized, this is called a bank conflict and decreases the performance. Furthermore, it is a good programing practice to load frequently accessed data from global to shared memory.

The fastest, both in terms of throughput and latency, and the smallest of the available memories are the registers. They are exclusive to a single thread, allocated and stored in the register file and not cached in any way as there is no need to do so.

As it can be easily noticed memory access patterns are very important part of the CUDA programming and one should take great care to access the memory in an effective way. A good reference in this topic is the section 9.2 of *CUDA Best Practices Guide* [17].

2.3.6. Compute capability

Compute capability is a version number comprising major and minor revision that identifies hardware design and hence the features supported by the GPU. GPUs with the same major version number share the same core architecture, following architectures were released so far:

- 1.x – Tesla (unsupported since CUDA version 7.0),
- 2.x – Fermi [18],
- 3.x – Kepler [19],
- 5.x – Maxwell [20],
- 6.x – Pascal [9],
- 7.x – Volta [21].

Compute capability should not be confused with the CUDA version, which is the version of software platform used by developers for writing CUDA-enabled applications and launching them on NVIDIA GPUs. As of June 2017, the latest CUDA version is 8.0 and 9.0 is soon to be released together with Volta architecture [22]. When developing a theoretical model for a simulation it is also important to consider architectural differences like: caching policies, number and type of CUDA cores, instruction latencies, available shared memory and so on, as those have a direct effect on the kernel execution. Several such differences are listed in Table 2.1 and Table 2.2, for a detailed description please refer to chapter G of *CUDA C Programming Guide* [8]. Instructions latencies are covered in section 4.4.

Table 2.1. Architectural differences depending on the compute capability

Compute capability	2.0	2.1	3.x	5.x	6.0	6.1; 6.2	7.0
CUDA cores per SM	32	48	192	128	64	128	64[21]
SFU units per SM	4	8	32	32	16	32	16
Warp schedulers per SM	2	2	4	4	2	4	4
Max instructions issued by a single scheduler per warp per cycle	1	2	2	2[22]			2
Global memory caching policy	L1 + L2 (default); L2 only (optional)		L2 only (default); optionally also in L1 for read-only data	L2 only; L1[23] for read-only data and optionally for mutable data			L2 + L1[24]

Table 2.2. Shared memory and register constraints depending on the compute capability

Compute capability	2.x	3.x	5.x	6.x	7.x
Max shared mem per SM [KB]	48	48; 3.7 - 112	5.0 / 5.3 - 64; 5.2 - 96	64; 6.1 - 96	96
Max shared mem per block [KB]	48				
Max 32bit registers per SM	32K	64K; 3.7 - 128K	64K		
Max 32bit registers per block	32K	64K; 3.2 - 32K	64K; 5.3 - 32K	64K; 6.2 - 32K	64K
Max 32bit registers per thread	63		255		

2.3.7. Compilation process

CUDA code written in programming languages like C, C++ or Fortran can be compiled into two representations: PTX (Parallel Thread Execution) or SASS (Shader Assembly). PTX is a model of virtual machine and a definition of ISA to be used with this machine, it exposes GPU to the higher layers as a data-parallel computing device. PTX code is an intermediate representation of a program and is translated into a machine code (cubin object) either by NVCC[25] in a separate compilation step or by leveraging just-in-time compilation by the CUDA runtime driver upon being scheduled for launch on a specific device [23]. This allows a high level of interoperability between varying GPU architectures. SASS is the machine code for GPU, generated for a specific architecture and maps directly to instructions being processed by the functional units. It can be either generated directly by NVCC during the compilation process or, as it was already mentioned, generated on-the-fly from PTX source code.

2.4. OpenCL

OpenCL [24] is a software stack similar to CUDA in many aspects but more generic and targeted at a wider range of devices, not only does it support computations on a GPU but also on CPUs and other accelerators. It provides a layer of abstraction over the hardware by defining its own constructs and makes it easy for the programmer to write a parallel code in a highly

[21] Unlike previous architectures INT32 and FP32 datapaths are parallel.
[22] NVIDIA documentation is inconsistent in this matter, see chapter 2.9 of Volkov's work [49].
[23] Since Maxwell, unified L1 cache is used for global and texture memory accesses.
[24] This information is partial and comes from a post [21] on official NVIDIA's blog. There is also an announcement that the L1 unified cache and shared memory are now the same physical subsystem.
[25] NVIDIA CUDA compiler [23]

platform-independent manner. No surprise that it is also widely used and thus should be considered here as well.

The abstract hardware model used in OpenCL can be mapped directly to the one known from CUDA. At the highest level, there is a platform (host machine) equipped with one or more compute devices (GPUs), a compute device is comprised of compute units (SMs) which contain processing elements (CUDA cores).

The execution model of OpenCL [25] also distinguishes two parts of the code: kernels that will be executed on one of the available devices and a host program. Kernel is launched within a context created and managed by the host, context defines the execution environment and is comprised of device, memory, and kernel objects. A kernel object contains kernel's source code loaded and compiled by an OpenCL API call, then the memory object holding arguments for the kernel call is assigned to it and the whole bundle must be enqueued in a command queue for execution. A command queue is associated with a single device and used to schedule commands like kernel invocation, memory copies or synchronization functions to run on it. It is in some way similar to the stream known from CUDA, it must be managed explicitly though. When a kernel is submitted for execution an index space called NDRange is created, it is composed of work-groups which are further decomposed into work-items, work-item is a counterpart of the thread from CUDA.

When an OpenCL kernel is executed on a CUDA-capable GPU, a direct mapping of work-groups into thread blocks and work-items into CUDA threads is performed. Then a grid composed of them that resembles the NDRange is loaded into the GPU and the kernel code is executed. Memory model of the OpenCL is divided into host and device memory, host memory is managed outside of the OpenCL and the device memory is further divided into:

- global memory – accessible globally for read and write to all work-items from all work-groups, hence shared amongst every work-items in a NDRange,
- constant memory – same as global memory but read-only,
- local memory – exclusive to a single work-group,
- private memory – exclusive to a single work-item.

Table 2.3. Comparison of CUDA and OpenCL

CUDA	OpenCL
Hardware model	
Host	Platform
GPU	Compute Device
SM	Compute Unit
CUDA Core	Processing Element
Execution model	
Kernel	Kernel
Stream	Command Queue
Grid of Thread Blocks	NDRange
Thread Block	Work-group
Thread	Work-item
Memory model	
Global Memory	Global Memory
Constant Memory	Constant Memory
Shared Memory	Local Memory
Local Memory / Registers	Private Memory

2.5. MERPSYS

Developed at the ETI faculty of Gdansk University of Technology, MERPSYS[26] [2] is a simulation environment for distributed systems that allows its users to predict execution time, power usage and failure probability of their applications. The application is described in a form of Java code imbued with special functions implementing communication (MPI-like) and computation primitives. The system is modeled as a hierarchical structure of components representing both computational devices and interconnects, each of them having its own characteristics defined. It consists of four software components: GUI, server, database and simulator.

Block based representation [26, pp. 18-31] is employed as a theoretical concept behind the application model, what this means is that every application may be expressed in a form of sequential blocks of three major types:

- computation – various arithmetic operations,
- communication – data exchange between running processes,
- read / write – accesses to data stored in shared memory, hard drives, etc.

The above is used in MERPSYS as a set of extensions to the Java language that allow to define computation and communication blocks while preserving the original structure of the code. All that is required from the user is to rewrite his application in a pseudo-code using properly parametrized (input data size, operation type, etc.) MERPSYS-specific constructs in place of actual computation or communication code. The term pseudo is perfectly justified here as no real actions will be performed and simulation blocks will be used instead. Templates for the most common paradigms of parallel computation are already available within the system, those include: master-slave, SPMD (geometrical parallelism), divide and conquer, and pipelined processing.

A system model is created by selecting hardware components and placing them in the editor's workspace. It may consist of several nested levels, for example: two machines connected using InfiniBand network and inside each of them a GPU and a processor connected via PCIe bus. Hardware model is a term used to describe the characteristics of a given hardware setup, such a model has special functions defined that reflect, e.g.: power usage formula for an active or idle component, communication time for P2P, scatter or gather operations, etc. These functions are then used by a simulator to calculate the effective time taken by various actions. Hardware components have attributes that describe their characteristics, for computational ones: performance and power consumption, while for a communication: bandwidth and startup time. Both the hardware model and component models are easily extensible and parametrizable. A database of hardware components is provided, that can be extended with new records. It should be further noted that the hardware model is sometimes referred to as *computational model*.

For each computational component user defines an arbitrary number of *labels* that are used to tie application and hardware models together. They are used in application model to

[26] http://merpsys.eti.pg.gda.pl (accessed 20 May 2017)

mark parts of the code representing distinct processes or threads and allow the scheduler to map them to the hardware components. Shortly speaking, label tells the scheduler how many processes of a given type may be launched on a given hardware component. Number of the labels, and thus the count of processes to launch is specified when launching a simulation allowing for effortless parametrization of the simulation runs.

A simulation configuration screen also allows to define attributes that are passed to the application model, an example of such may be a data size, block configuration of a GPU, etc. A request to launch a simulation is passed from GUI to the Server which in turn directs it to one of the simulators connected to one of its simulation queues, simulation queue to be used may be specified as well.

MERPSYS was successfully used by its authors to model execution time, energy consumption and reliability of DAC and SPMD applications [27], K-means algorithm and volunteer computing systems. At the time of writing it is not yet publicly available, but there are plans to make it so.

We will now discuss the simulator design in a greater detail whilst focusing on the aspects crucial to this work, a more detailed description providing better insight into the whole MERPSYS environment can be found in work of Czarnul, et al. [28]. Let us assume that we need to calculate execution time for a single computation block specified in the application model as in Figure 2.5, that is being launched against the hardware model shown in Figure 2.6. When such a construct is encountered during simulation it will be evaluated by the simulation engine to determine the amount of time that a real application would spend processing it. The number of theoretically issued instructions will be computed based on the input data size - *compDataSize* and complexity function - *computationalComplexity*. It will be then used by *getTime* shallow function[27] defined in the hardware model that is used by the simulator to calculate the execution time based on the input data size.

```
1. String computationalComplexity = "function " + ConstVar.complexityFunctionName +
2.    "(" + ConstVar.parameters + ") {" + "return " +
3.    ConstVar.getDataSize + "*" + ConstVar.getDataSize + "*5; }";
4. sim.computation(compDataSize, ComputationType.GPU, SoftwareStack.Undefined, 1,
5.    computationalComplexity, OperationType.Calculations, OptimizationType.None);
```

Figure 2.5. An example computation block from the MERPSYS application model editor

[27] A shallow function is a term used in MERPSYS' hardware model to describe the generic functions used to compute various characteristics like communication and computation time, power consumption, etc.

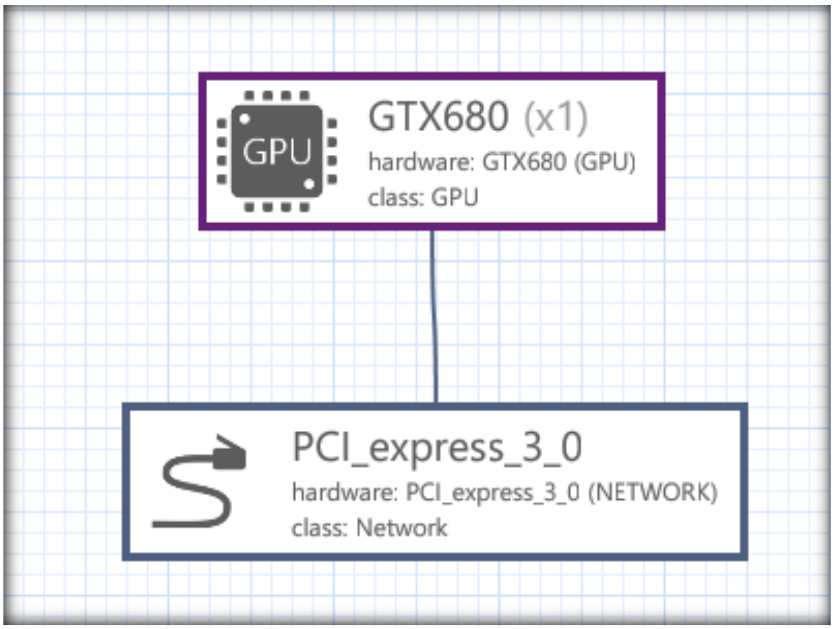

Figure 2.6. An example hardware device with an interconnect as shown in the MERPSYS hardware model editor

All of the shallow functions, and so the *getTime* function have the access to parameters defined for each device. A parameter is a characteristic of a hardware component like a power consumption, available shared memory, etc. Both the shallow functions and the device parameters may be easily added or customized using MERPSYS' Web UI. To give an example, in the process of calculating processing time the computational efficiency of the device must be used and such computational efficiency will be a parameter of the device. Figure 2.7 shows an example device together with its parameters, with *performanceS* being the mentioned parameter resembling the computational efficiency. Figure 2.8 shows how the parameter is used in the *getTime* function, as we can see the parameter is referenced directly in the function code and used to calculate the total time needed to complete the computations. This function is called by the simulator and the value returned is then added to the total execution time of the whole application, then a next event is processed[28].

Parameters of the simulation may be specified dynamically. In our example *compDataSize*, which is a global variable in the application model will have its value assigned during the simulation launch per the value specified in simulation launch screen, as shown in Figure 2.9. This allows for an easy parametrization of the subsequent simulation runs.

[28] MERPSYS is a discrete event based simulator, for the formulation of a discrete event simulation problem please refer to chapter 9.1 of book by Czarnul, et al. [26].

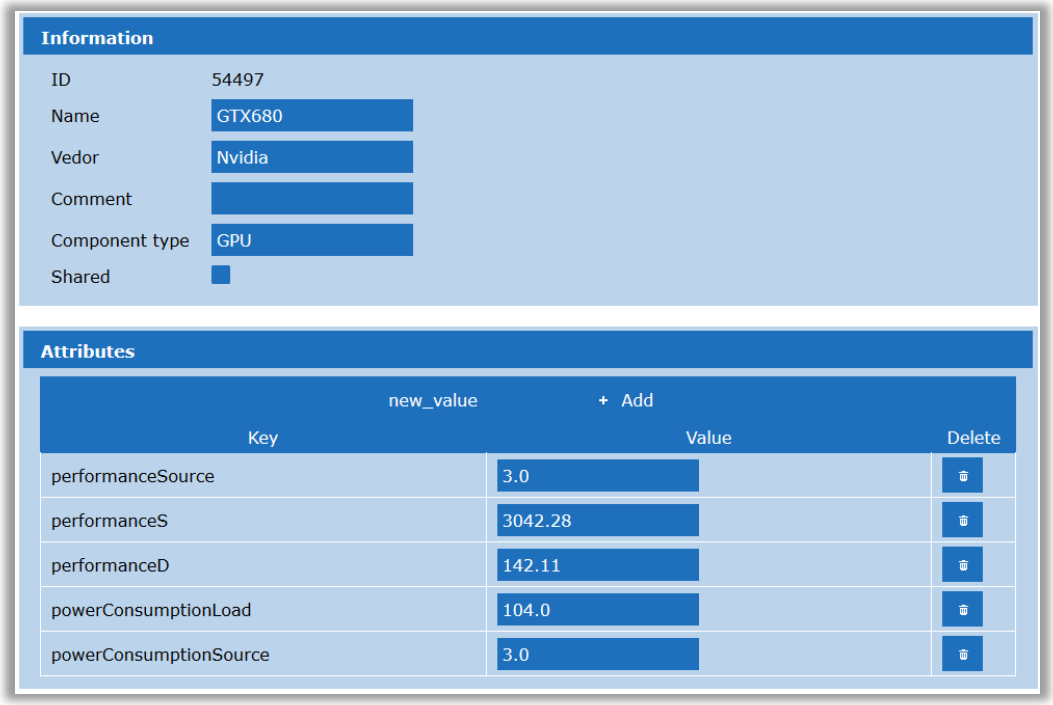

Figure 2.7. Device parameters

```
1. function getTime(parameters) {
2.    return parameters.get('processNumberPerInstance') *
3.       parameters.get('threadsNumberInEachProcess') *
4.       (parameters.get('numberOfOperations')) /
5.       performanceS;
6. }
```

Figure 2.8. Example function used in MERPSYS hardware model

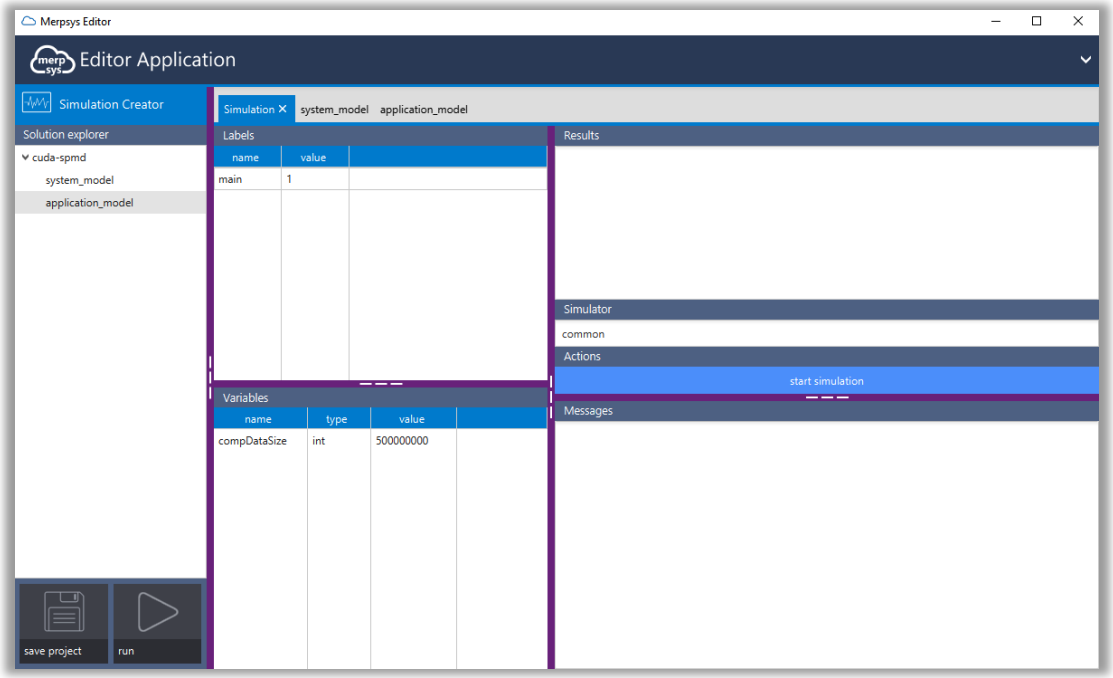

Figure 2.9. MERPSYS simulation launch screen

This page intentionally left blank.

CHAPTER 3. RELATED WORK

3.1. Introduction

Now that we are familiar with the GPU computing concepts and terminology, the time has come to browse through the existing solutions identifying their capabilities in terms of the GPU modeling. This chapter is divided into two major parts. The first one focuses on generic simulators, not particularly related to the GPU itself, but important because of their conceptual relationship to MERPSYS, which is also a general-purpose simulator capable of targeting a wide array of use cases. Approaches to defining hardware and application models and fitness for GPU modeling were scrutinized for these simulators and a simple comparison in a tabular form was prepared. Second part directly addresses the topic of modeling an execution of a kernel on a GPU, we study seven performance models briefly describing the ideas and assumptions behind them.

3.2. General-purpose simulators

Simulators discussed in this section are named as “general-purpose” because they are not tied to any specific device type or programming language and provide means to perform various kinds of simulation for many processing paradigms on parallel and distributed systems, these include: voluntary computing, grid computing, HPC clusters, cloud environments and more. Designing an application for such environments is not a trivial task as one needs to consider execution time, energy efficiency, reliability, maintainability and many other factors. Furthermore, distributed computing brings on new challenges like maintaining data consistency, effectiveness of data exchange, scalability and massive increase in the scale of computation. It is often the case that deployment-like environments are not available for testing and verification of the developed software; this is where simulation comes in to help [29, pp. 653-654].

All of simulators addressed in this study share a common trait, they implement discrete-event simulation model, what means that the entire process of simulation is governed by events being processed and passed between entities. Such an event may represent a computation, communication, synchronization or a resource access, with the details specified by the simulation environment itself. Figure 3.1 shows relationships between simulators that will be covered in this chapter, it is an extension of Figure 9.3 from the book *Modeling Large-Scale Computing Systems. Concepts and Models* [26]. An important fact about the simulation model is a decomposition of application and hardware models, where the application model represents the program being modeled and the hardware model is a definition of resources on which the mentioned program executes. Additionally, we look at these models in the scope of a GPU simulation, identifying their fitness for this purpose which is crucial from the perspective of our work. Table 3.1 is a comparison of all of the simulators covered, with an addition of MERPSYS, which was described in section 2.5.

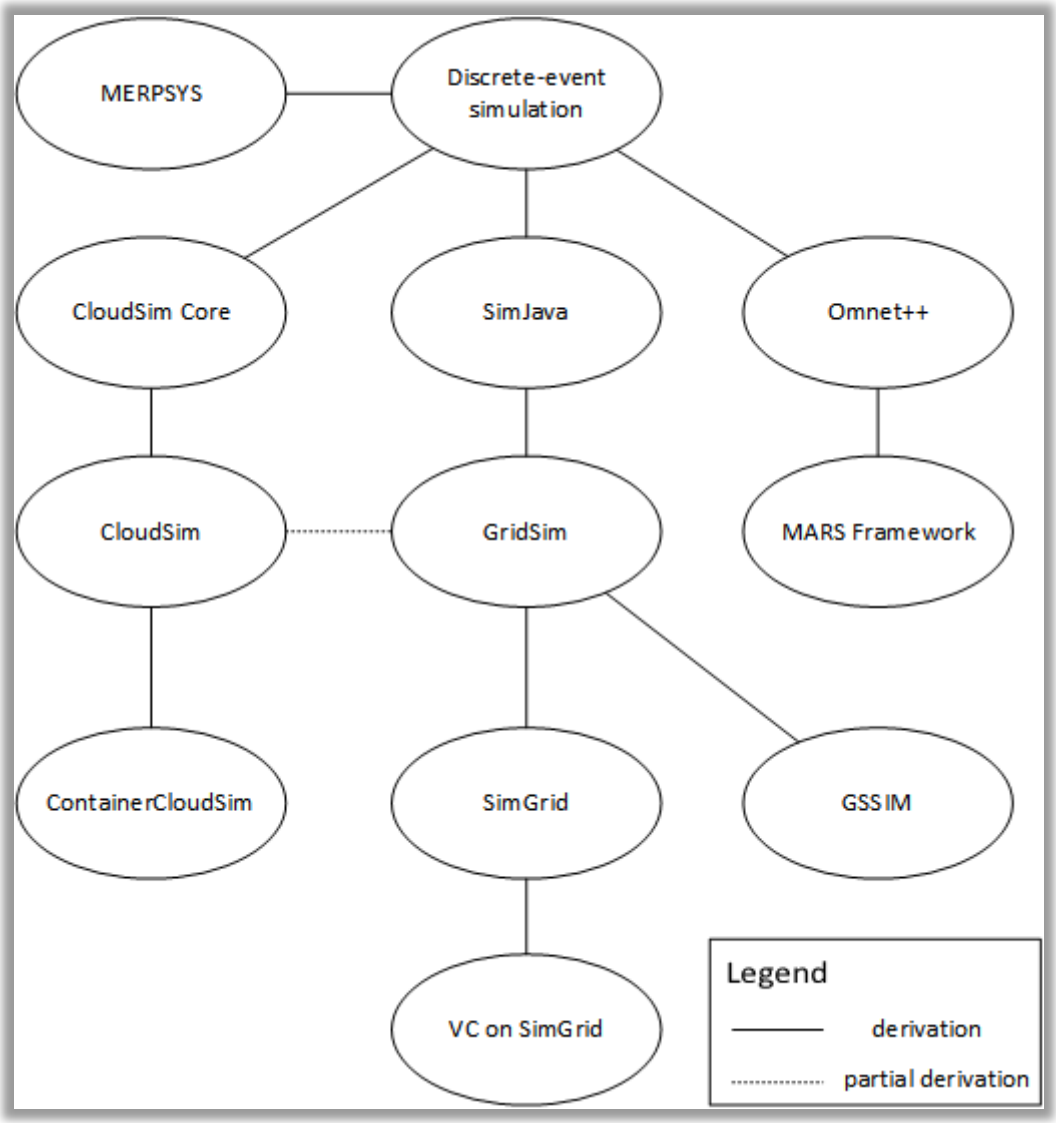

Figure 3.1. Relationships between simulators

Table 3.1. Simulators comparison

	Main focus	**System model format**	**Application model format**	**Communication between hardware entities**	**Level of detail available when modeling a hardware piece**	**Fit for GPU modeling**
GridSim	Large scale, geographically distributed systems (grids).	Written directly in Java by extending classes provided by framework, instantiating them and defining connections.	Defined in a form of gridlets.	Based on synchronous or asynchronous events.	Poor - computational power defined in terms of MIPS or SPEC rating.	Not at the level of detail we require.

	Main focus	**System model format**	**Application model format**	**Communication between hardware entities**	**Level of detail available when modeling a hardware piece**	**Fit for GPU modeling**
SimGrid	Cloud, grid, HPC, P2P, VC.	Defined using XML files.	SimDAG, MSG, GRAS, SMPI.	Analytical model of network is used.	High - complex analytical models may be used.	Theoretically yes, would require more in-depth investigation of the possibility to extend the existing framework (custom computational resource).
CloudSim	Various type of clouds, including hybrid and containerized environments.	Written directly in Java by extending classes provided by framework.	Defined in a form of cloudlets.	Event based, BRITE topology used to specify communication delay added to the events.	Operates at the level of entire VMs.	Not at all.
GSSIM	Grids.	Defined in XML files.	Workloads in SWF or GWF with XML extension.	Network flow model.	Plugins that take many factors into account may be written to describe the performance.	Possibly, by writing plugins to define complex application performance models.
MARS	HPC clusters with the focus on network optimizations.	OMNET++ network topology extended with custom model of a computing node.	MPI trace files (logs of MPI calls) obtained from real application execution.	OMNET++ messages sent through a simulated network.	Task modules execute logs containing MPI call traces, so there's not room for customization.	Not at all.
MERPSYS	HPC clusters, grids, VC.	Drag-n-drop editor to define hardware and connections. Model behavior defined by modifiable JavaScript functions. Customization using web UI.	Written in provided editor application using an extended version of Java.	Event based.	Very high, possibility to add custom functions describing device behavior, device is highly parametrizable.	Yes, the hardware model may be easily customized to use the additional data needed for GPU modeling. The data may be specified in the application model.

3.2.1. GridSim

GridSim [30] is a simulation toolkit targeted at performing simulations of large scale, geographically distributed systems called grids. Elements that partake in such systems, be it small nodes or large clusters, can be composed of a diverse hardware, and thus vary in terms of computing power, energy consumption and failure probability.

Geographical distribution and multiple administrative domains are also important factors to consider when scheduling application execution to specific nodes as it directly impacts the cost of computation, since the energy price may vary per provider and changes during the day, also data transfers between the nodes may be costly. Furthermore, reliability of computation must be ensured and that task is not an easy one in a highly-distributed environment. Considering all these things, the authors propose a generic framework based on SimJava discrete event simulation infrastructure, that allows to execute performance evaluation of grids in a repeatable and controlled environment.

The toolkit focuses on modeling scheduling algorithms for time-shared or space-shared heterogeneous resources with different I/O capabilities that are located in different administrative domains and time zones, these resources may be scheduled statically or dynamically. The toolkit allows to model various types of PE (Processing Elements), these can be: single processor, SMP (Shared Memory Multiprocessor), or a cluster of computers with a distributed memory. PE can be combined to form a bigger entity, which is then a part of the grid being simulated.

On the other hand, modeling of the internal structure of a hardware unit, like a GPU, is not possible. Units are simply assigned a processing power, defined in terms of MIPS or SPEC rating, so there is no field available for modeling the details of an execution on a single unit.

The grid (system) is modeled directly in the Java code by extending the base classes provided by the framework. Communication between simulation entities is realized either as synchronous or asynchronous events. GridSim does not explicitly define any specific application model, but authors claim support for various computational models: DAG, DAC or task farming. The application itself is defined in a form of *gridlets*, a gridlet is a package containing all the information related to a job execution, e.g.: computational requirements, inputs / outputs size or I/O operations count.

Framework was used to simulate and evaluate several scheduling algorithms in terms of computation cost and execution time for various grid configurations, budgets and deadlines. It is available under a GPL license on the project website[29], the last stable release was in November 2010.

[29] http://www.cloudbus.org/gridsim/ (accessed 10 February 2017)

3.2.2. SimGrid

SimGrid [31] is a simulation framework designed to enable evaluation of cluster, grid and P2P algorithms. The authors identify the need for a complete, standardized and robust simulation environment as an essential element for contemporary and future research in the field of distributed computing. Justification for that is the inability to provide scalable, easily configurable, reliable and reproducible environment when using hardware simulators and the abundance of self-made simulators used in experiments by different research groups. They state that the lack of one common simulation platform makes comparison of the results of various research papers significantly harder and impacts the ability to continue research started by others by reproducing the results and extending it.

The framework has a layered structure [32], on the top there is a user interface allowing to specify the application model by using one of the provided APIs, then comes the SURF layer, in which the resources (system model) are defined and on the very bottom is the layer responsible for modeling the communication network which is used to pass the data.

The User Interface layer provides four ways to model the application. *SimDag* allows to simulate applications that are defined in a form of execution flow graphs (DAG), for which the dependencies between tasks may be defined as well constraints regarding the scheduling of specific task on a specific resource. *MSG* is designed for evaluation of scheduling heuristics for CSP (Concurrent Sequential Processes), a model where multiple agents cooperate with each other to complete the specified tasks, exchange of messages is a vital part of this process. *GRAS* (Grid Reality and Simulation) makes it possible to develop a distributed application within the simulator, evaluate it and then deploy it on an actual hardware platform just by changing the library that application links against, without any changes required to the source code. The last of them, *SMPI*, is a simple way to perform a simulation using an existing MPI application, without any changes required. It works by intercepting MPI calls and directing them to the simulator.

A system model is defined as a set of resources managed by the SURF layer, the actions are issued based on the application definition coming from the GUI, the SURF is then responsible for the resource assignment and computation of the execution time, and thus completion date for each of the actions. Then, based on these completion dates, simulation time is advanced appropriately and execution control is returned to the higher layer, and so the cycle continues until there is no more actions remaining to execute and the simulation finishes. Examples of models provided for the simulation of the communication between resources are MinMax sharing strategy, analytical models for TCP congestion control algorithms and other.

A single resource, computational or communicational, has its own analytical model specified, which defines its efficiency with the possibility to take various factors into account. The models may range from very simple – considering just flops per second to model a CPU, to complex ones – network model considering RTT, TCP window size and loss factor to calculate bandwidth.

Modified version of SimGrid was used to simulate Volunteer Computing environments [33], since the VC environment may span hundreds of thousands of hosts improvements were needed in terms of simulation scalability. MSG API was used for simulation implementation and improvements targeted at reduction of complexity of crucial functions were made in SURF and LMM layers. This, in turn, allowed to remove several bottlenecks and improved the performance for VC simulation by orders of magnitude when compared to original version of the framework.

Framework is being actively developed by a large community, it is available on the project site[30] under a GPL license and was used as a foundation for nearly two hundred published articles. Last stable release was in April 2016.

3.2.3. CloudSim

CloudSim [34] is a toolkit that supports modeling and simulation of a broad range of aspects related to the cloud computing where several key factors need to be considered. First, clouds are very dynamic environments with rapidly changing supply patterns and demand for computation power varying throughout the day, with the possibility of occasional spikes occurring. The next is varying system size, services may scale on-demand and thus produce complex behavioral patterns. Furthermore, users of the cloud technologies often have competing, dynamically changing QoS requirements and applications hosted in a cloud need to take all of this into account. Benchmarking an application using varying workload, availability and reliability patterns on a real infrastructure like Amazon EC2 or Microsoft Azure, may generate significant costs, requires a lot of effort related to the environment reconfiguration for different test scenarios and makes the tests hardly reproducible. With all of these considered, the authors propose a solution allowing to run such tests in an easily configurable and controlled environment in a time effective manner. CloudSim is capable of effectively simulating a large data center on a single machine.

The toolkit consists of several layers and enables high customizability of the simulation, starting from modeling entire clouds, through service brokers, VM provisioning strategies, resource allocation policies (energy-aware as well), as far as defining capabilities of the physical hosts and the properties of the network that connects them. Network behavior is simulated using BRITE[31] topology. It is also able to simulate cloud-burst scaling strategies and federated cloud environments, which often are used to create a hybrid clouds interconnecting private and public ones.

The system is modeled by defining the number of physical hosts available, their computational capabilities, network properties, VM scheduling policies and cloud availability scenarios. Number of users, their demands, and thus the whole workload may be specified as well, there is also a possibility to create a dynamically changing workload. Application model is defined in a form of *cloudlets*. A cloudlet is an application service running on a single or multiple VMs. Most of the above is performed by extending the classes provided at the topmost layers of

[30] http://simgrid.gforge.inria.fr/ (accessed 10 February 2017)
[31] https://www.cs.bu.edu/brite/index.html (accessed 10 February 2017)

the simulator architecture. If a more in-depth analysis of the cloud behavior is needed, then the toolkit provides means to extend the core functionalities related to the VM provisioning and resource allocation, which are a part of the lower layers of simulator's architecture. Communication delay between simulation entities (hosts, storage or users) is established based on mapping of the hosts to network nodes and network topology properties defined used BRITE configuration files.

Newest version of the toolkit introduces the ability to model the containerized environments [35], it is especially important considering the rapid development of container-based technologies in contemporary cloud deployments. To achieve this a new layer was introduced on top of the CloudSim simulator core, which is also a simulation environment on its own, that handles provisioning of the containers (creation, resource allocation, assignment to the VMs) and manages their lifecycle.

The toolkit is available for download[32] under a GPL license, latest stable release was in May 2016.

3.2.4. GSSIM

GSSIM (Grid Scheduling Simulator) [36] [37] is a framework built on top of the GridSim, that aims to provide a convenient interface for describing and generating workloads, allow importing existing ones into the GridSim, share them easily with other researchers and thus make the process of reusing and comparing the results easier.

Two types of scheduling components are introduced: grid brokers (global) and resource providers (local), both can support multiple sharing strategies. To assure high flexibility and ease of use, information about the jobs is structured hierarchically. At the very top, there is a queue of jobs submitted to a grid scheduler, jobs consist of a single task or a group of tasks forming a workflow and at the bottom level there is a single task with its requirements for resources. Input data to GSSIM is composed of resource and workload descriptions, the former being an application model and the latter a system model.

Workload carries information about jobs, their resource requirements, execution dependencies and more. It is described using SWF (Standard Workload Format) or GWF (Grid Workload Format), with the extensions (detailed job definitions) stored in XML files. The framework also supports generation of workloads with a complex set of parameters.

System model consists of definitions for resource providers (autonomous systems representing a single administrative domain) and network topology, these are also stored in XML files. Resources are characterized by the number of available CPUs, memory, support for advanced reservation mechanisms, failure probability, etc. Network topology is defined in a form of existing links and their parameters. Users can write their own scheduling plugins; a plugin must implement either interface for grid level scheduler or local level scheduler. Grid schedulers are responsible for assigning jobs to the resources in different administrative domains and local schedulers are managing the jobs within a single domain, either in best-effort or QoS-based

[32] http://www.cloudbus.org/cloudsim/ (accessed 11 February 2017)

manner. Local schedulers may be structured hierarchically to mimic the structure of real environments, for example a cluster consisting of several computation nodes.

Complex application performance models may be used. One can write its own plugin used for calculation of task execution time parametrized by processor type, available memory, network parameters, task length, and more. There is also a default plugin implementing a linear relation, with regard to task size, between resource computational capability and execution time. Network flow model is used to improve the simulation speed, data transfers are modeled as entire flows with a fair sharing of bandwidth between flows using the same link. Energy efficiency may also be considered using one of the provided models: static, resource load or application specific.

Authors claim [37, pp. 8-9] that the GSSIM has been successfully used in several research projects. The project may be download from a repository[33], unfortunately project web page is no longer accessible, but according to the authors it was once available as a web application allowing to create and execute experiments as well as store them in a public repository for further use and sharing.

3.2.5. MARS

MARS (MPI Application Replay Network Simulator) [38] is a framework targeted at simulating application execution on large scale HPC clusters with a high degree of accuracy. The main point of focus is the interconnection network connecting the nodes, but the processing in the node itself is also modeled in sufficient detail. Network-related aspects of high performance computing are very important in terms of the data exchange, task placement, synchronization, etc., yet they were left on the side in favor of modeling the computational elements.

Authors identify this problem and propose a solution in a form of a framework that is built on top of a discrete, event-driven OMNET++[34] framework extended with an abstract model of computing node in place of statistical packet generators. Computing node models are driven by MPI traces, supplied in a form of files containing logs of MPI calls from real-world software. From the system model point of view, the framework offers several network topologies, configurable parameters for processors, switch and network adapter models, a rich set of routing schemes and customizable task placement. Network topology is defined in a form of OMNET++ network description.

Application is modeled using MPI trace files, OMNET++ configuration file specifies mapping of trace files to task modules, which are in turn part of the node modules representing computational nodes. A task module is responsible for reading the trace file and issuing commands based on its contents. If a computational entry is encountered, then the module computes the delay that it should apply, the delay originates from computational requirements of an action, processor speed, etc. Upon encountering communicational entry an OMNET++

[33] http://apps.man.poznan.pl/trac/gssim/browser (accessed 12 February 2017)
[34] https://omnest.com/omnest-is.php (accessed 12 February 2017)

message is formed and sent through the network. MARS also provides a convenient way to gather statistics from application execution and visualize the simulation results.

Several case studies are supplied in the paper to confirm the framework usability in terms of high-level system design, performance projection and application tuning for HPC systems. This software is a proprietary project of IBM Research Laboratory and is not publicly available.

3.3. GPU performance models

We will now proceed with the analysis of performance models targeted specifically at the GPUs, several approaches are to be distinguished in this area, three of the most common are [39]:

- analytical methods,
- quantitative and compiler-based methods,
- statistical and machine learning methods.

The first kind focuses on creation of a theoretical model for a kernel execution expressed as set of equations and feeding it with parameters obtained by direct analysis of the kernel's code and the underlying hardware, an example of such models are those from sections 3.3.1, 3.3.2 and 3.3.7. Quantitative methods on the other hand evaluate the characteristics and behavior of a given kernel either by measuring or microbenchmarking kernel execution on an actual hardware or by executing it, or parts of it, on a GPU simulator. These are often integrated into a compilation process of a kernel and are more automated than analytical ones, examples of such are: 3.3.3, 3.3.4, 3.3.5 and 3.3.6. We do not cover third type in our analysis as it is not relevant in terms of developing a model that could be used in a general-purpose simulation environment such as MERPSYS. These are often a separate framework that is highly automated and requires additional training so that the neural network correctly recognizes patterns and behavior of the kernel code. Since the aim of this work is extending MERPSYS with a performance model for a GPU, we find it entirely reasonable to omit machine learning methods, which on their own could be a focus of whole another study.

3.3.1. Analytical model based on BSP

BSP (Bulk Synchronous Parallel) [40] is a simple, general and flexible model of parallel computation, it is aimed to provide a bridging interface between a parallel software and hardware. It does so by defining the following three elements, their relations and order of execution: components performing computation and / or memory accesses, a router that delivers messages exchanged by the components and synchronization mechanisms.

The above model may be used for modeling a single kernel execution on a GPU [41], an observation is made that the kernel execution time may be modeled with an acceptable level of accuracy when we consider it as several steps taking different number of cycles to complete. These steps are: computation, global memory accesses and shared memory accesses. They are then multiplied by the number of threads launched and divided by number of cores available

on the device and their clock rate. Additionally, a λ parameter is introduced that reflects the specific characteristics of the kernel code being modeled (optimizations, memory access patterns, etc.). Global synchronization step from the BSP model is omitted as the global synchronization on the GPU occurs only after the kernel execution and the thread blocks spawned by the kernel execute independently of each other. The execution time is thus approximated by the following equation:

$$T_k = \frac{t \times (comp + comm_{GM} + comm_{SM})}{R \times P \times \lambda} \tag{3.1}$$

Where:

T_k – estimated kernel execution time

t – number of threads launched by the kernel

$comp$ – computation cost of each of the threads

$comm_{GM}$ – communication with global memory cost of each of the threads

$comm_{SM}$ – communication with the shared memory cost of each of the threads

R – core clock rate

P – number of cores available on the GPU

λ – scaling parameter

Accesses to global memory are further divided into communication with the memory itself and the L1 and L2 caches. The costs of computations and communication are determined based on the inspection of the kernel code. Arithmetic instructions and numbers of loads and stores are estimated and multiplied by the number of clock cycles required to execute them, then they are summed up to determine final values of: $comp,\ comm_{GM}, comm_{SM}$.

The model was tested on two applications: matrix multiplication and maximum subarray problem with various optimizations applied resulting in different values of λ parameter. For most of the cases the predicted execution time was within 10% error margin when compared to the measured one. Lambda values computed using a single test run were successfully used for estimating the execution time for different workload sizes and on different GPU boards. In the latter case the accuracy of the estimation dropped, although it remained at a reasonable level. This is an expected behavior considering that the GPUs vary in architecture, what directly affects the memory access latencies and computational capacities.

3.3.2. Memory parallelism aware analytical model

GPUs exhibit very high degree of parallelism, it is in fact a major factor performance-wise as the latency of a single warp caused by it waiting for a memory access may be hidden by other warps being executed at the same time, given that the sufficient number of warps is available and they are not interdependent. Thus, the effectiveness of an execution of the whole application depends mostly on hiding the latency of the memory instructions and a model is proposed [42], which focuses on estimating the number of memory requests that may be executed concurrently. This number is represented as a metric called MWP (Memory Warp Parallelism), additional metric called CWP (Computation Warp Parallelism) represents the

amount of computation that can be done when a warp waits for a memory access to finish, this metric can be perceived as an arithmetic intensity of a warp. Using these two metrics the overall cost related to the memory operations, and thus the application execution time, may be estimated.

MWP represents maximum warps per SM that can access the memory simultaneously during the period when a SM executes a memory instruction originating from one warp until all memory requests from the same warp are serviced and the SM may proceed with executing the next instruction from that warp. Hence, it is a measure of how much memory parallelism is present in the system. It is calculated based on memory bandwidth, number of memory banks, number of warps running per SM, etc. Device memory is modeled as a set of queues, with a single queue being assigned to each SM.

CWP represents the number of warps that the SM can execute during one memory warp waiting period, plus one. It is a characteristic of a specific application (kernel), its arithmetic intensity.

The model is quite complex, with 21 distinct parameters defining it and over a dozen equations used to calculate their values, naming few of them:

- number of warps per SM,
- number of threads and thread blocks,
- total execution cycles (computed based on MWP and CWP),
- dynamic number of instructions (obtained from PTX assembly and size of the input data),
- CPI (Cycles Per Instruction),
- memory access patterns (coalesced / uncoalesced),
- synchronization effects (thread synchronization in a block introduces an additional delay).

Some of those are obtained directly from source code analysis, memory model parameters are obtained by executing microbenchmarks on a real hardware, few parameters are user defined, and finally there are ones that are calculated directly in the simulation based on those parameters that are already available.

The model was tested using aforementioned microbenchmarks and an additional set of six merge benchmarks, estimation error in the former case was 5.4% and in the latter 13.3%.

Moreover, an extension called IPP (Integrated Power and Performance Model) [43] was developed. It uses the above analytical model as a data source for predicted execution time of the application and microbenchmarks to establish the parameters of the power model. IPP analyzes the instructions comprising the kernel to determine the access rates to the different hardware components of the SM (various memory units, ALU, etc.). Based on these it estimates the optimal number of active SMs that would achieve the best performance per watt ratio. The model was validated against results obtained from a hardware power meter for real application runs and the error rate for predicted power consumption was ~8.94%.

3.3.3. Work flow graph

Automated approach to performance modeling is also possible, a work by Baghsorkhi et al. [44] presents a concept and implementation of a work flow graph based description of a kernel. In this approach kernel's code is fed directly into the compiler, which disassembles it and identifies how it makes use of the most important features of a GPU microarchitecture, then a directed graph reflecting the kernel structure is created and used to estimate its execution time.

The key factor considered in this model is a warp level parallelism (WLP), which is a measure of the application's capability to hide a single warp latency caused mostly by memory stalls, or in other words, how effective the SM may perform the work assigned to it. If memory bandwidth is not fully saturated then for as long as there are free warps available, a SM can postpone the execution of a stalled warp and start executing a new one while waiting for the memory access from the first one to finish. Intra warp parallelism (instruction level parallelism), although less important, is considered as well. It is a measure of the ability to deal with stalls caused inside a warp by instructions waiting for memory, register dependencies, etc.

Work flow graph represents an average warp and is an extension of a control flow graph of a GPU kernel, with nodes being the instructions comprising the kernel: global / shared memory accesses, computational, barriers, and the edges defining the order in which instructions are executed. This order considers not only the code structure but the data dependencies as well. At the beginning the are no weights assigned to the edges, only the precedence relation of the instructions is known, but not the actual transition cost. This means that the initial WFG is an abstract representation of a kernel not connected to any specific hardware piece. Initial WFG is constructed by the provided compiler, it analyzes the kernel source code and translates it into program dependence graph (PDG) representation, then preorder and postorder walks are performed on the PDG, this is done to extract the code characteristic allowing to compute the values of WLP and ILP.

Next step is a specialization of the WFG to fit a particular GPU, for this purpose a symbolic evaluation of conditional branches and memory accesses based on actual hardware parameters (memory: bandwidth, banks count, r/w latencies, coalescing rules; warp size; SIMD engine width; etc.) is performed and weights are assigned to the edges of the graph. Those parameters may be either read from a user input (simulation description file) or directly from the hardware by running microbenchmarks. As the conditional branches may introduce execution path divergence the edges in a graph are weighted appropriately to reflect this.

The WFG based model was evaluated using three sets of benchmarks: dense matrix multiplication and FFT, prefix sum scan, and sparse matrix-vector multiplication. Their purpose was to verify the model accuracy in terms of the impact on the overall performance, respectively for: memory bandwidth and latency, SIMD divergence and bank conflicts, and data dependent conditions and irregular memory access patterns. These benchmarks were executed for different optimizations present and different launch configurations (number of threads per

block), in every case the estimated execution times were within acceptable accuracy bounds when compared to the measures taken during actual test runs.

3.3.4. Microbenchmark based

Another interesting approach is the one based on a direct analysis of a GPU native instruction set extracted from compiled kernel's code, supplemented by microbenchmarks [45]. Although this model does not directly focus on modeling and prediction of application's execution time but instead on identifying performance bottlenecks, it is worth mentioning here as a look from another perspective on the GPU performance modeling problem. It focuses on three major components of the overall execution time: instruction pipeline, accesses to shared memory and accesses to global memory, a description of modeling process for each of these will follow.

The data being fed to the later stages is obtained using functional simulator Barra, this includes: number and type of dynamic instructions, number of shared and global memory transactions and the synchronization points.

Instruction pipeline modeling works by assigning instructions to different groups based on their computational cost, i.e. number of functional units able to execute them that are available on the SM and required clock cycles. A microbenchmark is then run for different amounts of warp level parallelism and the pipeline throughput is estimated. GPUs are constructed in a way that allows them to hide pipeline stalls by executing many threads grouped into warps in an interleaved fashion (if one warp is blocked then another takes its place in the execution unit), the capability to perform this effectively is governed by the number of parallel warps. Block configuration affects this number directly so the instruction throughput is measured for different configurations and based on this the instruction pipeline throughput is computed.

For shared memory accesses, bandwidth at the corresponding amount of warp level parallelism is taken, the better it is the higher will be the saturation of the shared memory. Then the effective number of memory transaction is estimated by specifying the degree of bank conflicts and adjusting the total number of memory transactions obtained from the program statistics.

Global memory access time is modeled using a memory transactions simulator, that computes the number of memory accesses that would occur on the hardware level. Three major factors play an important role here: number of blocks, number of threads per block and the number of memory transactions per thread. The simulator also considers the fact that the memory accesses may be coalesced or not.

Based on these three components the final performance model is created, which then outputs the code characteristics: expected execution time, identified bottlenecks, throughput for instructions and memory, efficiency of memory accesses, etc.

The model was evaluated using dense matrix multiplication, tridiagonal systems solver and sparse matrix-vector multiplication to verify the accuracy of the modeling for each of the three major components, since these kernels are respectively: instruction pipeline, shared

memory and global memory bound. Simulation accuracy for various workload sizes was within 5% to 15% and furthermore the process of successfully locating the bottlenecks in these kernels' code and improving their performance was shown.

3.3.5. Ocelot framework

Precise modeling of the kernel execution is also possible by the means of reading the PTX assembly, translating it into a representation suitable for modeling on a single CPU thread, launching such simulation and gathering execution metrics. This approach was proposed by Kerr, et al. [46] [47], it facilitates Ocelot dynamic compilation framework[35] as an entry point which feeds the PTX directly into a complex functional emulator for GPU ISA paired with a dedicated runtime framework and analysis modules. This is an effective approach that allows for a very in-depth analysis of programs written in CUDA.

The emulator can execute kernels compiled into a PTX assembly, assembly file is parsed by it and analyzed to produce a control flow graph representation. The parser analyzes the memory requirements of statically declared variables in the kernel's code and converts infinite size register files used by PTX to a well-defined one. Then the memory is allocated on the host, on which the simulation is launched. Kernel execution is handled by the emulator by executing CTAs (Cooperative Thread Arrays) one at a time and one after another. Translation flow is as follows:

1. CUDA kernel is compiled into a PTX which is then translated into a LLVM representation,
2. highly parallel execution model is transformed and adjusted so that its semantics change and fit a single thread of a CPU and thus can be launched as such,
3. instructions defined by PTX that require special handling (like atomic memory accesses or texture sampling) are translated into an equivalent representation.

This model was successfully used to analyze over 50 GPU kernels coming from four well known suites: CUDA SDK, UIUC Parboil, VSIPL API, RIAA. The results were then validated and evaluated by comparing them to the metrics obtained from test runs on an actual hardware. The emulator allows to gather several metrics related to the kernel execution and it is also possible that the user defines custom metric-gathering event handlers. The analysis for each of the kernels focused on a few crucial ones:

- activity factor (average number of threads active at a given time, the higher the number of divergent branches, the lower this metric will be),
- branch divergence,
- memory usage efficiency (global, shared, texture, constant),
- data flow and data sharing using shared memory,
- MIMD parallelism (the theoretical parallelism level that may be achieved by the application if it is given an infinite number of SMs and not constrained by memory bandwidth or latency in any way),

[35] http://gpuocelot.gatech.edu/ (accessed 20 February 2017)

- SIMD parallelism (the inter-CTA parallelism).

Based on these, the most common problems are highlighted and few recommendations are proposed on how to improve the performance of the analyzed kernels.

3.3.6. Interval analysis

Interval analysis is an alternative to the traditional analytical models, instead of using the information describing the natural flow of the events in the code being analyzed (sequence of computation and communication) and summing them up to determine the resulting cost (computation time), it focuses on locating the events which cause the performance degradation like memory stalls, cache misses, etc. This information is then used to estimate the performance impact of each of these events and penalize the ideal model of the execution, i.e. the one in which those events would not occur and all instructions would execute without any disturbance.

GPUMech [48] works by profiling the instruction trace of every GPU warp and then running a clustering algorithm against them to identify a representative warp, i.e. the one which best resembles the execution characteristic of all warps. This is especially important if the analyzed kernel exhibits high control-flow divergence, because in such case random pick may result in a warp being chosen that represents not-so-common execution path and the whole simulation being inaccurate. The traces are obtained using Ocelot framework described earlier. Once selected, the representative warp is used to create a multi-warp model that is a base for estimation of the kernel performance and generation of CPI stacks. The decision to use a single representative warp is governed by the fact that precise modeling of each, or at least significant number of warps, although leading to much better and more accurate results, would result in an unacceptable simulation running time and thus is not applicable in practice.

Interval algorithm uses as its input the instruction latency for every instruction of the kernel and a complete instruction trace of a warp that carries the dependency information, based on these two the interval profile is constructed for each of the warps. Additionally, a cache simulator is used to model cache misses impact on the warp performance.

K-means clustering algorithm[36] is used to select the representative warp that will be in the very center of the cluster spanning the warps with similar interval profiles. Overall, two clusters are used: one to select the significant (similar warps) and the other one to discard those of much lesser significance. The input to the algorithm - feature vector for each of the warps consists of warp performance, given it terms of IPC, and the number of the instructions executed.

The multithreaded model assumes at first that all warps are running on the SM in perfect conditions, i.e. without any resource contention. This is done to determine the number of the instructions of other, non-representative warps, that will be available to hide the stall cycles that occur in the representative warp. In an ideal case, all the stalls would be covered by the instructions from other warps, but most often it does not happen and results in a performance degradation. In the next step, resource contention resulting from cache misses, uncoalesced

[36] http://stanford.edu/~cpiech/cs221/handouts/kmeans.html (accessed 1 September 2017)

memory accesses and global memory bandwidth is added to the model and the result is produced.

The model was evaluated using numerous kernels and compared to detailed GPU simulators. Its usefulness was proven by achieving acceptable error rate of ~14% and a significant improvement in simulation time when compared to a detailed simulator.

3.3.7. Latency and throughput bounds aware analytical model

The highly parallel architecture and processing model of the GPU makes it not feasible to express the overall computation time as a sum of processing times of the individual instructions as these may overlap. This problem is addressed in a recent work by Volkov [49] where the total execution time of a kernel on a GPU is described in terms of latency, throughput and concurrency.

Execution of a kernel is modeled at a warp level in scope of a single streaming multiprocessor. The model focuses on obtaining a key metric describing the parallel process – a warp throughput, from which the total kernel execution time can be derived. Given Little's law[37], the latency, throughput and concurrency may be combined using equation 3.2. Concurrency in this case is synonymous to occupancy, i.e. the number of active warps residing at a SM at a given time. Furthermore, an assumption is made that both the occupancy and the throughput are sustained for the entire time of the kernel's execution, it is later proven to be valid by the author. An important part of the model are the bounds for warp throughput and latency, as shown in equation 3.3, warp throughput is limited by throughput bound imposed by the hardware and its latency may not be lower than the one resulting from the instruction sequence comprising the kernel.

Warp latency bound is obtained by direct inspection of the compiled kernel assembly code. A kernel execution graph is constructed with nodes representing the instructions and edges representing the latencies between them, latencies from instruction issue, register dependencies, and memory accesses are considered. A chain of dependent instructions with the biggest sum of latencies is found in the graph and is then used as a mean latency of all warps. This assumption is also valid because the kernels analyzed by the author are not highly divergent, i.e. the execution paths for all threads are the same in clear majority of the cases.

Warp throughput bound is calculated based on requirements for resources available on the SM. Each SM consists of different functional units: CUDA cores, SFUs, schedulers, and load / store units. These resources are utilized at a different ratio depending on the instructions composition of the kernel. To obtain the throughput bound each of these units is considered separately in terms of requirements for its processing power and then the tightest bound is used as a warp throughput bound.

The resulting model is denoted by equation 3.4, that is obtained by substituting mean warp latency from equation 3.3 with value obtained from equation 3.2 and then transforming the result to a final form. An important trait of this model is that is has two modes reflecting the

[37] http://web.mit.edu/sgraves/www/papers/Little%27s%20Law-Published.pdf (accessed 16 May 2017)

characteristics of a kernel, which can be either latency or throughput bound. In former case the throughput increases linearly with the occupancy and is dependent on the SM latencies and in the latter, it is fixed and dependent on the SM throughput limits. This two modes behavior is shown in figure 3.4 from Volkov's work [49].

$$\overline{occupancy} = \overline{warp\ latency} \times warp\ throughput \tag{3.2}$$

$$warp\ throughput \leq throughput\ bound \wedge \overline{warp\ latency} \geq latency\ bound \tag{3.3}$$

$$warp\ throughput \approx \min\left(\frac{occupancy}{latency\ \ bound}, throughput\ bound\right) \tag{3.4}$$

Where:

$\overline{occupancy}$ – mean number of warps processed in parallel at a given point in time

$\overline{warp\ latency}$ – mean latency (processing time) of a single warp

$warp\ throughput$ – processing rate of warps

$throughput\ bound$ – upper bound on warp throughput imposed by the SM throughput limits

$latency\ bound$ – lower bound on warp latency imposed by the SM latencies

In the aforementioned figure, we can also see that the model is accurate for low and high occupancies but loses its accuracy in between, this is due to gradual saturation effect, which was also identified by the author. This effect causes the resources to not be saturated linearly, with the increase of contention in accessing a resource its efficiency drops and thus more concurrent accesses are required for achieving peak utilization than would otherwise be needed if the characteristic was linear. The most prominent example for this is the behavior of internal memory bus of the GPU, hence the author proposes a more complex approach to modeling a global memory throughput, which employs a parametrized memory access model.

In his study Volkov also points out a common misconception that the increase of occupancy always results in a better utilization of GPU resources as it hides latency and thus is essential for achieving peak throughput. As we have already presented, the occupancy at which the peak throughput is attained is solely dependent on kernel characteristics and the hardware being used. It should be also noted that the author uses a very accurate testbed and testing methodology to extract the device characteristics (latencies, throughputs) and proves that the obtained results are better than those of other well-known studies in this area, including the ones presented earlier in this chapter. He also reproduces the results of those studies showing where they fall short when compared to his model.

To further back up the validity of his claims we ought to bring up another study of existing GPU performance models [39] dated April 2016 (Volkov's work is from August 2016). In this study a thorough analysis of the existing models is performed, that covers most of these mentioned in Volkov's work as well. It identifies several problems with the existing models and concludes that there is no outstanding model in this area yet, as the existing ones lack sufficient accuracy and simplicity to become one. As we will further show in this work, the model proposed by Volkov has the potential to supersede all of its predecessors, being both accurate and reasonably simple.

This page intentionally left blank.

CHAPTER 4. PROPOSED SOLUTION

4.1. Introduction

Having analyzed the related work, we now proceed with a proposition of a solution for modeling parallel processing on a GPU that will be implemented for the MERPSYS platform. Model proposed by Volkov was intentionally described at the very end of the previous chapter because we have chosen it as a base of our further research. We will adjust and modify it per our needs, with the details given in section 4.2, but the core concepts remain the same and thus it should be underlined that what we will develop is a practical implementation of this model. We have selected it because of its accuracy, flexibility and fitness for implementation in MERPSYS, what we mean by that is that we can easily distinguish in it the elements that should be a part of application, hardware and computational models.

In the next section, we explicitly state our contribution to the model. Next, we analyze the requirements that should be met by our solution, what follows is a detailed definition of a theoretical model for a kernel execution with all the required equations and parameters. Lastly, we describe our solution in terms of MERPSYS' application and hardware models and provide a general overview of how it will work.

4.2. Contributions of this work

This work makes following contributions. First, we incorporate the model proposed by Volkov [49] within the MERPSYS simulator [2], showing that it is well suited for practical implementations. We have used a simplified version of this model by reducing the scope of hardware units considered to CUDA cores, schedulers, and global memory system, for which we decided to use a trivial non-parametrized access model. For this simplified version, we provide a complete set of equations (4.1 through 4.6) together with a detailed description of input parameters, creating an easy to grasp explanation of all the building blocks of the model. We also present parts of the model, for which in the original work only a textual description was given, in a formalized form using mathematical equations. We also extend the model with a scaling parameter, allowing to fine-tune it to better fit hardware and kernel characteristics. We propose an equation for calculating global memory bandwidth (4.7) and introduce data transfer time calculation, whilst the original model considered only kernel execution time. What is more, we show how occupancy value could be obtained, in the original model it was assumed to be known. Lastly, we have gathered measurements of instructions latencies for various architectures from several other research papers and presented them in a tabular form. These latencies are a crucial component of the model and are needed to construct the kernel execution graph.

4.3. Requirements specification and business modeling

There are two major sources of requirements for this work, first group is related to the GPU modeling itself whist the second one concerns the MERPSYS platform. To better

understand the selection of the model that we have made, the latter group will be stated and evaluated first.

One of the major factors influencing the choice of this thesis' topic is that MERPSYS so far has not been validated and extensively used for a detailed GPU based simulation. Although the GPUs were a part of various simulations as computational components, in all cases the focus was on analyzing the behavior of MPI based distributed applications. This approach used metrics like the power consumption and abstract processing power, hence it operated at a very high level of abstraction and skipped any architectural details of a device. The purpose of this work is to identify a detailed GPU performance model that can be implemented in MERPSYS and integrate it within the existing components of the simulation environment. This should not only allow the user to perform a more detailed and accurate simulations targeted specifically at the GPU for simple, standalone CUDA applications, but also improve the accuracy of scenarios with complex, distributed programs where a CUDA application is just a single part of a larger whole. There also should be a possibility for the user to parametrize the application using the existing MERPSYS' simulation launch screen so that the inputs like data size or grid configuration can be easily changed to evaluate various configurations. Considering this fact, it is crucial that the new solution fits the current simulation processing model of MERPSYS and is as easy to use as possible.

Next goal to achieve is the ability to execute a single application model against many different GPUs by simply changing the device in the system model to another one, the same way as it is currently done in MERPSYS. This raises a need for new hardware parameters being placed in the definition of a hardware unit in MERPSYS, since these are applied to the application model during the execution of a simulation and thus the result depends on their values, without requiring any changes of the application model itself. These use cases for our solution are summarized on Figure 4.1, for a use case diagram of MERPSYS see [26, p. 4].

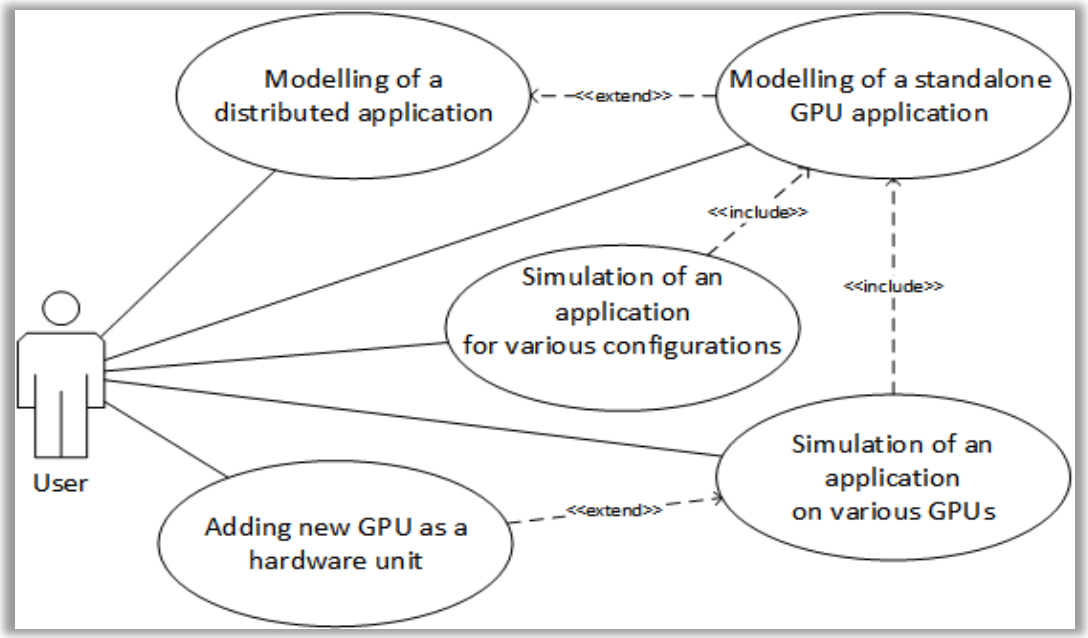

Figure 4.1. Use case diagram

Consequently, during the implementation we will evaluate the flexibility and extensibility of the MERPSYS platform and verify whether its design allows for an easy introduction of a new type of simulation models, in this case a detailed model for GPU simulation. This work should also provide a systematic walkthrough for modeling CUDA based applications in MERPSYS, i.e. the creation of an application model. As a part of this, a sample simulation should be prepared and its accuracy verified. Finally, because of this work MERPSYS shall offer a better, in-depth and more accurate approach to GPU modeling.

Back to the GPU modeling, the model of our choice should cover at least the basic characteristics of a kernel, namely arithmetic and control flow instructions, and accesses to global memory, since we narrow our analysis to these as shown in section 4.4.3. However, if it could address more elements like shared memory or synchronization primitives, then obviously, it would be an even better choice. Even though in this work we focus entirely on NVIDIA GPUs and CUDA architecture, the model should be universal and adaptable so that it can be applied not only to the kernels written in CUDA but in OpenCL as well. Furthermore, it should also be possible to use it for AMD GPUs. This will allow for further extension and generalization of the solution once it is implemented.

Considering all mentioned requirements, it was decided that the model proposed by Volkov, that was described in section 3.3.7 will be used as a base for our solution. It is both simple and accurate, handles all important building blocks of a kernel, offers a clear decomposition of application and hardware models, and could be used with both the OpenCL and AMD GPUs[38].

4.4. Theoretical model for kernel execution time

To model the execution time of a single kernel on a single GPU we will rely on a model described in section 3.3.7, we define equation 4.1 for this purpose. The number of warps launched is computed as shown in equation 4.2. For warp throughput, we use equation 3.4 presented here for the sake of brevity as equation 4.3, the methodology for obtaining occupancy, latency bound, and throughput bound will follow. Although being relatively simple this model considers all of the important elements that contribute to the overall execution time of a kernel in most of the scenarios. For each of these elements we will describe their meaning and the approach used to determine their values.

Since the entire model is based on a concept of modeling a concurrent process in terms of a warp being executed on a SM and the metrics are calculated for that single warp, it is required to extrapolate the findings to reflect the actual number of warps that were executed on the GPU and thus compute the result. For this purpose, equation 4.2 is used that takes as an input the launch configuration of a kernel, precisely speaking the sizes of grid and block, and the result is the total number of warps that were launched to process the kernel. At the time of writing warp size is constant and equals 32 for all GPU architectures.

[38] It would require a change in the approach to obtaining application and hardware parameters, but we will not elaborate on this now as it is not in the scope of this work.

$$T_k = \frac{warps\ launched}{warp\ throughput \times SMs \times SM_{clock} \times \lambda} \tag{4.1}$$

$$warps\ launched = grid\ size \times \left\lceil \frac{block\ size}{warp\ size} \right\rceil \tag{4.2}$$

$$warp\ throughput\ \approx \min\left(\frac{occupancy}{latency\ \ bound}, throughput\ bound\right) \tag{4.3}$$

Where:

T_k – estimated kernel execution time [s]

$warps\ launched$ – number of warps launched

$warp\ throughput$ – processing rate of warps [1 / cycle]

SMs – number of streaming multiprocessors

SM_{clock} – SM core clock [cycles / s]

λ – scaling parameter

$grid\ size$ – number of blocks in a grid

$block\ size$ – number of threads in a block

SMs and SM_{clock} parameters are self-explanatory, similarly as above these are needed to extrapolate the model from a single warp in scope of a single multiprocessor to one that fits the processing model of the GPU. These two parameters when multiplied yield the processing power of the entire device in cycles per second.

The λ value is a ratio between estimated and measured execution times that adjusts the model to fit the characteristics of a given kernel launched on a specific device architecture. It shall be obtained experimentally by measuring the execution time of a real application combined with running a simulation with an initial model. Once calculated the new λ can be used to perform simulation across varying data sizes and different GPUs. It was experimentally proven by the authors of another model [41] [50] that a single lambda value is sufficient for accurate simulations in scope of a single device architecture. When used for a different architecture the accuracy of the model drops and it is advised that λ is recalculated.

$$\lambda = \frac{t_{estimated}}{t_{measured}} \tag{4.4}$$

Where:

λ – scaling parameter

$t_{estimated}$ – execution time obtained from simulation with $\lambda = 1$

$t_{measured}$ – measured execution time of the CUDA application being modeled

Calculation of warp throughput, the most essential part of the model, as given by equation 4.3 is the most challenging task. Three parameters need to be extracted from the kernel code in close consideration of a specific GPU architecture that one intends to model. These parameters are warp latency and throughput bounds, and the achieved occupancy, we will describe them in detail in subsequent sections.

4.4.1. Occupancy

To obtain the value of the occupancy we will rely on CUDA Occupancy Calculator [51], we start by extracting requirements for shared memory needed by each of the thread blocks and number of registers needed by each of the threads. There are at least three ways to determine those, each of them having its benefits and drawbacks:

- direct inspection of the code – prone to an error and time consuming but does not require any external tools,
- compilation with verbose PTX assembly output[39] - fast but requires the application to be recompiled,
- inspection using the profiler – fastest, completely accurate, but requires the profiler to be available. It is also worth mentioning that when using the profiler the occupancy value can be read directly from it. The profiler also shows a mean value of a real occupancy that was achieved across the entire execution of a kernel.

Having obtained these values, we enter them in the calculator together with the desired block size, compute capability of the target device, shared memory configuration, and caches configuration. Figure 4.2 shows an example of that.

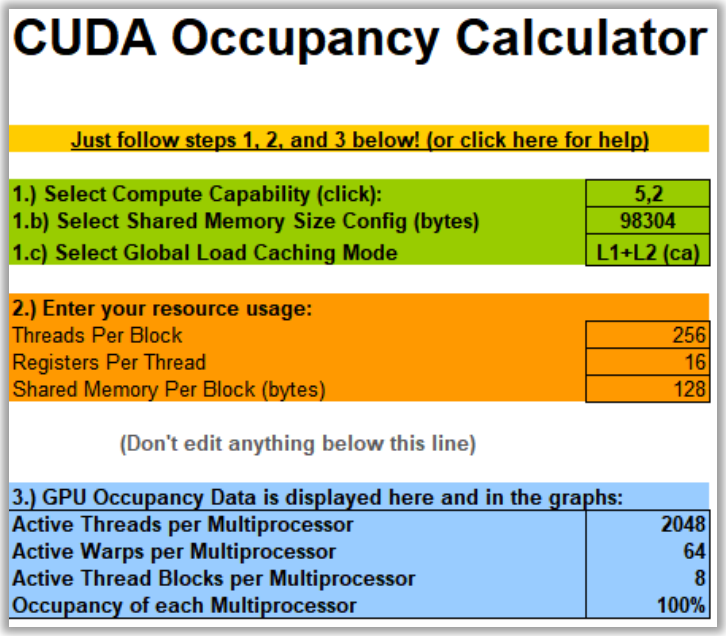

Figure 4.2. Screenshot from CUDA occupancy calculator

Introduction of the occupancy calculation directly to our model is a potential future work and would likely be a simplified and approximate version of what is offered by NVIDIA's occupancy calculator, as we would not consider the effect of allocation granularities for registers, shared memory and warps. We would also need to determine the hardware limits imposed by the GPU architecture and have them included in the hardware model, for this purpose Table 2.2 could be used.

[39] http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#ptxas-options (accessed 10 June 2017)

4.4.2. Latency bound

We construct an execution graph of the kernel with nodes representing instructions and edges representing latencies as shown in section 3.8 of Volkov's work [49] and then apply critical path method to it to find the latency. Critical path is the latency bound only if we assume that all of the warps are the same – they execute the very same instructions, so this approach will not work if there is a substantial control flow divergence in a kernel, in such case, the different set of instructions resulting from different branches being executed should be considered. The graph considers latencies of different origin: register dependencies and thus instruction execution latency, instruction issue latency, and memory stalls.

This approach is valid because there is no out-of-order execution present in the GPUs and is more precise than when making an assumption that a kernel is a simple sequence of dependent instructions[40] as in [41], [52] or [53] and simply summing the latencies of all instructions comprising the sequence, treating the result as a total cost in clock cycles of executing this kernel by a single warp. The latter approaches fall short because there are independent functional units within the SM, what results in latency hiding for example in case of waiting for memory accesses. Furthermore, even in case of a single functional unit type the instructions are processed in a pipelined-based manner and thus ILP exists.

Volkov reports in his work [49, pp. 30-34] that there are two other types of latency, the latency between two independent instructions from a single warp, called "ILP latency" and latency resulting from a replacement of a terminated thread block, we address both of these. What is more, depending on the architecture multiple warp schedulers per SM may be present and two instructions per wrap may be issued every instruction issue time if mutually independent instructions are available for execution[41], the effect of dual issue is considered as well, in which case the ILP latency is expressed as 0 cycles.

We omit double precision and special function units, both of which have their own pipelines with different latencies for various instructions. Moreover, our approach to modeling global memory accesses is simplified as we do not consider gradual saturation effect (see sections 6.3 - 6.5 [49]), which manifests itself in a form of increasing memory access latency as the number of memory transaction increases and memory bus gets saturated.

The very heart of the model are computational and communicational costs, those shall be obtained by a direct inspection of the kernel code. Each of the instructions issued by a thread is assumed to require a well-known number of clock cycles to complete, this is a simplified approach not considering varying memory access times, varying instruction latency depending on the instruction sequence and other effects, but should be accurate enough. Based on research by Andersch et al. [54] and Volkov [49] two tables are presented. Table 4.1 lists latencies for few common instructions and Table 4.2 contains memory access latencies.

[40] i.e. an instruction can only be issued once its predecessor in the sequence has finished executing because of back-to-back register dependencies.
[41] See chapter G of *CUDA C Programming Guide* [8].

For a detailed information regarding meaning of the mnemonics refer to chapter 4 of *CUDA Binary Utilities* [55].

Table 4.1. Instructions latencies across subsequent generations of GPUs

Operation	Tesla GT200	Fermi GF106	Kepler GK104	Maxwell GM107
add, sub (integer)	24	16	9	6
mad (integer)	120	22	9	13
mul (integer)	96	20	9	13
div (signed)	684	322	168	243
rem (signed)	784	315	163	232
and, or, xor	24	16	9	6
shl, shr	24	18	9	6
add, sub (32-bit FP)	24	16	9	6
mad (32-bit FP)	24	18	9	6
mul (32-bit FP)	24	16	9	6
div (32-bit FP)	137[42]	1038	758	374
mov	24	16	9	6
setp	24	16	9	6
bra (taken)	N/A	32	N/A	12
bra (not taken)	N/A	28	N/A	10
ILP latency	N/A	N/A	3	3
Terminated block replacement	N/A	N/A	201	150

Table 4.2. Memory access latencies across subsequent generations of GPUs

Memory unit	Tesla GT200	Fermi GF106	Kepler GK104	Maxwell GM107
Global	440	685	300	350
Global L2 cache	N/A	310	175	194
Global L1 cache	N/A	45	30	N/A
Shared	38	50	33	28

Although the latencies for *mov* and *setp* instructions are not explicitly stated in the mentioned research papers we assume them to be taking the same amount of time as other basic operations (*add*, *and*, etc.) as they are also executed by CUDA cores and thus share the same pipeline latency [49, p. 33].

Latencies of branch instructions are not provided for Maxwell architecture, on which this work will be based, so given the observation that for Kepler it equals double the latency of CUDA core instruction we assume this relation to hold for Maxwell too, what gives us 12 cycles latency for a taken branch.

Furthermore, in his work Volkov provides the value of ILP latency only for Kepler architecture, for which it equals 3 cycles. Since we need this value for Maxwell as well, we

[42] Although this result may look strange, in the original study [59] authors made a following statement: "However, single-precision floating point division is translated to a short inlined sequence of instructions with much lower latency.". It also does not mean that the Tesla architecture had higher performance for single precision div that its successors – instruction latency must not be confused with throughput.

assume that it also equals 3 cycles. It is worth pointing out that even if this latency is a bit different for Maxwell, e.g. equals 2 or 4 cycles, it will not significantly affect the accuracy of the simulation. This is because when selecting a critical path in the kernel execution graph an edge labelled with ILP latency is seldom taken because the instructions in the kernel are very often dependent on each other, this is visible in Figure 5.6.

4.4.3. Throughput bound

To obtain the throughput bound each of the hardware resources available on the SM must be considered separately, these are: CUDA cores, SFUs, double precision units, schedulers, and global and shared memories. We will narrow our analysis to three of them as denoted by equation 4.5. The tightest bound, that is the highest number of cycles required to execute warp's instructions due to a limited processing power of the functional units is selected as the limiter of the performance and then a reciprocal is calculated to obtain the bound represented in warps per cycle[43].

$$CPW_{bound} \approx \max\left(\frac{warp\ size \times INS_{CUDA}}{CUDA\ cores}, \frac{INS_{issued}}{schedulers}, \frac{GMem_{bytes}}{GMem_{bandwidth}}\right) \tag{4.5}$$

$$throughput\ bound = \frac{1}{CPW_{bound}} \tag{4.6}$$

Where:

CPW_{bound} – cycles per warp bound [cycles]

INS_{CUDA} – number of instructions to be executed by CUDA cores

$CUDA\ cores$ – number of CUDA cores available on the SM

INS_{issued} – total number of instructions issued (including dual-issues and re-issues)[44]

$schedulers$ – number of schedulers available on the SM

$GMem_{bytes}$ – number of bytes transferred for each warp [bytes]

$GMem_{bandwidth}$ – peak throughput of the memory system per SM [bytes / cycle]

$throughput\ bound$ – upper bound on the warp's throughput [1 / cycle]

There are two sets of parameters present in the equation 4.5. The first one is being related directly to the hardware characteristics of a given device, it includes: warp size, number of CUDA cores and warp schedulers, and the throughput of the global memory. At the time of writing warp size is a constant value of 32 for all GPU architectures. Values for the next two parameters depending on the GPU architecture are listed in Table 2.1.

Theoretical bandwidth of the GPU's global memory available per single SM every cycle is given by equation 4.7, for which all the parameters are read directly from GPU hardware specification except the *data rate* that is derived from memory type and equals 2 or 4

[43] Warps per cycle (WPC) metric is an average number of warps executed every clock cycle and is akin to IPC with the difference being that the GPU processes instructions at a granularity of a warp, not a single thread. To better understand the WPC vs CPW relation one can refer to explanations of IPC vs CPI.

[44] Single scheduler performs one instruction issue per SM clock cycle. For example, if we have a total of 40 instructions to issue and 4 schedulers available then 10 clock cycles would be needed. This is if every instruction is issued in a regular manner and only once, dual-issues and re-issues are covered at a later point in this section.

respectively for GDDR3 and GDDR5 [56, p. 6]. Although the peak bandwidth is not sustainable in practice for the entire execution of a kernel because the memory clock may drop due to dynamic frequency scaling and furthermore would require an ideal access pattern and sufficient occupancy, a value close to this theoretical limit is attainable as reported by Volkov in section 6.3 of [49]. Considering this fact, we will use this bandwidth in our model.

$$GMem_{bandwidth} = \frac{GMem_{clock} \times \frac{bus\ width}{8} \times data\ rate}{SMs \times SM_{clock}} \tag{4.7}$$

Where:

$GMem_{bandwidth}$ – global memory bandwidth [bytes / cycle]

$GMem_{clock}$ – global memory clock [Hz]

$bus\ width$ – global memory bus width [bits]

$data\ rate$ – data rate multiplier depending on the kind of memory used

SMs – number of SMs

SM_{clock} – SM clock [cycles / s]

What is more, we do not differentiate between loads and stores and assume them to behave in the same way. Additionally, the model is applicable only to a very specific scenario where there are no cache hits and the accesses are fully coalesced. If we were to include the possibility of cached accesses in our model then we would need to consider the fact that depending on the device architecture accesses to global memory are cached in L1 and L2 caches, hence the number of cache hits should be subtracted from the actual number of DRAM accesses. The ratio of cache hits could be obtained by launching the application under the profiler [52, p. 10].

The second set of parameters: INS_{CUDA}, INS_{issued} and $GMem_{bytes}$ describes application characteristics, all three are extracted directly from compiled SASS code of the kernel. Calculating number of CUDA core instructions is very straightforward, we simply count all occurrences of operations that are executed using these cores: FP32, INT32, logical, etc. Getting a sum of all instructions that were issued is a little bit trickier because dual-issues and re-issues of instructions need to be included. In former case, two instructions are issued in a single issue cycle and hence are counted as one. In the latter, a single instruction must be counted multiple times depending on the number of re-issues, a re-issue typically occurs when there are bank conflicts when accessing shared memory or when an access to global memory is uncoalesced and must be split into several separate transactions. This effect also affects number of bytes transferred per warp, for example when fetching a single 32-bit value per thread, a stride of two would result in 256 bytes transferred instead of 128 and a single additional re-issue of memory transaction instruction, stride of 4 would cause 3 additional re-issues and increase the number of bytes to be transferred to 512.

4.5. Estimating data transfer time

Performing computations on a GPU requires the data to be transferred to the device prior to launching a kernel and once its executions completes the results must be fetched back

to the main operating memory. Memory transfer between the host and the device may be considered an instance of a point-to-point communication, for which the communication time is given by a following equation [26, p. 40]:

$$T = t_s + \frac{n}{bw} \tag{4.8}$$

Where:

T – data transfer time

t_s – startup time

n – data size

bw – bandwidth

A startup time is the time needed to initiate data transfer that depends on latency and runtime overhead, it can be measured by sending a very small data packet, e.g. a few bytes as will be further shown in section 5.4.2. Bandwidth is dependent on the configuration of the PCIe bus that connects the devices, for example a theoretical bandwidth of PCIe v3.0 16x bus in single direction is 16GB/s or 15.8 GB/s if we consider 128b/130b encoding[45]. This value however does not consider a communication protocol overhead and other factors that may limit the throughput. Table 4.3 down below summarizes the values of a maximum theoretical data throughput for various PCIe configurations. Version 4.0, that was announced recently on 8th of June[46] was intentionally omitted as it is not yet implemented in the hardware being sold, the latest version available on the market is 3.1.

Table 4.3. Maximum throughput for different PCIe configurations

	Throughput [GB/s]			
version / lanes	x1	x4	x8	x16
1.x	0.25	1	2	4
2.x	0.5	2	4	8
3.x	0.985	3.94	7.9	15.8

4.6. Fitting the model into MERPSYS

For the solution to work several parameters must be defined in MERPSYS' application model so that they can be used in combination with the parameters defined in the hardware model to calculate the running time of a kernel and data transfer times when launching a simulation.

First batch of the parameters are those specifying the characteristics of a single warp: INS_{CUDA}, INS_{issued}, $GMem_{bytes}$ and $latency\ bound$. Next, we have kernel launch configuration, i.e. the number of blocks in a grid and a size of a single block, these need to be specified separately so that the block size value can be used to calculate number of warps launched. The additional benefit of such separation, in contrast to directly passing the total number of threads,

[45] http://pcisig.com/faq?keys=How+does+the+PCIe+3.0+8GT%2Fs+%22double%22+the+PCIe+2.0+5GT%2Fs+bit+rate%3F (accessed 30 June 2017)

[46] https://techreport.com/news/32064/pcie-4-0-specification-finally-out-with-16-gt-s-on-tap (accessed 30 June 2017)

is that it directly resembles the model used in CUDA where we specify grid and block dimensions separately. Occupancy and λ parameters are also a part of the application model. Finally, the model must also contain a parameter related to the input data size, which is needed for calculation of the data transfer times in both directions.

When referring to the hardware model, we will consider three elements: hardware units specification (GPU and interconnect), computational model (functions describing hardware behavior) and system model (devices paired with interconnect defined using the editor application). The first part of the hardware model are the parameters describing the device and interconnect, for GPU these are: number of SMs, SM core clock, number of CUDA cores and schedulers per SM, global memory throughput per SM, and warp size. In case of the interconnect we need its bandwidth and startup times for both transfer directions. Both types of hardware units must be added to the MERPSYS' components database with mentioned parameters defined in their specification as attributes. The second part, the computational model, are the functions used to compute the execution time of the kernel and data transfer times, these are in fact an implementation of the equations from previous sections, that constitute the application performance model. The third part is an abstract representation of a testbed used to perform the simulation – GPU, interconnect, and the CPU. It will be defined in MERPSYS' system model editor and its purpose is to tie together all of the previously mentioned elements, i.e. hardware units, computational model, and application model (via *labels*). The technical details of the whole implementation will be discussed in section 5.5.

Let us review what has been proposed in this chapter and connect all the pieces forming a final solution. Starting from the most user-facing part, the application model, an application is defined in MERPSYS together with all the parameters mentioned earlier. The user must inspect the code of the kernel being modeled to determine required metrics and use proper MERPSYS constructs to describe the application behavior. Then the sizes of a grid and a block, and the occupancy are specified. When all parameters are in place, the simulation is launched with the scaling parameter equal to 1. The hardware model, also created by the user, contains definition of a hardware setup, that links devices parametrized with proper attributes describing their computational efficiency. Using functions from computational model, calculations are performed and the user is presented with the estimated kernel execution time, based on which he obtains the new value of lambda parameter. At this point the model is calibrated and functions related to the data transfer may be added to it[47]. It will be then ready to perform further simulations with modified input parameters like: size of the input data, device type or grid size, for investigating how these affect the application execution time.

[47] MERPSYS presents the user with the overall running time of an application, so to calibrate the kernel performance model we first need to execute an isolated simulation, otherwise we would not be able to determine the scaling parameter as given by equation 4.4. This is because the application running time may include other factors like data transfers, etc. The usual approach is to have data transfer functions commented-out in the application model during the first simulation run and uncomment them once the model is calibrated.

This page intentionally left blank.

CHAPTER 5. IMPLEMENTATION

5.1. Introduction

In the previous chapter a detailed description of the performance model was presented, in this one a methodology for obtaining the input parameters of the model is given. We will show how to write a simple CUDA kernel, compile it, analyze compiled source and extract its characteristics. It also contains a systematic walkthrough for constructing a kernel execution graph and applying formulas for throughput bound.

Second part of this chapter focuses on techniques of kernel analysis, including but not limited to: kernel execution time measurement based on CUDA events, kernel clock cycles measurement, writing a simple benchmark kernel with isolated set of arithmetic instructions, and enhancing the code with sections visible under profiler.

In the last part, we tackle the topic of communication between host and the device, analyzing the overhead of a kernel launch, data transfer times and throughput of PCIe bus. All the benchmarking and analysis in this chapter was performed on a NVIDIA GeForce GTX 970 device using CUDA 8.0 runtime with the code compiled for compute capability 5.2 using default compilation options. The host system is installed with Intel Core i7 4790k @4.4GHz, 16GB DDR3 RAM @2400Mhz and the PCIe bus connecting both devices is v3.1 16x. Unless explicitly stated, the measurements of a given type were performed 10 times and the mean value was used.

5.1.1. Writing CUDA kernel

We begin the modeling process by writing a CUDA kernel, the one presented below is a slightly modified version of *saxpy*[48] kernel from a blog post by Harris M. [57]. The main difference when compared to a regular saxpy is that instead of a single multiplication an equivalent number of additions is performed, this allows us to benchmark kernels of various compute intensities. Highlighted part of the code is responsible for a compute intensity of the kernel, i.e. the number of arithmetic instructions executed by each warp. Now let us briefly describe the source code:

- *__global__* tells NVCC that this function is a kernel,
- return type must be void as kernels do not return any values,
- it takes 4 arguments: an integer indicating data size, multiplication parameter *a*, and two pointers to arrays of floating point values allocated in the global memory of the GPU,
- in line 3 global index of the thread is computed, then in line 4 a check is performed to ensure that it is not out of bounds,
- the input value is read from the input array (line 5) and a temporary accumulator is initialized with its value (line 7),

[48] saxpy stands for "single precision *a* times *x* plus *y*" and is one of the basic linear algebra subroutines, for more information please see: https://devblogs.nvidia.com/parallelforall/six-ways-saxpy/ (accessed 15 January 2017)

- *#pragma* from line 9 tells the compiler to not perform any loop unrolling[49],
- the value read from input array is added a required number of times to the accumulator,
- in line 14 the resulting value is summed with the one in the output array.

```
1. __global__ void saxpy2(int n, int a, float *x, float *y)
2. {
3.    int i = blockIdx.x*blockDim.x + threadIdx.x;
4.    if (i < n) {
5.       float t1 = x[i];
6.
7.       float tmp = 0;
8.
9.       #pragma unroll 1
10.      for(int cnt=0; cnt < a; ++cnt) {
11.         tmp += t1;
12.      }
13.
14.      y[i] += tmp;
15.   }
16.}
```

Figure 5.1. Simple CUDA kernel

5.1.2. Analyzing compiled kernel

To accurately determine the kernel's instruction composition, we cannot simply analyze the C or C++ source code since we do not know the optimizations that took place and thus the actual instructions that were generated by the compiler. It is obvious that we need to look at a compiled version of the code, its PTX or SASS representation. PTX is an abstraction over a GPU architecture that is not tied to any hardware type and thus does not accurately resemble the instructions that will be executed by the GPU, hence an inspection of generated SASS code, that is architecture specific, is needed. To extract SASS code from a compiled binary file we will use *cuobjdump*. Figure 5.2 is a SASS representation of the CUDA kernel from Figure 5.1, the highlighted part represents the loop governing the kernel's arithmetic intensity. Mnemonics presented below are quite intuitive to anyone who has previously had an experience with assembly languages, the detailed description is available as well [7] [55]. Few of the instructions worth mentioning include:

- FADD – 32-bit floating point addition,
- IADD32I – 32-bit signed integer addition,
- MOV – value transfer from / to a register,
- LDG – load value from a global memory into a register,
- STG – store value from a register in a global memory,
- ISETP.LT – compare operand values and set predicate register, in this case "less than",
- XMAD – multiply two values and add third value,
- @P0 BRA 0xADDR – jump to a location 0xADDR if the value stored in a predicate register evaluates to true.

[49] Loop unrolling is an optimization where some or all of subsequent loop iterations are converted into series of instructions, for more details see: http://docs.nvidia.com/cuda/cuda-c-programming-guide/#pragma-unroll (accessed 15 January 2017)

An important thing to consider is the state space[50] in which the operands reside, operating on values from a global or shared memory will be much costlier than when using registers. The code highlighted in orange (computational loop) operates almost entirely on values stored in registers, there is only a single reference to a constant memory space, which is one of the fastest as well. Brackets indicate instructions that are dual-issued.

```
1.          MOV R1, c[0x0][0x20];
2.          S2R R0, SR_CTAID.X;
3.          S2R R2, SR_TID.X;
4.          XMAD.MRG R3, R0.reuse, c[0x0] [0x8].H1, RZ;
5.          XMAD R2, R0.reuse, c[0x0] [0x8], R2;
6.          XMAD.PSL.CBCC R0, R0.H1, R3.H1, R2;
7.          ISETP.GE.AND P0, PT, R0, c[0x0][0x140], PT;
8.      @P0 EXIT;
9.          MOV R3, c[0x0][0x144];
10.         SHL R6, R0.reuse, 0x2;
11.         ISETP.LT.AND P0, PT, R3, 0x1, PT;
12.         SHR R7, R0, 0x1e;
13.         IADD R2.CC, R6, c[0x0][0x148];
14.         MOV R0, RZ;
15.{        IADD.X R3, R7, c[0x0][0x14c];
16.     @P0 BRA 0x100;        }
17.{        MOV R5, RZ;
18.         LDG.E R0, [R2];       }
19.         MOV R4, RZ;
20.         IADD32I R4, R4, 0x1;
21.         ISETP.LT.AND P0, PT, R4, c[0x0][0x144], PT;
22.{        FADD R5, R0, R5;
23.     @P0 BRA 0xd0;        }
24.         MOV R0, R5;
25.         IADD R2.CC, R6, c[0x0][0x150];
26.         IADD.X R3, R7, c[0x0][0x154];
27.         LDG.E R5, [R2];
28.         FADD R0, R0, R5;
29.         STG.E [R2], R0;
30.         EXIT;
31.         BRA 0x140;
32.         NOP;
33.         NOP;
34.         NOP;
35.         NOP;
36.         NOP;
```

Figure 5.2. SASS representation of the saxpy2 kernel

It should be further noted that the SASS will not be the same for different architectures, so if the target architecture changes then the model should be re-evaluated, otherwise it will become less accurate. Kernel call parameters are stored in a constant memory space starting from address 0x140, referring to the code of saxpy2 kernel it looks as follows:

- c[0x0][0x140] – n,
- c[0x0][0x144] – a,
- c[0x0][0x148], c[0x0][0x14c] – *x (low and high part),
- c[0x0][0x150], c[0x0][0x154] – *y (low and high part).

Pointers are 64bit values, that is the reason they are stored as two separate 32bit parts.

[50] In other words, the kind of storage area used, sections 5.1 and 6.6 of *Parallel Thread Execution ISA* [7] contain detailed information.

5.2. Extracting parameters from the kernel

In this section, we present the methodology and a detailed walkthrough for extraction of three essential parameters of the performance model that are needed by equation 4.3: occupancy, latency bound and throughput bound.

5.2.1. Obtaining the occupancy

First method to obtain the parameters needed for occupancy calculation is to compile the code with verbose ptxas output, this is done by passing *-Xptxas -v* command line option to NVCC. A sample result is shown in Figure 5.3. It tells us that the kernel uses 8 registers per thread and does not use any shared memory, otherwise information about amount of shared memory used would be included in the output.

```
ptxas info    : Compiling entry function 'saxpy2' for 'sm_52'
ptxas info    : Function properties for saxpy2
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 8 registers, 344 bytes cmem[0]
```

Figure 5.3. Verbose ptxas output for saxpy2 kernel

Having determined the number of registers and amount of shared memory required we can enter them together with the block size in the NVIDIA occupancy calculator [51] to get the theoretical occupancy. Another way is to profile the application, NVIDIA Visual Profiler[51] not only displays the values of shared memory and registers used but it is even capable of calculating both the theoretical and achieved occupancies, Figure 5.4 shows a screenshot taken from the profiler.

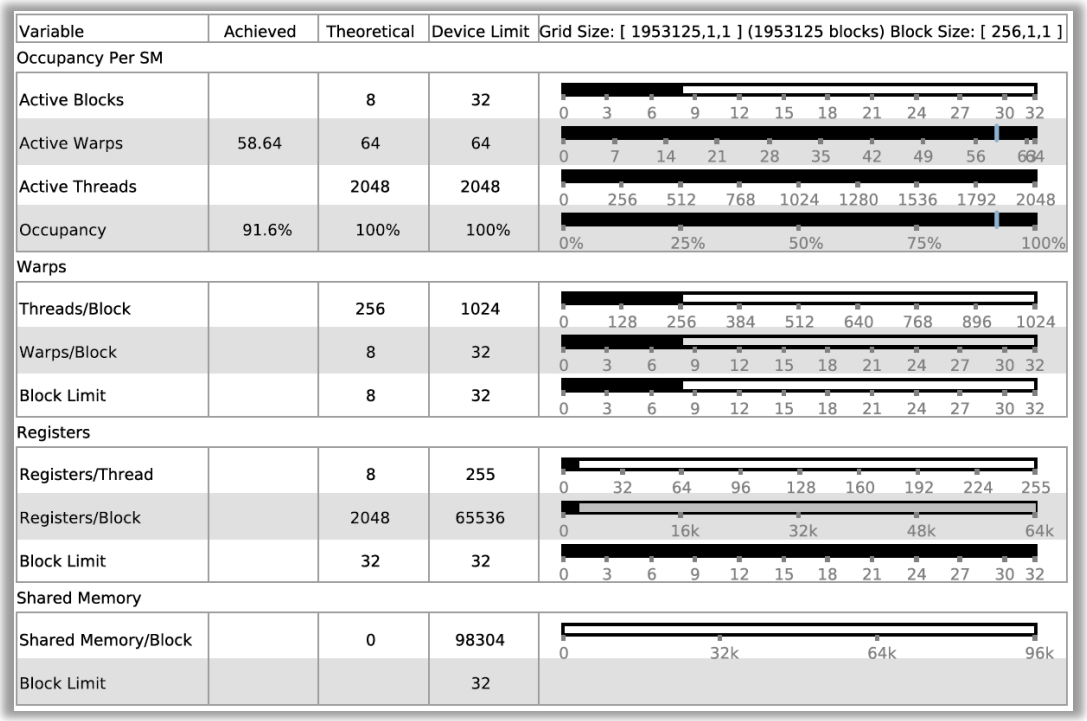

Figure 5.4. NVVP occupancy screen for saxpy2 kernel

[51] https://developer.nvidia.com/nvidia-visual-profiler (accessed 4 April 2017)

5.2.2. Building a kernel execution graph

To construct an execution graph of saxpy2 kernel we start off by creating a directed graph representing the sequence of instructions from Figure 5.2. Each instruction is a single node, additional “start” and “end” nodes mark respectively the entry and exit point of the kernel, for the sake of brevity we skip the irrelevant instructions from the very end. Edges represent the dependencies between the instructions and are labelled with latency values from Table 4.1 and Table 4.2. Straight, solid arrows are used for the ILP latency, which equals 3 cycles for Maxwell architecture, there are a total of 3 dual-issues, these edges are labeled with 0.

Next step is to include latencies from register dependency, these are shown as dotted arrows. We do this by reading the SASS line by line and verifying whether any of the input registers of the current instruction has its value written to by one of the preceding instructions, if yes then a register dependency exists and we add a new edge connecting the predecessor with successor labeled with latency value of the preceding instruction. If this instruction is a memory access, then we use the latency of the storage area being referenced, which in case of this kernel is always the global memory as cache hit ratio equals 0%. This is true because there is no re-use of data - each element is accessed by a single thread and exactly once. To ensure the correctness of this assumption we inspect the profiler output for this kernel regarding the memory throughputs as shown in Figure 5.5. Given that all the global memory accesses must go through the L2 cache and the total throughput reported for L2 Cache and Device Memory (global memory) is almost identical if we disregard some minor differences in the number of accesses, we conclude that all the accesses directly referenced the global memory thus cache hit ratio equals 0%.

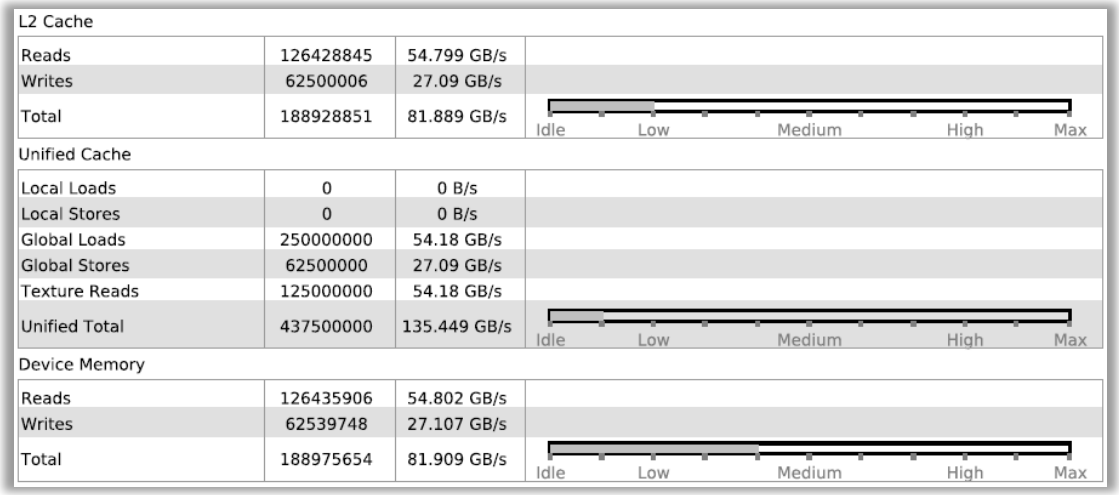

Figure 5.5. Memory bandwidth utilization output from the profiler for saxpy2 kernel

Lastly, we label the inbound edge of the virtual end node with a terminated thread block replacement latency. Once the graph is complete and labelled we use the critical path method to find the latency bound of the kernel. Figure 5.6 shows the resulting graph with critical path highlighted in blue.

The part of the graph colored in orange is the computational loop, which may execute multiple times depending on the computational intensity parameter passed to the kernel. Critical path of the loop itself is highlighted in orange and when calculating the latency bound needs to be multiplied by the number of loop iterations. The final equation for latency bound is as follows:

$$latency\ bound = 92 + 700 + 150 + 24a = 942 + 24a \tag{5.1}$$

Where first term is sum of instructions' latencies, second one is latency of memory accesses, third is thread block replacement latency, and the fourth one is latency of the loop multiplied by the computational intensity parameter.

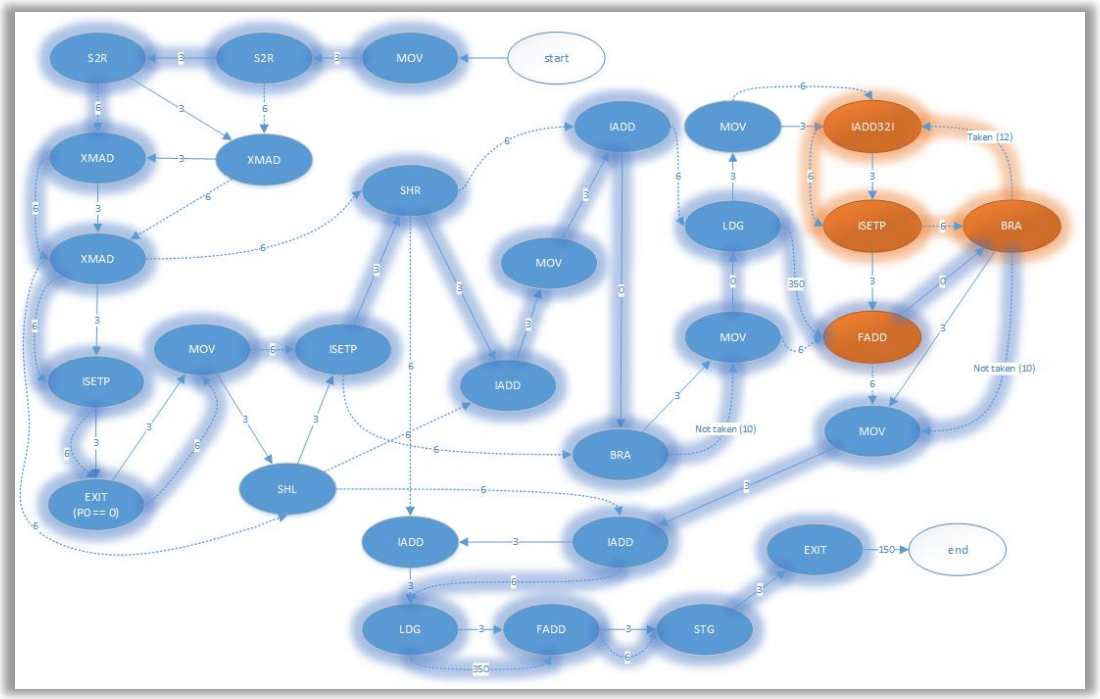

Figure 5.6. Execution graph for saxpy2 kernel

5.2.3. Determining the throughput limits

As a first step in throughput limit calculation we need to count the instructions in generated SASS. It can be done either by reading directly from the assembly listing - Figure 5.2 or based on the kernel execution graph - Figure 5.6, whichever method is preferred. In our example, we have 23 CUDA core instructions that are executed once and 4 instructions in a loop, so the resulting formula is: $INS_{CUDA} = 23 + 4a$. It should be noted that in this kernel everything except memory accesses is processed by CUDA cores, this would not be the case if there were any instructions designated for a double precision or SFU units.

When getting the number of instructions issued we must remember to address the dual-issues and re-issues. In our kernel, we have 3 accesses to global memory, all three are fully coalesced[52] so there are no re-issues taking place. There are 3 dual-issues[53], each resulting in

[52] We can easily tell it by looking at the source code from Figure 5.1, the memory access patter is trivial and has a stride of 1. In more complicated scenarios the profiler may be used to investigate the number of memory transactions per single instruction and on this basis, one can determine the number of re-issues.

a pair of instructions being issued together during a single cycle. Summing up, we have 23 CUDA core instructions issued once and four of these in a loop, three memory instructions, and three dual-issues which we subtract from the total number, so we get: $INS_{issued} = 23 + 3 - 3 + 4a = 23 + 4a$.

Number of bytes transferred to and from a global memory depends on four factors: number of the memory access instructions, cache hit rate, operand size, and access pattern. saxpy2 kernel contains 3 memory access instructions: two loads and a single store, but as we have already stated when defining the theoretical model, for the sake of brevity we do not differentiate between these two as the difference is marginal. In all three cases, a single 4-byte value is accessed per thread and as was mentioned in preceding paragraph, the accesses are fully coalesced so there are no redundant memory transactions when accessing the data, this gives us: $GMem_{bytes} = 3 \times 4\ Bytes \times 32 = 384\ bytes$. Furthermore, as shown in the previous section there are no cache hits hence all these bytes are accessed directly in the global memory and we can entirely omit both the L2 and L1 caches. If this was not the case then we would separately consider a fraction of 348 bytes equal to cache hit ratio as being accessed from cache, which has a higher throughput than the global memory itself.

Now that we have all the throughput related parameters extracted from the kernel we need to list those specific to the device the application will run on, which is GTX 970 in our case. First two parameters can be read directly from Table 2.1, compute capability of our card is 5.2, hence we have: $CUDA\ cores = 128$, $schedulers = 4$. What remains now is the third parameter – bandwidth of the global memory, we calculate it using equation 4.7 substituting the required values as given in Table 6.1: $GMem_{clock} = 1753\ Mhz$, $bus\ width = 256\ bits$, $data\ rate = 4$, $SMs = 13$, $SM_{clock} = 1253\ Mhz$. This gives us:

$$GMem_{bandwidth} = \frac{1753 \times \frac{256}{8} \times 4}{13 \times 1253} = 13.78\ \frac{Bytes}{cycle} \tag{5.2}$$

The final step is to gather up all these parameters, substitute them to equation 4.5 and get the reciprocal:

$$throughput\ bound = \frac{1}{\max\left(\frac{32 \times (23 + 4a)}{128}, \frac{23 + 4a}{4}, \frac{384}{13.78}\right)} \tag{5.3}$$

5.3. Kernel analysis techniques

This section provides a generic overview of the kernel analysis techniques, which are an essential part of many studies and provide means to benchmark and measure the kernels executed on a GPU. Instead of focusing on a specific characteristic of a particular kernel we will show by example how to prepare a simple benchmark, measure its running time using CUDA events and determine the number of clock cycles elapsed with intrinsic *clock64()* function. We will also present an automated approach to the time measurement paired with additional section markers which ease the process of analyzing application execution timeline in the profiler.

[53] Visible as instructions grouped together with brackets in Figure 5.2

5.3.1. Writing a simple benchmark

The number and type of the instructions to be executed by a single thread running a kernel is by far the most important contributor to the overall execution time, it is also very demanding in terms of developing a theoretical model because of how highly parallel and complicated the GPU execution model is. Considering the high complexity of the hardware our analysis will be narrowed down to measuring the execution time of a very simple kernel with a single computation loop performing single-precision arithmetic. Execution time will be measured using a timer class described in section 5.3.3.

Partial source code of the benchmark is presented in Figure 5.7, it should be noted that it does not include all granularity[54] configurations that were used. saxpy3 kernel was launched 1000 times for 6 different granularities and a broad range of loop iterations – the more iterations performed, the higher the arithmetic intensity of the kernel was. Mean execution times of a single kernel launch obtained by running the benchmark are presented in Figure 5.9.

```
1.  __constant__ bool dummyFlag = false;
2.
3.  template <int Granularity>
4.  __global__ void saxpy3(int n, int a, float* y) {
5.     int i = blockIdx.x*blockDim.x + threadIdx.x;
6.     if (i < n) {
7.        float input = 1;
8.        float tmp = 0;
9.
10.    for(int cnt = 0; cnt < a; ++cnt) {
11.       #pragma unroll
12.       for(int k = 0; k < Granularity; ++k) {
13.          tmp += input;
14.       }
15.    }
16.
17.    if( dummyFlag ) {
18.       y[i] += tmp;
19.    }
20. }
21.}
22.
23.void arithmeticInstructionsBenchmark(int computeIntensity) {
24.  int N = 10 * 1024 * 1024;
25.  int STEPS = 1000;
26.  {
27.     SimpleTimer timer;
28.     for (int step = 0; step < STEPS; ++step)
29.        saxpy3<1><<<(N+255)/256, 256>>>(N, computeIntensity, 0);
30.  }
31.}
```

Figure 5.7. Arithmetic instructions benchmark source code

To make the granularity of the kernel easily customizable a template with a parameter specifying it was used. Since *Granularity* is a compile time constant value, the inner loop gets completely unrolled by the compiler and thus we obtain a single loop with the desired arithmetic instructions count. Because we want to measure only the time spent on executing the arithmetic instructions we had to get rid of accesses to the global memory. Although removing load

[54] Granularity is a number of arithmetic instructions executed every loop cycle.

instruction is easy and straightforward the same cannot be done for store operation as the optimizer would consider the kernel's code as not producing any results and remove it entirely. To avoid "dead code removal" optimization we introduce a dummy flag stored in the device constant memory space and encapsulate the store instruction in a conditional block dependent on this flag. Its value can potentially change during the runtime and thus the optimizer is tricked into thinking that the results of the kernel are used. Furthermore, as the flag resides in constant memory space the additional overhead is reduced to the minimum and does not affect the obtained results in any significant way. Still, it is a compromise between ease of use and desired accuracy, to achieve very accurate results without the need to introduce any additional flags inline PTX assembly can be used as shown in chapter III of work by Bombieri, et al. [58].

5.3.2. Measuring clock cycles

Measurement of the clock cycles spent executing the kernel required a slight modification of its code, new version is shown in Figure 5.8 with new parts being highlighted. Each thread stores the value of the clock registry[55] before proceeding to computational part and upon completing it calculates the difference with the current value and stores it in an array allocated in a global memory, which is then used on the host to calculate the average number of cycles elapsed during execution of a single thread. The results are presented in Figure 5.10.

```
1.template <int Granularity>
2.__global__ void saxpy3_clock_cycles(int n, int a, float* x, long long int* y) {
3.  int i = blockIdx.x*blockDim.x + threadIdx.x;
4.  if (i < n) {
5.     long long int start = clock64();
6.     float input = 1;
7.     float tmp = 0;
8.
9.     for(int cnt=0; cnt < a; ++cnt) {
10.       #pragma unroll
11.       for(int k = 0; k < Granularity; ++k) {
12.          tmp += input;
13.       }
14.    }
15.
16.    y[i] = clock64() - start;
17.
18.    if( dummyFlag ) {
19.       x[i] += tmp;
20.    }
21. }
22.}
```

Figure 5.8. saxpy3 kernel with additional code measuring clock cycles

Using clock registry to measure the cycles for different sets of instructions is a common approach for extracting hardware latency (e.g. timing a long sequence of dependent floating point additions to obtain the latency of the arithmetic pipeline) and throughput characteristics of a GPU that was used among others in [16], [49], [59], [53] and [58]. This work is also based on values obtained this way and we wanted to experimentally confirm it to be reliable. If we

[55] http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#time-function (accessed 10 March 2017)

compare the measured execution time of a kernel with the average number of clock cycles executed per warp we would expect this metrics to exhibit the same behavior for different granularities and iterations count because the execution time can be derived directly from the latter value as shown in equation 4.1. When we look closely at Figure 5.9 and Figure 5.10 it is visible that the plots for different granularities (from 1 to 32) are the same on both charts, this is the expected result that proves this clock cycles measurement methodology to be correct.

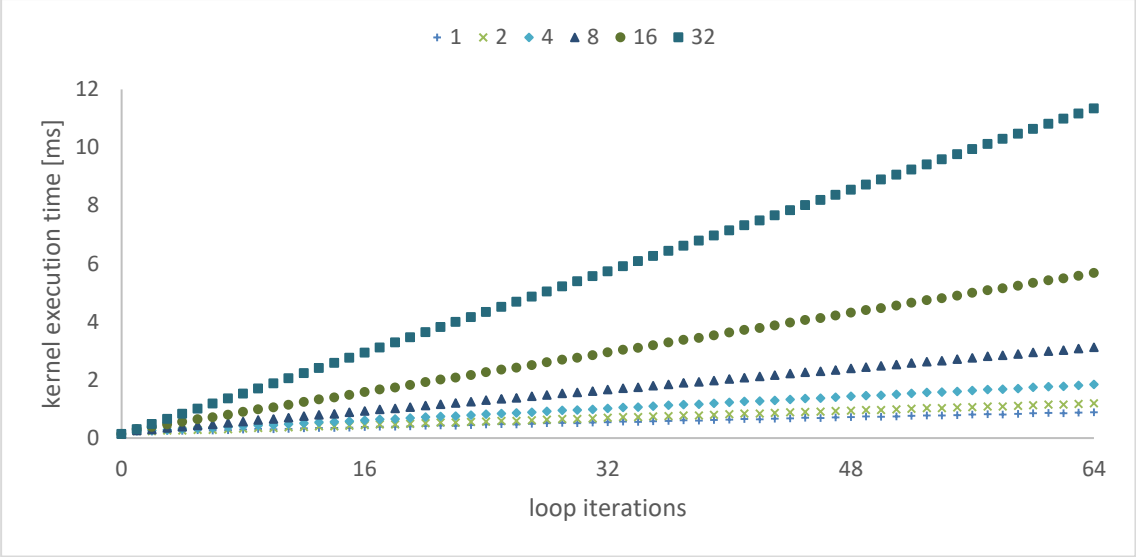

Figure 5.9. Kernel execution time in function of its arithmetic intensity

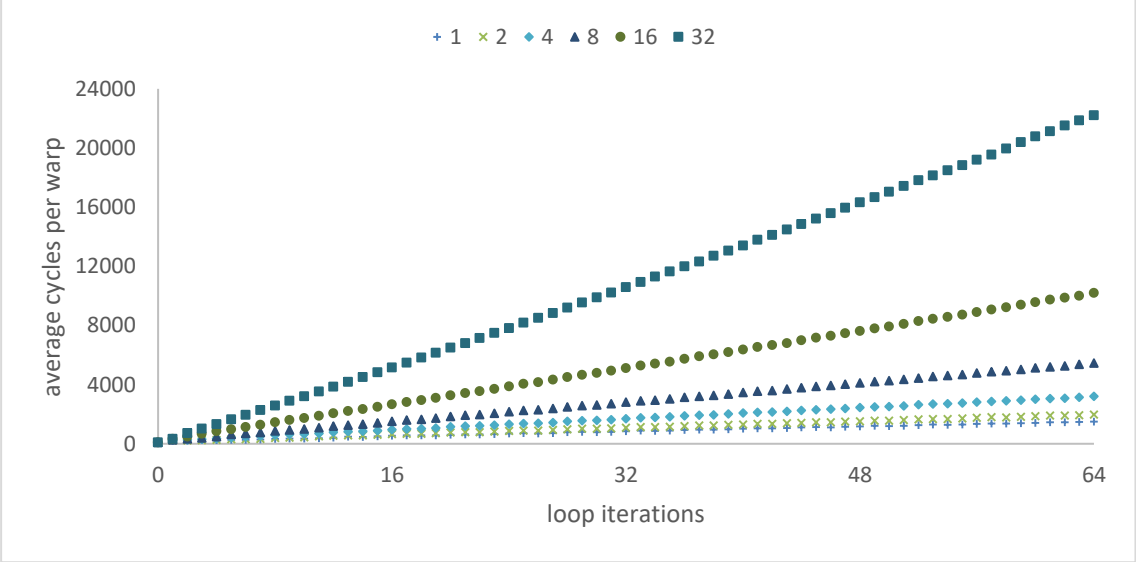

Figure 5.10. Average clock cycles elapsed when executing a single warp

5.3.3. Time measurement and profiling code

To accurately measure the execution time of various operations performed using a GPU a simple timer class that is very convenient to use was prepared. It utilizes idiom common to C++ known as RAII[56] (Resource Acquisition Is Initialization), time measurement is performed by

[56] http://en.cppreference.com/w/cpp/language/raii (accessed 10 March 2017)

creating two CUDA events[57] and registering them in the default stream. Start event is inserted into the stream before the operation of interest (e.g. kernel launch, data copy) using the class' constructor. The end event is registered inside the destructor's body, then a synchronization with the GPU is performed and elapsed time is computed. The destructor is invoked when the lifetime of the object expires, to control the lifetime of the object we introduce an additional block[58] enclosing the operation of interest, for an example usage see Figure 5.7. In addition to providing an accurate time measurement, this class also leverages the NVTX library[59] to make the analysis of the execution traces in the profiler much easier, for an example output see Figure 5.15 ("empty kernel" and "empty kernel synchronized" markers).

```
#include <nvToolsExt.h>
#include <iostream>
#include <iomanip>
class SimpleTimer {
public:
   SimpleTimer(std::string name = "") : name(name) {
   cudaEventCreate(&start);
      cudaEventCreate(&end);
      nvtxEventAttributes_t nvtxEvent = createNvtxEvent();
      nvtxRangePushEx(&nvtxEvent);
      cudaEventRecord(start);
   }

   ~SimpleTimer() {
      cudaEventRecord(end);
      cudaEventSynchronize(end);
      nvtxRangePop();
      float time_ms;
      cudaEventElapsedTime(&time_ms, start, end);
      std::cout << std::fixed << std::setprecision(6) << name << " " << time_ms;
   }

private:
   std::string name;
   cudaEvent_t start, end;

   nvtxEventAttributes_t createNvtxEvent() {
      const uint32_t colors[] = { 0x0000ff00, 0x000000ff, 0x00ffff00,
         0x00ff00ff, 0x0000ffff, 0x00ff0000, 0x00ffffff };
      const int num_colors = sizeof(colors) / sizeof(uint32_t);
      static int colorCounter = 0;
      nvtxEventAttributes_t eventAttrib = {0};
      eventAttrib.version = NVTX_VERSION;
      eventAttrib.size = NVTX_EVENT_ATTRIB_STRUCT_SIZE;
      eventAttrib.colorType = NVTX_COLOR_ARGB;
      eventAttrib.color = colors[colorCounter++ % num_colors];
      eventAttrib.messageType = NVTX_MESSAGE_TYPE_ASCII;
      eventAttrib.message.ascii = name.c_str();
      return eventAttrib;
   }
};
```

Figure 5.11. Time measurement code

[57] https://devblogs.nvidia.com/parallelforall/how-implement-performance-metrics-cuda-cc/ (accessed 10 March 2017)

[58] http://en.cppreference.com/w/cpp/language/scope#Block_scope (accessed 10 March 2017)

[59] https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-generate-custom-application-profile-timelines-nvtx/ (accessed 10 March 2017)

5.4. Modeling communication between host and the device

For a moment we leave the analysis of a kernel behind and move to another important topic in this case related not only to the GPU but also to the host, which is the communication between host and the device. We distinguish two essential elements here: kernel launch and data transfers, both are present in every application that offloads parts of the computation to the GPU and contribute to the overall running time, hence need to be included in theoretical model.

5.4.1. Kernel launch overhead

An important factor to consider when modeling the computations on the GPU is the overhead of a kernel launch, even though this is an asynchronous operation it still requires communication with the device driver to offload the computations and thus takes a nonzero amount of time. Furthermore, we consider the execution overhead as the total time required to launch a kernel, not only the asynchronous operation itself. Hence, a synchronization point is inserted after the launch operation that will be used to compute the total running time. To measure the overhead, a simple benchmark that is presented in Figure 5.12 was used. It is a loop that launches an empty kernel 100 times and measures the total elapsed time using CUDA event based time measurement.

```
1.  __global__ void empty_kernel(){};
3.
4.  void kernelLaunchBenchmark(int gridSize, int blockSize) {
5.     SimpleTimer timer("empty kernel");
6.     int STEPS=100;
7.     for (int step = 0; step < STEPS; ++step) {
8.        empty_kernel<<<gridSize,blockSize>>>();
9.        //cudaDeviceSynchronize();
10.    }
11. }
```

Figure 5.12. Kernel launch benchmark source code

Results representing the launch time of a single kernel obtained from launching the benchmark for varying grid configurations are presented in Figure 5.13. Execution time increases linearly with the grid size, it is also visible that measurements are almost identical for block sizes of 1, 2, 4, 8, 16, 32, 128 and 256 threads. Then the execution time increases for sizes of 512 and 1024 threads, a small increase can be noticed for 64 threads, likely because of some hardware characteristics. Figure 5.14 shows the execution times for representative block sizes.

It was experimentally confirmed that the number of arguments ranging from 0 to 8 passed to the kernel does not affect its launch time. However, the grid configuration (block number and block size) plays an important role, since if there are more thread blocks than the SMs can handle at a given time then some of them need to wait before being processed. This is visible in both figures as a point where a steady increase of the kernel execution time begins, so for block counts of 128 – 512, depending on which blocks size we look at. We have thus confirmed the existence of a latency caused by replacement of a terminated thread block that

was mentioned in section 4.4.2. In MERPSYS' application model, the effect of a block replacement latency will be covered by the latency bound as shown in equation 5.1.

Synchronized launch of a kernel was also benchmarked (function call commented out in line 9 in Figure 5.12) but it turned out to not have any effect on the running time of the benchmark, measured times were almost identical. To further ensure the correctness of the results traces from NVVP were analyzed and it was confirmed that indeed the synchronization took place after each kernel launch, but since in both scenarios the kernels were executed in rapid succession it did not affect the execution time, Figure 5.15 shows the profiler output.

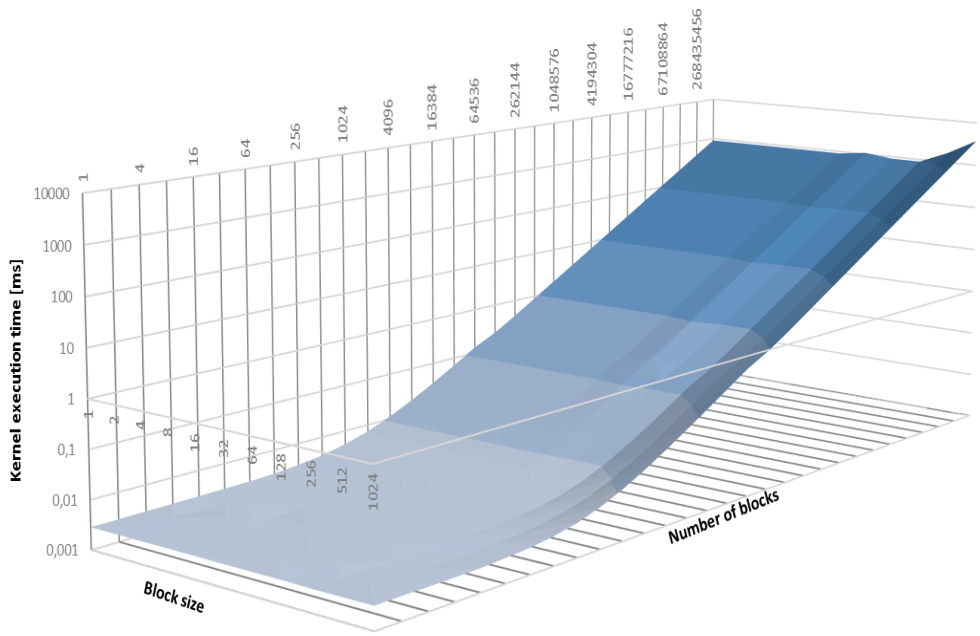

Figure 5.13. Kernel launch overhead in function of grid configuration

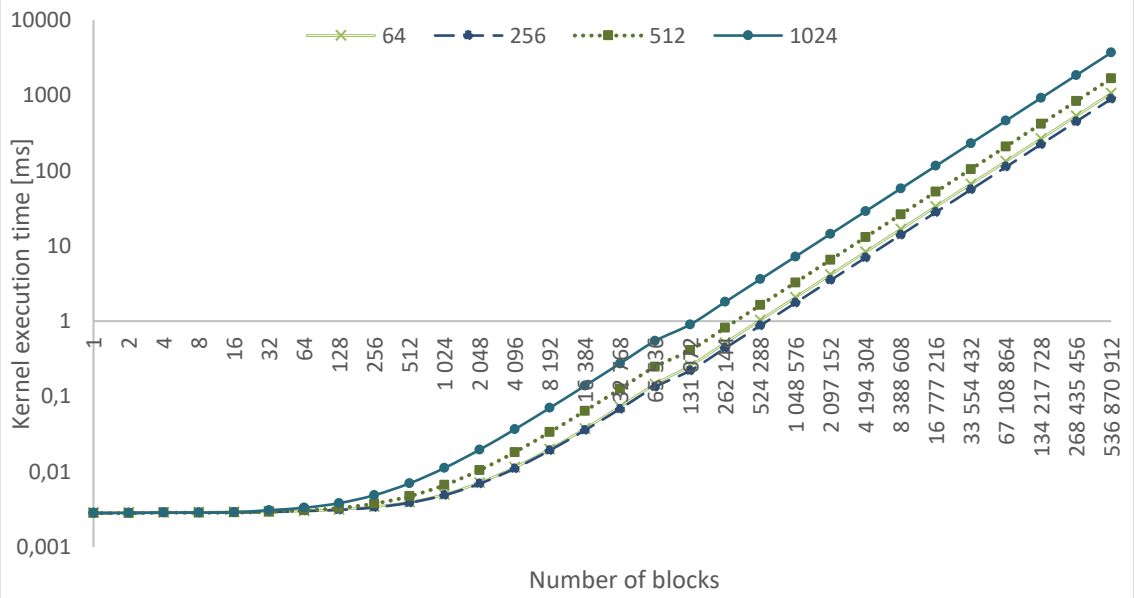

Figure 5.14. Kernel launch overhead for representative block sizes

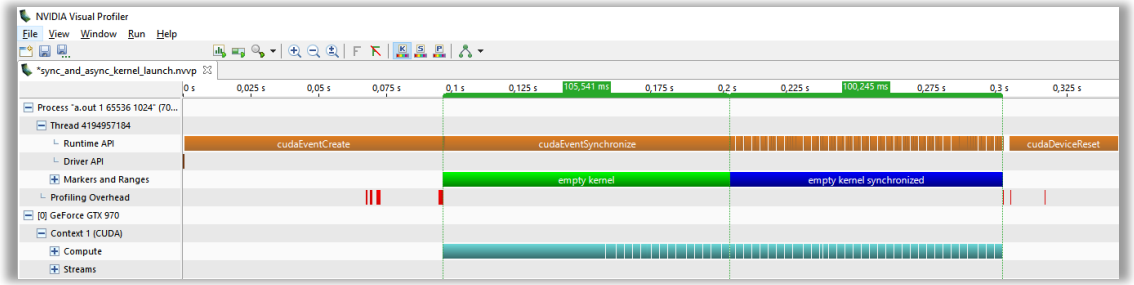

Figure 5.15. Kernel launch traces from NVVP

5.4.2. Data transfers

To measure data transfer times, we will consider a situation where buffers in host and device memories are allocated explicitly and the data transfers are explicitly stated in the code. Worth mentioning is the fact that recent devices support Unified Memory model (see appendix J of *CUDA C Programming Guide* [8]), which is a lot more programmer friendly but also would be much harder to simulate as in this case data buffers and data transfers are managed directly by the CUDA runtime and thus the operations that we intend to measure execution time of are performed transparently to the user. Figure 5.16 shows the source code of a benchmark that will be used, it consists of four major parts: data allocation, data transfers from the host to the device, data transfers from the device to the host and release of the allocated memory. For both directions data transfers were repeated 1000 times in a loop and the total loop running time was measured, Figure 5.17 shows the mean time of a single data transfer depending on the size of the data.

```
1. void memcpyBenchmark(int N) {
2.     int STEPS = 1000;
3.     float* x = new float[N];
4.     float* d_x;
5.     cudaMalloc(&d_x, N*sizeof(float));
6.
7.     {
8.        SimpleTimer timer("memcpy HtD");
9.        for (int step = 0; step < STEPS; ++step) {
10.          cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
11.       }
12.    }
13.
14.    {
15.       SimpleTimer timer("memcpy DtH");
16.       for (int step = 0; step < STEPS; ++step) {
17.          cudaMemcpy(x, d_x, N*sizeof(float), cudaMemcpyDeviceToHost);
18.       }
19.    }
20.
21.    delete[] x;
22.    cudaFree(d_x);
23.}
```

Figure 5.16. Data transfer benchmark source code

Measurements were performed for data sizes up to 2GB as this is the upper limit imposed by the hardware used for tests – NVIDIA GeForce GTX 970 has 4GB of the device memory and part of it was used for graphical display. Nevertheless, Figure 5.17 shows that for

large data sizes the trend is purely linear so by using the equation 4.8 our findings can be easily extrapolated to reflect data sizes exceeding this hardware limit. On the other hand, mean value of the transfer times obtained for very small data sizes (from 4 to 512 bytes - listed in Table 5.1) can be considered a startup time of the communication. A significantly lower value for transfer size of 0 bytes is a result of a special handling of a case when there is no data to be transferred, likely the CUDA runtime does not initiate the transfer at all and so the function call has no effect.

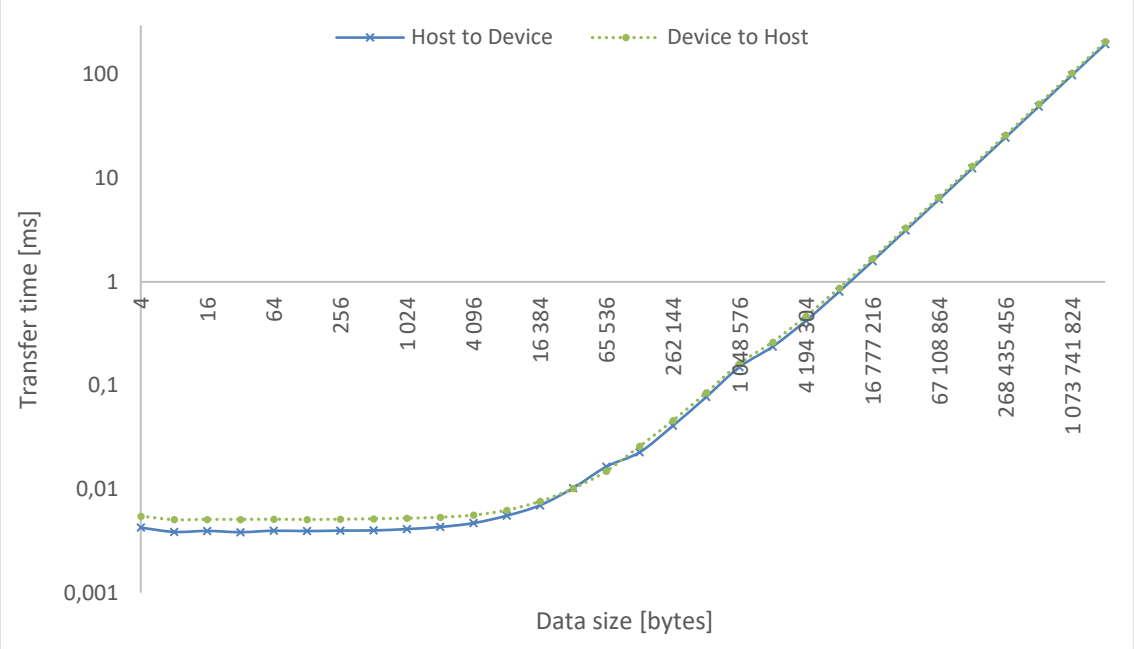

Figure 5.17. Measured data transfer times

Table 5.1. Measured data transfer times for small data sizes

Data size [B]	Host to Device [ms]	Device to Host [ms]
0	0.0005354	0.0005030
4	0.0042464	0.0054774
8	0.0038608	0.0050707
16	0.0039446	0.0051018
32	0.0038378	0.0051005
64	0.0039648	0.0051171
128	0.0039302	0.0050918
256	0.0039766	0.0051203
512	0.0039882	0.0051757

To calculate the actual throughput that was achieved we will use a transformed version of the equation 4.8 and apply it to values from Figure 5.17.

$$throughput = \frac{T - t_s}{n} \tag{5.4}$$

Where:

T – data transfer time

t_s – startup time

n – data size

Figure 5.18 shows the actual values of throughput that were achieved in both directions, one can notice that higher throughputs are reached when the data is transferred from the host to the device. As a remark, throughput was not calculated for data size smaller than 1024 bytes as it was assumed to be the threshold below which startup time is equal to the total transfer time, hence the equation does not apply because it would produce a throughput equal to 0.

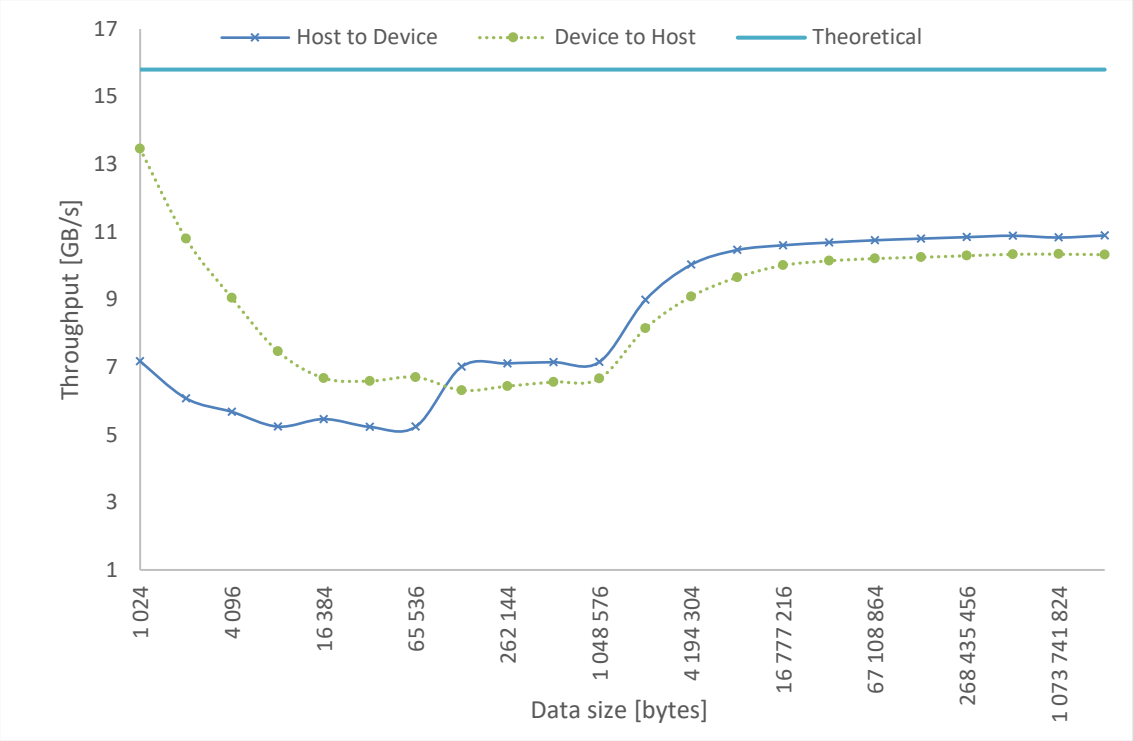

Figure 5.18. Measured data transfer throughput

Furthermore, it is visible that the bus is better utilized for larger data sizes but even for the largest transfer of 2GB the throughput was noticeably lower than the theoretically achievable one. Due to this fact, in our implementation we introduce an additional scaling parameter λ to equation 4.8 that allows us to fine-tune the link bandwidth so that it reflects the actual characteristic of a given hardware setup.

$$T = t_s + \frac{n}{bw \times \lambda} \tag{5.5}$$

Note that both the λ and t_s will have different values for each transfer direction. Scaling parameter can be obtained similarly to the one for the kernel execution time denoted by equation 4.4, i.e. by dividing the estimated transfer time by the measured one. To get the value of the bandwidth parameter we first need to determine the PCIe configuration, for this purpose we use *nvidia-smi*[60], a command line tool supplied by NVIDIA as shown in Figure 5.19. It is important to remember that this query must be executed when the GPU is in use, this is because a lower than the actual number may be reported for both the link width and PCIe generation when the device is idle [60, p. 10]. Having obtained these two values, we can refer to Table 4.3 to get the bandwidth parameter.

[60] https://developer.nvidia.com/nvidia-system-management-interface (accessed 30 June 2017)

```
$ nvidia-smi --query-gpu="pcie.link.gen.current,pcie.link.width.current" \
$ --format=csv
```

Figure 5.19. nvidia-smi query for retrieving the current PCIe configuration

Our testbed with GTX 970 uses PCIe v3 16x, based on this and measurements from this chapter we substitute the parameters of equation 5.5 and thus we get the formula for data transfer time in function of its size for both directions as shown below.

$$T_{Host\ to\ Device} = 3.9687 \times 10^{-6} + \frac{n}{15.8 \times 10^{9} \times 0.689} \tag{5.6}$$

$$T_{Device\ to\ Host} = 5.1569 \times 10^{-6} + \frac{n}{15.8 \times 10^{9} \times 0.653} \tag{5.7}$$

It should be further noted that the analysis from this chapter has a very limited scope and did not consider things like Unified Virtual Addressing, Unified Memory, Pinned Memory[61] and different memory transfer functions supplied by the CUDA environment. However, it is sufficient for modeling of simple parallel applications written using CUDA that were used in this work.

5.5. Implementing the model in MERPSYS

In this section, we show how the theoretical model was implemented in MERPSYS. At first, we demonstrate how to create a system model representing our testbed, including the creation of new hardware units. Then we proceed to computational model definition, i.e. the programmatic representation of what was shown in sections 4.4 and 4.5. Next, we move to the application model definition, which will represent the CUDA application being modeled, in the meanwhile explaining how all these pieces are tied together and finally elaborate on how to customize the application launch parameters and execute the simulation.

Creation of hardware units that are customized with parameters required by our theoretical model was the first implementation step. As it was stated in the introduction to this chapter, all measurements were performed using NVIDIA GTX 970 and PCIe v3.1 16x, for these two devices we have created a representation in MERPSYS' database as shown in Figure 5.20. It should be further noted that the lambda parameter from equation 5.5 was included as an attribute of the PCIe hardware unit, we assumed that the effective bus utilization is a parameter of the hardware itself and its value does not change for different application types if the memory transfers are performed using the same functions. Shall a requirement for this parameter to be customizable via application model arise, it can be moved there and passed in the function calls the same way it was done for kernel parameters, as will be further shown.

[61] https://devblogs.nvidia.com/parallelforall/how-optimize-data-transfers-cuda-cc/ (accessed 30 June 2017)

Figure 5.20. Hardware units with customized attributes: GPU (left), interconnect (right)

Next thing to do is creation of a testbed representation using the hardware model editor in MERPSYS' editor application. Two hardware units that we have just created are used in this model together with a unit representing the CPU, note that this model directly resembles the actual hardware structure where GPU and CPU are connected using a PCIe bus. Figure 5.21 is a screenshot from the editor application showing the created model, GPU unit is selected with its properties visible on the right side, moving from the top to the bottom:

- Hardware – hardware unit represented by this element, in our case the GTX 970 GPU visible on the left of Figure 5.20,
- Count – the number of hardware units of a given type, for example in a multi-GPU setup we could set the count to 2, 3, 4, etc.,
- Name – displayed name of the element,
- Model – computational model connected to the element, in our case the one from Figure 5.22, description of which will follow,
- Labels – labels assigned to the element, these form the connection between hardware and application models. Name is the identifier of the label and multiplicity tells us the number of processes with this label that may be executed on a single unit of this type[62].

[62] For a reminder see section 2.5.

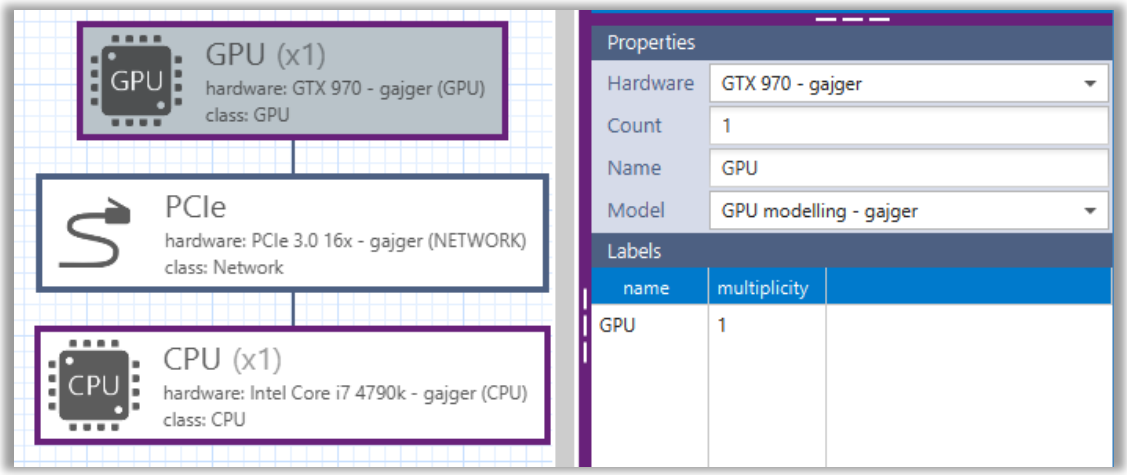

Figure 5.21. Testbed configuration in the system model editor

Another crucial component of the simulation suite is the computational model, in which we have implemented the equations comprising the model. These are written in JavaScript using MERPSYS' shallow functions, and are depicted in Figure 5.22. *getTimeGPUModel* represents equations 4.1 and 4.3, *getWarpsLaunched* equation 4.2, while *getThroughputBound* stands for equations 4.5 and 4.6. *getTimeP2PCPUtoGPU* and *getTimeP2PGPUtoCPU* are required for data transfer modeling and represent equations 5.6 and 5.7 respectively.

Parameters characterizing hardware, i.e. hardware unit attributes are directly accessible from within the computational model functions. A good example would be line 12 from Figure 5.22, which references *SMCoreClock* attribute of the GPU. On the other hand, application specific parameters are passed from the application model using proper arguments of the computational or communicational function calls. These are then accessible as JSON objects containing key-value pairs. When we look at the signature of *getTimeGPUModel* we can notice *params* argument, which is this JSON object. The value stored under a specific key is accessed using *params.get()* method.

The application model is an abstract representation of the program being modeled, in our case it consists of few data transfers and a kernel launch, the model is depicted in Figure 5.23 and the source code of the CUDA application is listed in Figure 6.1. At the very beginning parameters characterizing the kernel are defined, this part is based on what was described in section 5.2, next we have the actual application divided into two branches that are distinguished based on labels (types of processes), one for a CPU and one for a GPU. It is visible that the CPU only handles data transfers and each of the communication functions specified for it has its counterpart present in the GPU section. Sequencing of the operations that originate from the communication functions is managed internally in MERPSYS. The input parameters of the theoretical model for the kernel execution time are specified in the GPU section, these are put into a key-value map, that is converted by MERPSYS into a JSON object and passed to the appropriate function of the computational model, in our case it is *getTimeGPUModel*.

```
function getTimeGPUModel(params) {
   var warpsLaunched = getWarpsLaunched(params.get('gridSize'),
      params.get('blockSize'));
   var throughputBound_WPC = getThroughputBound(
      params.get('CUDACoreInstructions'), params.get('issuedInstructions'),
      params.get('GMEMBytesTransferred'));
   var latencyBound_WPC = params.get('occupancy') / params.get('latencyBound');
   var warpThroughput = Math.min(latencyBound_WPC, throughputBound_WPC);
   var kernelExecutionTime_cycles = warpsLaunched /
      (warpThroughput * SMs * params.get('kernelExecutionLambda'));
   var kernelExecutionTime_seconds = kernelExecutionTime_cycles /
      (SMCoreClock * Math.pow(10,6));

   return kernelExecutionTime_seconds * 1000000;
}

function getWarpsLaunched(gridSize, blockSize) {
   return gridSize * Math.ceil(blockSize/warpSize);
}

function getThroughputBound(CUDACoreInstructions, issuedInstructions,
   GMEMBytesTransferred) {
   var CUDACoreInstructions_CPW = CUDACoreInstructions * warpSize / CUDACores;
   var issuedInstructions_CPW = issuedInstructions / schedulers;
   var GMEMBytesTransferred_CPW = GMEMBytesTransferred / GMEMThroughput;
   return 1 / Math.max(CUDACoreInstructions_CPW, issuedInstructions_CPW,
      GMEMBytesTransferred_CPW);
}

function getTimeP2PGPUtoCPU(params) {
   var transferTime = startupDtH + params.get('dataSize') /
      (bandwidth * lambdaDtH);
   return transferTime * 1000000;
}

function getTimeP2PCPUtoGPU(params) {
   var transferTime = startupHtD + params.get('dataSize') /
      (bandwidth * lambdaHtD);
   return transferTime * 1000000;
}
```

Figure 5.22. Computational model

Most of the parameters defined in the application model are not constant values, but are instead computed based on the model input parameters, that are specified in the *Variables* section of the editor application, which may be customized either from the application model editor screen or simulation launch screen, the latter is shown in Figure 5.24. *Variables* section is in the bottom left corner, values located there may be easily changed between the simulation runs, what allows for an effortless customization. For example, if we want to test the application's behavior for a different number of input elements or change the characteristic of the kernel by tweaking *computeIntensity* parameter.

In the *Labels* section of the simulation screen, we define what processes are going to be launched. A label represents a process and its value is the number of processes of this type, note that these are not the same labels that we have already seen in the hardware model, although both kinds are directly related. The relation is that a process identified by a label "GPU" requires the very same label to be assigned to at least one of the hardware components.

```
1.  Integer CUDACoreInstructions = new Integer(23 + 4 * computeIntensity);
2.  Integer issuedInstructions = new Integer(26 + 4 * computeIntensity);
3.  Integer GMEMBytesTransferred = new Integer(384);
4.  Integer latencyBound = new Integer(942 + 24 * computeIntensity);
5.  Integer gridSize = new Integer((numElements + blockSize - 1) / blockSize);
6.  Double kernelExecutionLambda = new Double(0.703787);
7.
8.  if (tag.equals("CPU")) {
9.     sim.p2pCommunicationSend(4*(double)numElements, "GPU", ConstVar.HostToDevice);
10.    // artificial synchronization point - send in MERPSYS is non-blocking
11.    // whilst cudaMemcpy is blocking, hence the next transfer can only
12.    // be performed when the preceding one is complete
13.    sim.p2pCommunicationReceive("GPU");
14.    sim.p2pCommunicationSend(4*(double)numElements, "GPU", ConstVar.HostToDevice);
15.
16.    sim.p2pCommunicationReceive("GPU");
17. } else {
18.    sim.p2pCommunicationReceive("CPU");
19.    sim.p2pCommunicationSend(0, "CPU"); //synchronization
20.    sim.p2pCommunicationReceive("CPU");
21.
22.    Map GPUModelParams = new HashMap();
23.    GPUModelParams.put("CUDACoreInstructions", CUDACoreInstructions);
24.    GPUModelParams.put("issuedInstructions", issuedInstructions);
25.    GPUModelParams.put("GMEMBytesTransferred", GMEMBytesTransferred);
26.    GPUModelParams.put("latencyBound", latencyBound);
27.    GPUModelParams.put("blockSize", new Integer(blockSize));
28.    GPUModelParams.put("gridSize", gridSize);
29.    GPUModelParams.put("occupancy", new Double(occupancy));
30.    GPUModelParams.put("kernelExecutionLambda", kernelExecutionLambda);
31.    sim.computation(GPUModelParams,SoftwareStack.Undefined,OptimizationType.None);
32.
33.    sim.p2pCommunicationSend(4*(double)numElements, "CPU", ConstVar.DeviceToHost);
34. }
```

Figure 5.23. Application model

Furthermore, the maximum number of processes of a given type that may be launched depends on the hardware units' execution capacity, which we may easily compute by inspecting the hardware model and multiplying the label multiplicity by the count of hardware units. For example, in Figure 5.21 the label "GPU" has multiplicity of 1 and there is only a single unit, hence when launching the simulation, the number of processes of this type must also not exceed 1. As we have already seen, this label is also referenced in the application model using *tag.equals*() construction, this demonstrates that the labels are what binds all of the simulation pieces together.

With the variables and labels defined, what is left is to launch the simulation. We first specify the name of a simulation queue, *gajgerqueue* in our case, this name must match the one that was used when a simulator application was started. Otherwise, the simulation will be sent to a JMS queue on MERPSYS server that does not have any simulators connected and will stay there awaiting execution, hence the simulation will not be able to complete and we will not see the results. After specifying a proper name of the simulation queue, we start the simulation, and once it finishes its execution the outcome is presented in the top right corner in the *Results* section. The simulated application execution time is listed as the *Overall time*, we will refer to it in the next chapter when comparing predicted values with the ones measured from the actual runs.

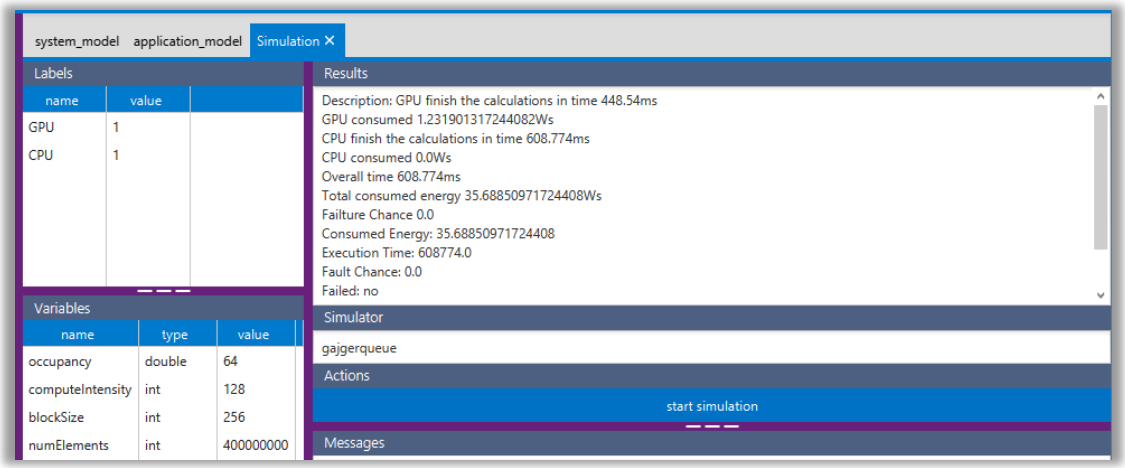

Figure 5.24. Simulation launch screen

CHAPTER 6. TESTS

In this chapter, we analyze the solution that was implemented, for this purpose we have prepared a simple CUDA application that allocates the data, copies it from host to the device, calls saxpy2[63] kernel and then fetches the results back into the host memory. Since the behavior of the saxpy procedure depends only on the data size and not its layout and we only want to measure the time needed to perform memory copies and execute the kernel, the results of the computations are not of our concern, thus we do not initialize the input arrays with any data as it is not meaningful in the scope of our tests. The code of the application used for tests is depicted in Figure 6.1.

```
1. void simpleCUDAApp(int arithmeticIntensity, int blockSize, int shMem, int N) {
2.   std::vector<float> x(N);
3.   std::vector<float> y(N);
4.   float *d_x, *d_y;
5.   cudaMalloc(&d_x, N*sizeof(float));
6.   cudaMalloc(&d_y, N*sizeof(float));
7.
8.   {
9.      SimpleTimer timer("Entire application");
10.     {
11.        SimpleTimer timer("memcpy HtD");
12.        cudaMemcpy(d_x, x.data(), N*sizeof(float), cudaMemcpyHostToDevice);
13.        cudaMemcpy(d_y, y.data(), N*sizeof(float), cudaMemcpyHostToDevice);
14.     }
15.
16.     {
17.        SimpleTimer timer("kernel");
18.        const int gridSize = (N+blockSize-1)/blockSize;
19.        saxpy2<<<gridSize, blockSize, shMem>>>(N, arithmeticIntensity, d_x, d_y);
20.     }
21.
22.     {
23.        SimpleTimer timer("memcpy DtH");
24.        cudaMemcpy(y.data(), d_y, N*sizeof(float), cudaMemcpyDeviceToHost);
25.     }
26.  }
27.
28.  cudaFree(d_x);
29.  cudaFree(d_y);
30.}
```

Figure 6.1. Source code of a simple CUDA application used for tests

The tests were performed using two testbeds, one with GTX 970 and PCIe v3.1 16x, and the second with GTX TITAN X (Maxwell edition) and PCIe v2.0 4x. The model was calibrated for the former setup and the kernel execution lambda was found to equal approximately 0.703787, it also turned out that the value of the lambda did not change for the second GPU. This is an expected behavior since both devices are of the same architecture (compute capability), which proves the correctness of the model. The devices differ in number of SMs, SM clock and in terms of memory system, with GTX TITAN X having wider memory bus, the differences and similarities are summarized in Table 6.1.

[63] Source code is listed in Figure 5.1.

Table 6.1. Testbed GPUs

	GTX 970	GTX TITAN X (Maxwell)
SMs	13	24
SM clock [MHz]	1253	1076
Compute capability	5.2	
Warp size	32	
Schedulers	4	
CUDA cores	128	
Memory clock [MHz]	1753	
Bus width [bits]	256	384
Data rate	4 (GDDR5)	
Global memory throughput per SM [bytes / cycle]	13.78	13.03
Kernel execution lambda	0.703787	

We have performed numerous tests for various configurations to verify accuracy of the model in four scenarios where different parameters change. In the first case, all parameters but the compute intensity were constant, second one included modification of the block size in addition to compute intensity, third scenario concerned varying occupancy and the fourth changing input data size. In the first three scenarios, we measured only the kernel execution time as the data size was constant and memory copying could be omitted. In the last one however, we included the measurements of the data transfer times in both directions, as well as the execution time of the entire application. All measurements in this chapter were repeated 10 times and then a mean value was used.

6.1. Varying block size and compute intensity

In first of the test scenarios we investigate the effect of a varying compute intensity of the kernel on its execution time. We have measured and simulated execution times for intensities spanning from 1 to 4096, while keeping all of the other parameters constant. This test verifies whether our theoretical model works for kernels exhibiting a different ratio of computations to communication. For *saxpy2* running with a maximum achievable occupancy of 64 warps per SM, if the intensity is low, then the kernel is bound by the throughput of a global memory, if it is high enough (32 in our case) it is bound by the CUDA cores throughput. The results for GTX 970 and GTX TITAN X are shown in Figure 6.2 and Figure 6.3 respectively.

We see on these figures that for both devices the model accurately determines the kernel execution time, however for lower compute intensities it is noticeably less precise as the actual measured values diverge more from the predictions. This is even better visible on the second set of charts, which include a varying block size and show a relative estimation error of our model, these are shown in Figure 6.4 and Figure 6.5. For low intensities the error reaches up to 53% and 18%, depending on the device, whilst for higher ones it is very low. This shows that the model handles throughput bound scenario very well when the bound is imposed by the CUDA cores but falls short when it is due to the global memory system, although the prediction is still within the acceptable limits especially for GTX TITAN X. Furthermore, the increased

estimation error for a memory throughput bound scenario is not surprising as we have decided to take a very simplified approach to the global memory modeling, by not considering the gradual saturation effect, using an equation to determine the theoretical memory throughput, and not adjusting it to the actual hardware characteristics. Based on these charts, we can also conclude that our model is capable of handling degenerated scenarios, where block size is lower than 32 and the warp is not fully populated with threads.

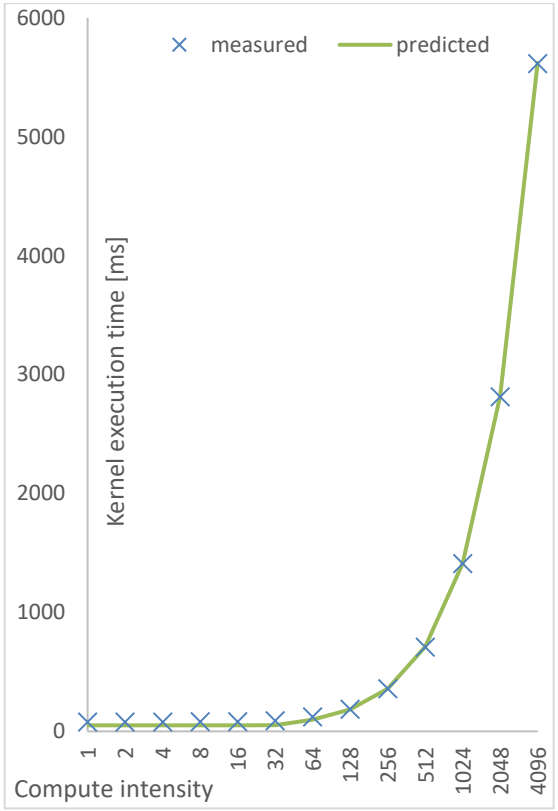

Figure 6.2. Measured and predicted kernel execution time in function of the compute intensity for GTX 970

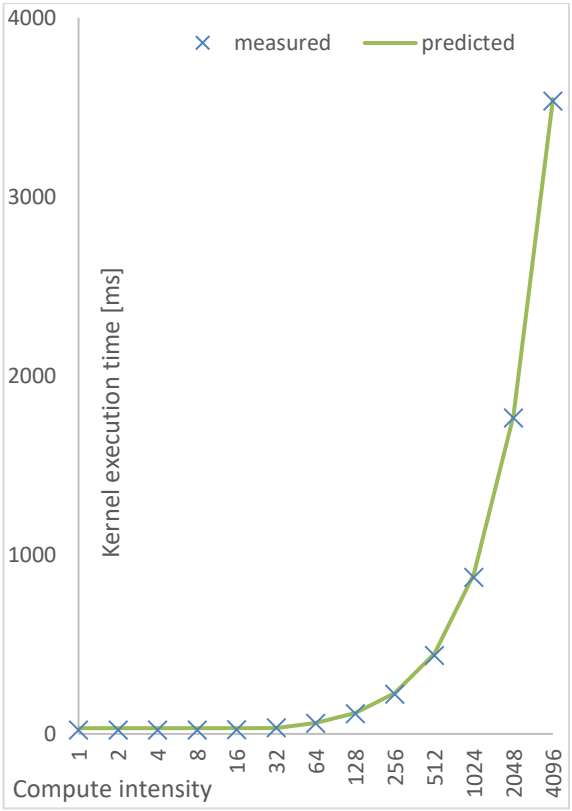

Figure 6.3. Measured and predicted kernel execution time in function of the compute intensity for GTX TITAN X

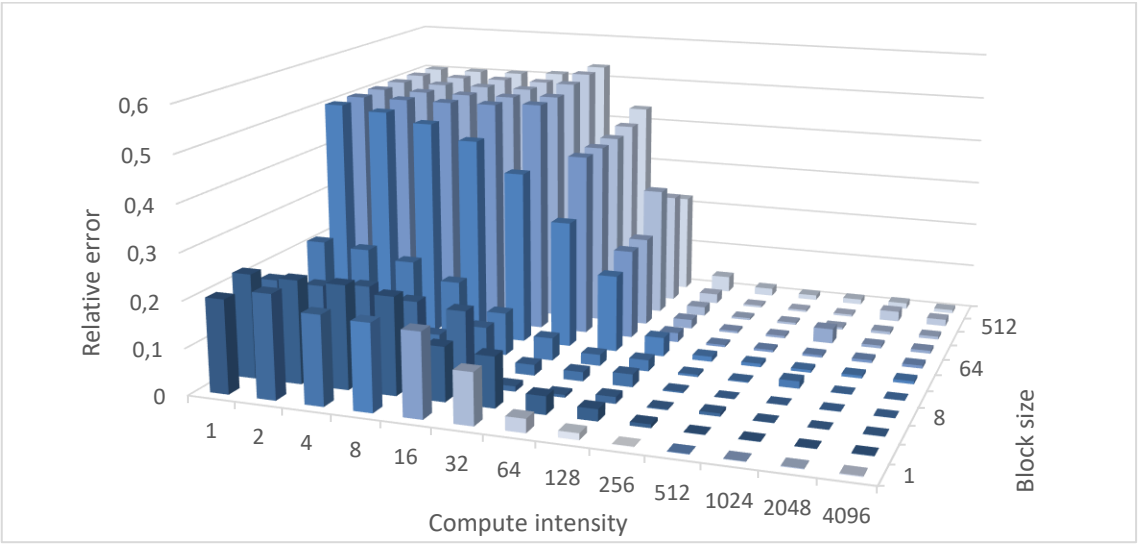

Figure 6.4. Relative prediction error of the model in function of the compute intensity and the block size for GTX 970

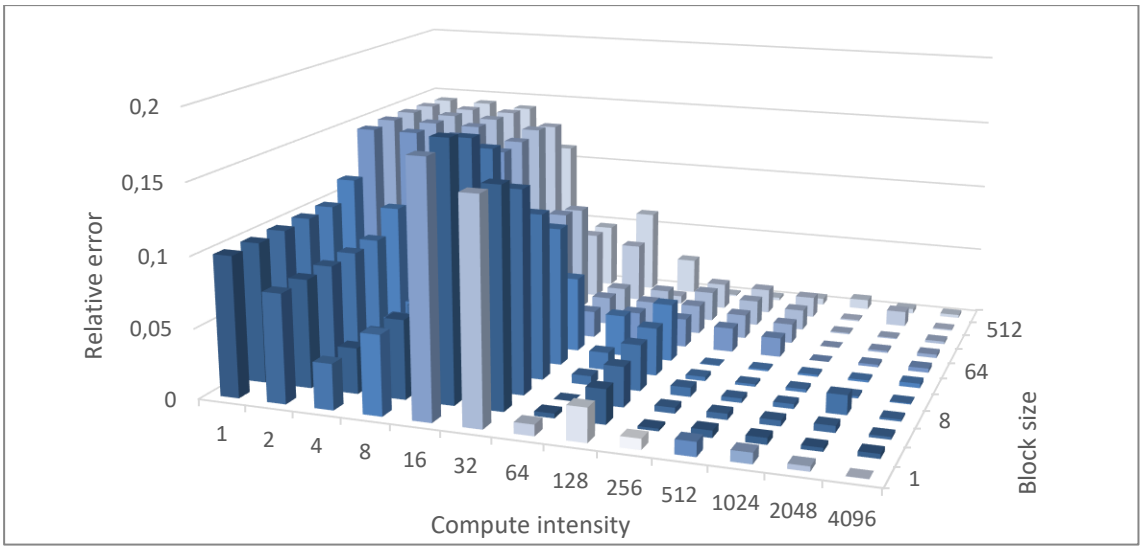

Figure 6.5. Relative prediction error of the model in function of the compute intensity and the block size for GTX TITAN X

6.2. Varying occupancy

Third test case verifies whether our implementation of the model handles two performance modes – latency and throughput bound. To address both, we adjust the occupancy achieved when executing the kernel. To control the occupancy without modifying the block size we allocate dummy shared memory that effectively limits the maximum number of blocks that may be assigned to a SM. Note that it is not technically possible to cover every single value of the occupancy due to the fact how block and warp allocation works but nevertheless we were able to address a wide range of occupancies from 2 warps per SM to a maximum of 64.

Figure 6.6 shows the test results for both devices, the model precisely determines the kernel execution time with an average relative error of 5.4% and 7.8% for GTX 970 and GTX TITAN X respectively. Furthermore, the transition from latency bound to throughput bound mode is represented accurately as well. However, for very small values of the occupancy the model overestimates the execution time by a small factor. Based on these charts we can also easily determine the occupancy at which the transition occurs, this is around 32 where rest of the chart becomes flat.

Various block sizes were used when performing this test and it was noticed that the block size had no significant effect on the kernel execution time and only thing that counted was the occupancy. The validity of this conclusion is confirmed by the fact that what matters is the effectiveness of the utilization of SM resources, which is directly related to achieved occupancy and not to the block size. The block size can only have an indirect effect by affecting the occupancy, which was not the case here once we excluded degenerated scenarios, where the size was smaller than 32. Our model is also based on calculation of the SM resources utilization, therefore it behaved correctly in this scenario.

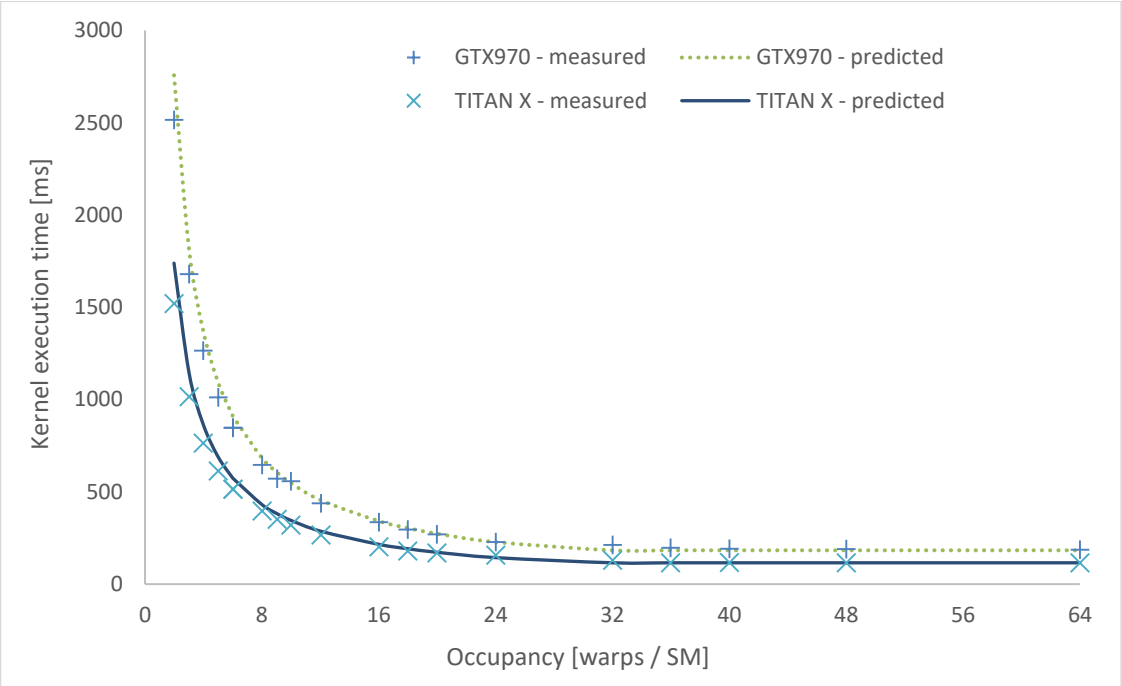

Figure 6.6. Measured and predicted kernel execution time in function of occupancy for both devices

6.3. Varying data size

At first, we look at the kernel execution time. The charts showing the execution time in function of data size are depicted in Figure 6.7 and Figure 6.8, the former for small data sizes and the latter for large ones. The model is less accurate when the data size is small and improves its accuracy as the number of elements to process increases. Important is the fact that even for the overestimated results, the functional relation between the data size and the execution time is maintained. This is acceptable since in real world scenarios the computations are performed for large data sizes and a case where the data is relatively small may be considered a degenerated one. Furthermore, the model was calibrated for a scenario with hundreds of thousands of elements, so its most accurate exactly where it should be.

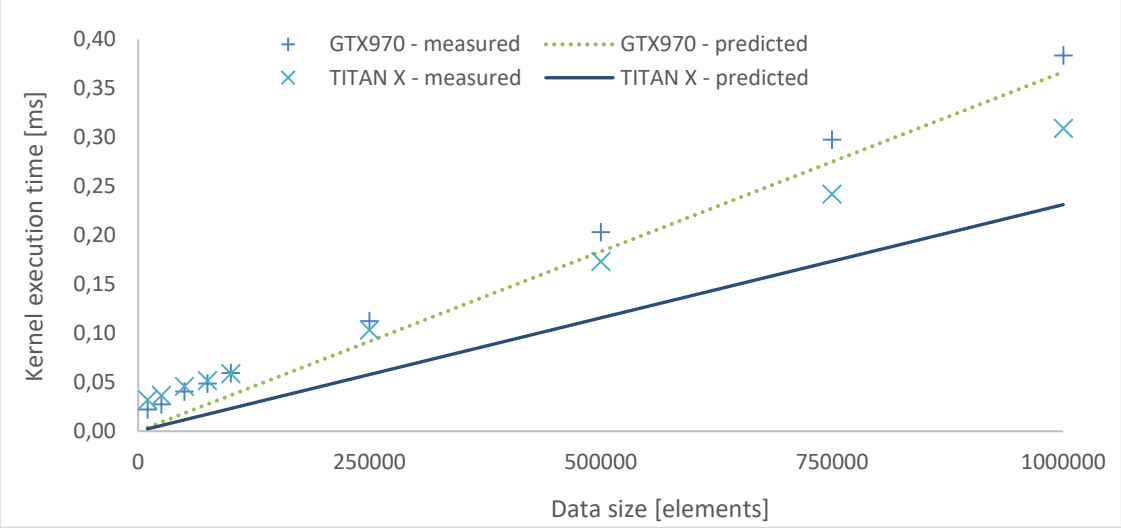

Figure 6.7. Measured and predicted kernel execution time for small data sizes on both devices

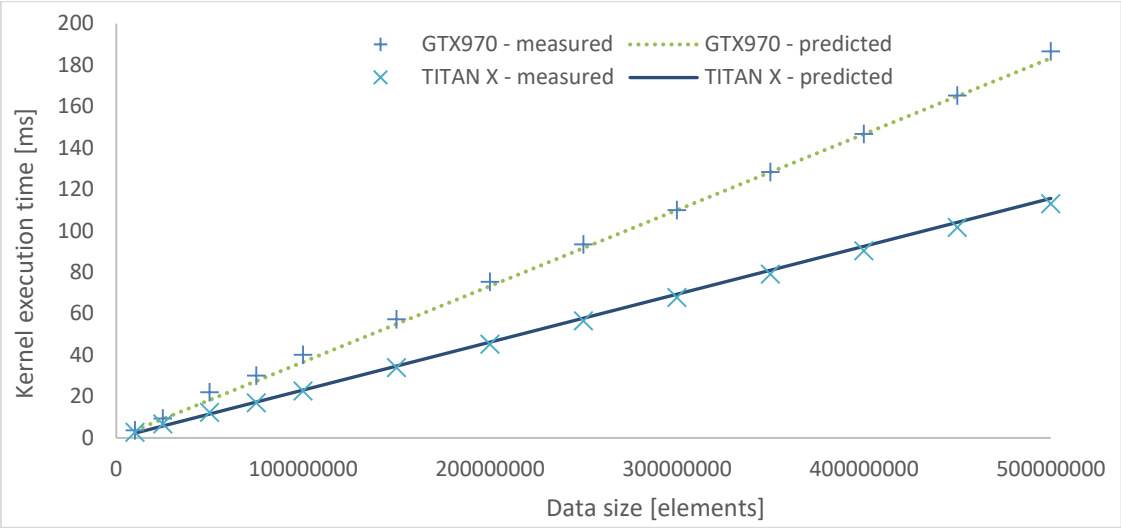

Figure 6.8. Measured and predicted kernel execution time for large data sizes on both devices

Since this test concerns varying data size it must include data transfer time estimation, unlike the previous ones which focused only on the kernel execution. Given that our testbeds used different configurations of the PCIe bus we measured the lambda and startup parameters separately for each of them. The results are presented in Table 6.2, these were then used as the parameters of the hardware unit, which are substituted to equation 5.5 in the computational model.

Table 6.2. PCIe bus characteristics for both testbeds

	GTX 970 (3.1 x16)	GTX TITAN X (2.0 x4)
Startup HtD[64] [ms]	0.00396868	0.0073276
Startup DtH [ms]	0.00515692	0.01167905
Lambda HtD	0.689	0.8435
Lambda DtH	0.653	0.8421
Bandwidth [GB/s]	15.8	2

Measured and estimated data transfer times are shown in Figure 6.9 and Figure 6.10. A decrease in accuracy for smaller data sizes because of an overestimation can be noticed, this is the same observation as in case of the kernel execution time. Note that the transfers from host to the device take twice as much time because there are two arrays of N elements to be transferred in this direction, compared to only a single array that is transferred back. We also experimentally prove here what was pointed out in section 5.4.2, that the lambda and startup parameters assume different values depending on the transfer direction. If this effect was not considered, then the results for both testbeds would be less precise. For large data sizes the predicted values very closely resemble the measured ones, thus we may conclude that the methodology proposed for measuring data transfer times is valid.

[64] HtD stands for “Host to Device” transfer direction, while DtH for “Device to Host”.

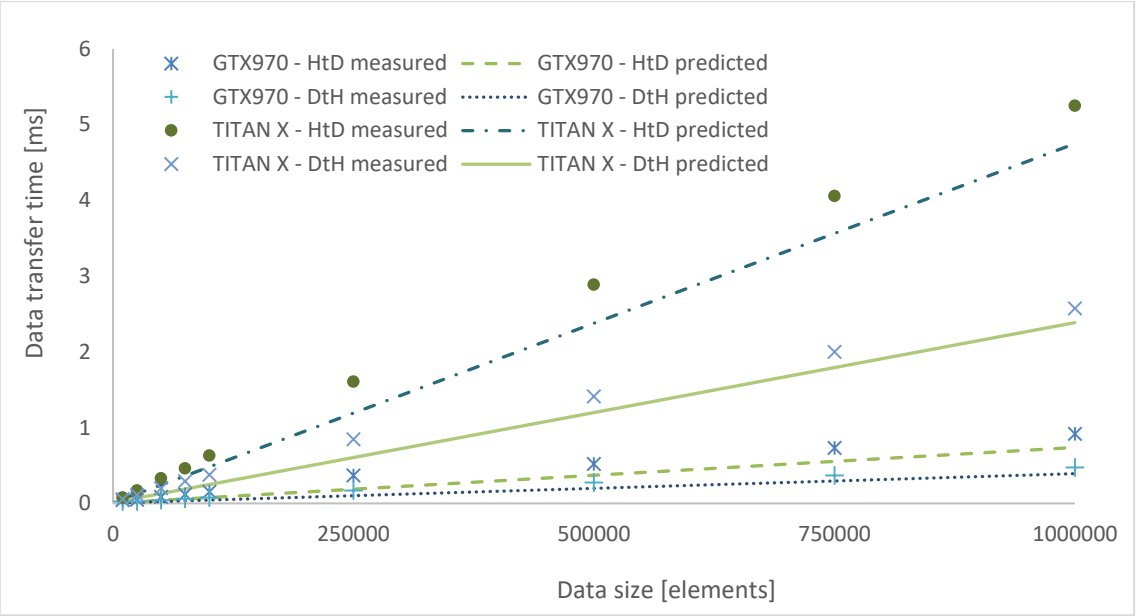

Figure 6.9. Measured and predicted data transfer times for small data sizes for both testbeds

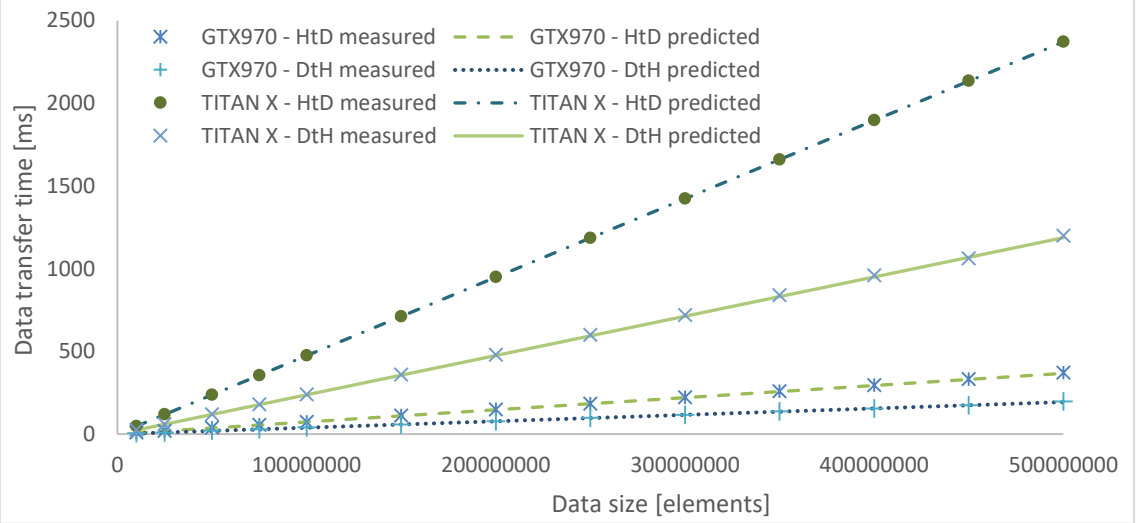

Figure 6.10. Measured and predicted data transfer times for large data sizes for both testbeds

Finally, we present another pair of charts - Figure 6.11 and Figure 6.12, these two represent the total running time of the application. Technically, these sum up the results from the previous charts presented earlier in this section, since what comprises the application execution time is the data transfer in both directions and the kernel execution. These two also conclude our work and prove its correctness by showing that the model we have implemented is capable of accurate prediction of the application execution time for the most essential scenario, achieving 1.8% (GTX 970) and 0.5% (GTX TITAN X) mean relative error for large data size (≥10000000)[65]. From an end user perspective, parameters like occupancy, block size and compute intensity will likely be determined once and then kept constant. What is probably the most common real-world use case, is verification of the application on different hardware setups

[65] For small data size (≤1000000) these equal 48.7% and 33.2% respectively due to noticeable overestimation and low execution time, which amplifies this effect.

and for different data sizes. Our model allows the users to assess the behavior of their applications for data sizes exceeding the hardware capabilities of units available to them and even use ones that are not in their possession, given that these are available in MERPSYS' database.

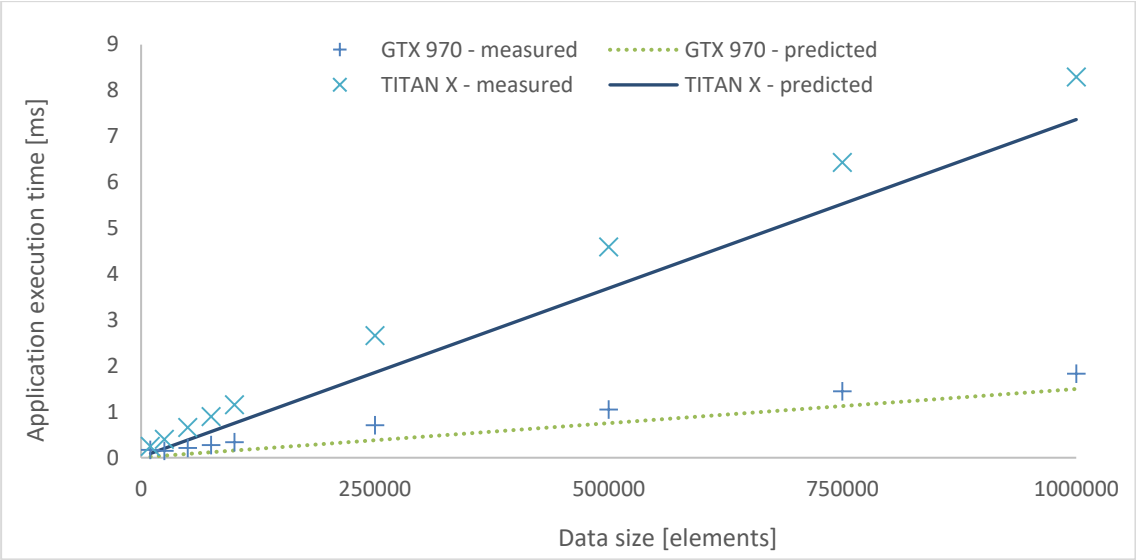

Figure 6.11. Measured and predicted application execution times for small data sizes for both testbeds

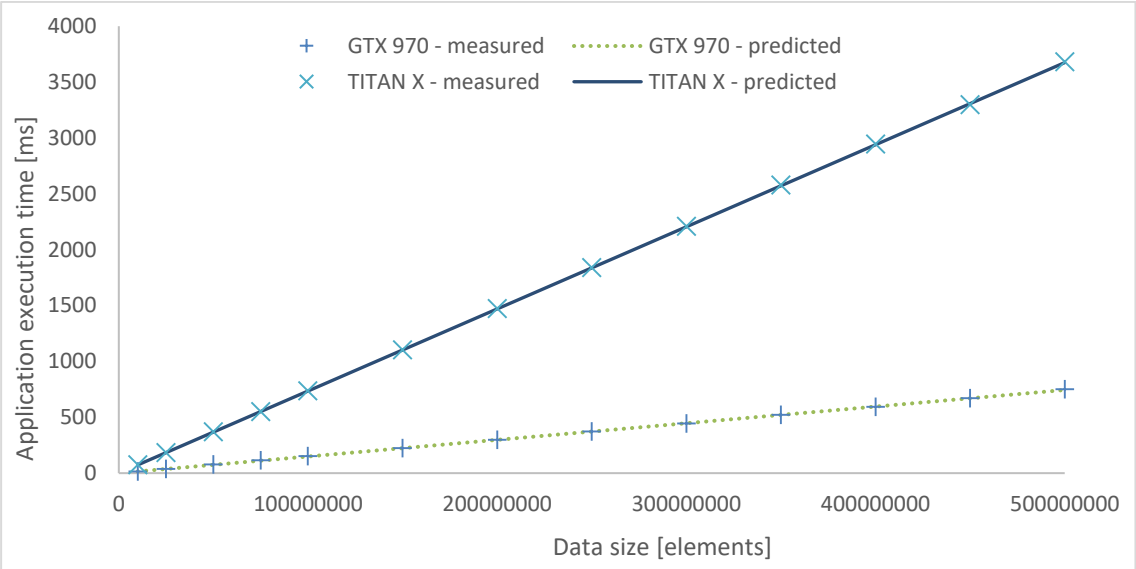

Figure 6.12. Measured and predicted application execution times for large data sizes for both testbeds

CHAPTER 7. CONCLUSIONS

The goal of this work was to design, implement and verify a performance model for a GPU using the MERPSYS platform. In Chapter 4 we have described our adaptation of an accurate and simple performance model proposed by Volkov and then in Chapter 5 we thoroughly documented the implementation process. In Chapter 6 we have verified its correctness for various launch configurations and kernel characteristics on two testbeds. By adjusting the model using a scaling parameter, we have obtained accurate predictions of the application running time for all input configurations. For the most essential scenario with a varying data size, when this size was not unnaturally small, the mean values of the error were 1.8% for GTX 970 and 0.5% for GTX TITAN X. Tests of a compute bound kernel with varying occupancy yielded an average prediction error of 5.4% and 7.8% respectively, what is also a very good result. However, the model was noticeably less precise when the input data size was small or the kernel was bound by a memory throughput, in latter case the relative prediction error varied from 1% to 53% with a mean value of 32.8% for GTX 970 and 1% to 18% (mean 9.9%) for GTX TITAN X. For compute throughput bound kernels, the error went down to a few percent depending on the launch configuration. It should be further noted that for every test case the functional relationship between the launch configuration parameter being investigated and the application running time was very closely reproduced by the model.

Aside from achieving very good results in the main area of focus, this work also makes a whole another set of important contributions. Starting with a brief history and rationale for the importance of the GPGPU paradigm, which transitions smoothly into a detailed, but still comprehensible description of the CUDA GPU architecture and processing model. We also show how to make use of various tools and libraries from the CUDA development framework to: write a simple kernel, compile and disassemble it, measure its running time in both cycles and seconds, and analyze it using a profiler to extract statistics of interest and inspect the application execution timeline. Additionally, an insight into the MERPSYS platform is provided, both from user's and developer's perspective by showing not only how to use the simulator, but also exposing the high-level system components, relations between them, and the underlying internal architecture.

We have reviewed related work in the fields of large scale systems and GPU modeling, for the former category a tabular comparison was presented. Regarding the latter, we considered several different approaches, then by compiling the results of various research papers we have described a theoretical GPU performance model and backed it up with all the data necessary for its parametrization during the actual implementation. The model was thoroughly documented and explained, mathematical equations were provided where needed and for each building block of the model a practical example was given. Furthermore, the proposed solution is not limited to the kernel modeling, but additionally includes an accurate formula for the data transfers. We have also shown in detail how to incorporate such model within the MERPSYS framework and extensively tested it using two devices representing a fairly recent Maxwell architecture.

When working on this thesis we have come to several important conclusions with the first one being that the design of the MERPSYS framework was well-thought and allows for an easy extension. Computational model functions are easily written using JavaScript and addition of new hardware units is effortless. Nevertheless, we still had to modify the source code of the editor and simulator applications in order to pass the additional parameters required by our GPU performance model. What turned out to be problematic was preparation of the environment for development purposes, compilation process of the framework is not entirely intuitive and there were some missing pieces which had to be addressed. Books [2] and [26] were useful when resolving the issues and helped to understand the framework architecture.

What is more, it became obvious to us that the architecture of the GPU, despite being very complex and highly parallel, is suitable for the performance modeling once a proper theoretical model is developed. Such a model still has to make some simplifying assumptions but as Volkov and this work based on his had shown, with a proper model the results may be very good or even excellent to some extent. CUDA as an architecture is now very mature, has large and dedicated community, good and precise documentation[66], and offers an extensive set of sophisticated tools that are of tremendous help when designing, developing and analyzing parallel applications running on a GPU.

7.1. Future work

While being self-contained, this work is by no means fully complete and there are numerous extensions which could be implemented. First, the scope of the modeling was substantially limited as we have omitted double precision units, SFUs and shared memory, all of these could be fairly easy added to the model. Moreover, the approach to modeling of global memory accesses was very simplified and did not consider caches, gradual saturation effect and varying access time depending on the transfer direction. We have also decided to target our model at the CUDA and NVIDIA GPUs, possible extension includes its generalization to be applicable to units produced by AMD and the OpenCL framework. The model does not include a formula for occupancy calculation, which could be incorporated into the existing set of the equations and hence remove the need of relying on an external tool for this purpose.

The solution would also largely benefit from an automation of the kernel analysis process. Currently the throughput limits and the kernel execution graph, hence the latency bound must be determined manually, this could be automated by developing a dedicated tool that parses the SASS of a kernel and outputs the data mentioned.

Lastly, the model required a scaling parameter, whilst according to the initial assumptions it should be accurate without it. This is a subject to further investigation and furthermore, the behavior of the model could also be verified on an architecture different than Maxwell, Pascal for example.

[66] However, it makes some simplifications as in case of high occupancy being required for an optimal utilization of the GPU's resources, which was proven by the Volkov to be a misleading and incorrect statement, since a peak utilization may be achieved even for low occupancies by employing ILP.

REFERENCES

1. T. Gajger, A. Podgórski and B. Pollok, *A module for management of energy consumption for a heterogeneous HPC system,* Gdańsk, Poland: WETI PG, 2016.
2. P. Czarnul, Ed., Modeling Large-Scale Computing Systems. Practical Approaches in MERPSYS, Gdańsk, Poland: Gdańsk University of Technology, 2016.
3. K. E. Sanders J., Cuda by example: An Introduction to General-Purpose GPU Programming, Ann Arbor, Michigan, USA: Addison-Wesley, 2011.
4. D. B. Kirk and W.-m. W. Hwu, Programming Massively Parallel Processors. A Hands-on Approach, Burlington, MA, USA: Morgan Kaufmann Publishers, 2010.
5. J. Cheng, M. Grossman and T. McKercher, Professional CUDA C Programming, Indianapolis, IN, USA: John Wiley & Sons, 2014.
6. O. Maitre, "Understanding NVIDIA GPGPU Hardware," in *Massively Parallel Evolutionary Computation on GPGPUs*, Berlin, Springer-Verlag, 2013, pp. 15-34.
7. NVIDIA Corporation, "Parallel Thread Execution ISA," 20 March 2017. [Online]. Available: http://docs.nvidia.com/cuda/parallel-thread-execution/. [Accessed March 2017].
8. NVIDIA Corporation, "CUDA C Programming Guide," 23 June 2017. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. [Accessed June 2017].
9. NVIDIA Corporation, *NVIDIA Tesla P100 The Most Advanced Datacenter Accelerator Ever Built, Featuring Pascal GP100, the World's Fastest GPU,* 2016.
10. N. Wilt, The CUDA Handbook. A Comprehensive guide to GPU Programming, Crawfordsville, Indiana, USA: Addison-Wesley, 2013.
11. T. T. Dao, J. Kim, S. Seo, B. Egger and J. Lee, "A Performance Model for GPUs with Caches," *IEEE Transactions on Parallel and Distributed Systems,* vol. 26, no. 7, pp. 1800-1813, 2014.
12. T. Scudiero, "Memory Bandwidth Bootcamp: Best Practices," in *GPU technology conference*, 2015.
13. T. Scudiero, "Memory Bandwidth Bootcamp: Beyond Best Practices," in *GPU technology conference*, 2015.
14. A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *International Symposium on Performance Analysis of Systems and Software*, Boston, MA, USA, 2009.
15. V. Volkov, "Better Performance at Lower Occupancy," in *GPU technology conference*, 2010.
16. X. Mei and X. Chu, "Dissecting GPU Memory Hierarchy through Microbenchmarking," *IEEE Transactions on Parallel and Distributed Systems,* vol. 28, no. 1, pp. 72-86, 2016.
17. NVIDIA Corporation, "CUDA C Best Practices Guide," 20 March 2017. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/. [Accessed March 2017].
18. NVIDIA Corporation, *NVIDIA's Next Generation CUDA Compute Architecture: Fermi,* 2009.

19. NVIDIA Corporation, *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110,* 2013.
20. NVIDIA Corporation, *NVIDIA GeForce GTX 980 Featuring Maxwell, The Most Advanced GPU Ever Made.,* 2014.
21. L. Durant, M. Harris, N. Stam and O. Giroux, "Inside Volta: The World's Most Advanced Data Center GPU," 10 May 2017. [Online]. Available: https://devblogs.nvidia.com/parallelforall/inside-volta/. [Accessed May 2017].
22. M. Harris, "CUDA 9 Features Revealed: Volta, Cooperative Groups and More," 11 May 2017. [Online]. Available: https://devblogs.nvidia.com/parallelforall/cuda-9-features-revealed/. [Accessed May 2017].
23. NVIDIA Corporation, "Nvidia CUDA Compiler Driver NVCC," 20 March 2017. [Online]. Available: http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/#gpu-compilation. [Accessed March 2017].
24. Khronos OpenCL Working Group, "The OpenCL Specification," 11 March 2016. [Online]. Available: https://www.khronos.org/registry/OpenCL/specs/opencl-2.2.pdf. [Accessed March 2017].
25. NVIDIA Corporation, "OpenCL Programming Guide for the CUDA Architecture," 31 August 2009. [Online]. Available: http://www.nvidia.com/content/cudazone/download/opencl/nvidia_opencl_programmingoverview.pdf. [Accessed March 2017].
26. P. Czarnul, Ed., Modeling Large-Scale Computing Systems. Concepts and Models, Gdańsk, Poland: Gdańsk University of Technology, 2013.
27. P. Czarnul, J. Kuchta, P. Rościszewski and J. Proficz, "Modeling energy consumption of parallel applications," in *Federated Conference on Computer Science and Information Systems (FedCSIS)*, Gdańsk, Poland, 2016.
28. P. Czarnul, J. Kuchta, M. Matuszek, J. Proficz, P. Rościszewski, M. Wójcik and J. Szymański, "MERPSYS: An environment for simulation of parallel application execution on large scale HPC systems," *Simulation Modelling Practice and Theory,* vol. 77, pp. 124-140, 2017.
29. A. Sulistio, C. S. Yeo and R. Buyya, "A taxonomy of computer-based simulations and its mapping to parallel and distributed systems simulation tools," *Software: Practice and Experience,* vol. 34, no. 7, pp. 653-673, 2004.
30. R. Buyya and M. Murshed, "GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing," *Concurrency and Computation: Practice and Experience,* vol. 14, no. 13-15, pp. 1175-1220, 2002.
31. H. Casanova, A. Legrand and M. Quinson, "SimGrid: A Generic Framework for Large-Scale Distributed Experiments," in *Ninth International Conference on Peer-to-Peer Computing*, Cambridge, UK, 2008.
32. M. Quinson, "Experimenting HPC Systems with Simulation," in *High Performance Computing & Simulation Conference*, Caen, 2010.
33. B. Donassolo, H. Casanova, A. Legrand and P. Velho, "Fast and Scalable Simulation of Volunteer

Computing Systems Using SimGrid," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, New York, NY, USA, 2010.

34. R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose and R. Buyaa, "CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software - Practice & Experience,* vol. 41, no. 1, pp. 23-50, 2011.
35. S. F. Piraghaj, A. V. Dastjerdi, R. N. Calheiros and R. Buyya, "ContainerCloudSim: An Environment for Modeling and Simulation of Containers in Cloud Data Centers," *Software: Practice and Experience,* vol. 47, no. 4, pp. 505-521, 2016.
36. K. Kurowski, J. Nabrzyski, A. Oleksiak and J. Węglarz, "GSSIM – Grid Scheduling Simulator," *Computational Methods in Science and Technology,* vol. 13, no. 2, pp. 121-129, 2007.
37. S. Bąk, M. Krystek, K. Kurowski, A. Oleksiak, W. Piątek and J. Wąglarz, "GSSIM - a tool for distributed computing experiments," *Scientific Programming,* vol. 19, no. 4, pp. 231-251, 2011.
38. W. E. Denzel, J. Li, P. Walker and Y. Jin, "A framework for end-to-end simulation of high-performance computing systems," in *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, Marseille, France, 2008.
39. S. Madougou, A. Varbanescu, C. de Laat and R. van Nieuwpoort, "The Landscape of GPGPU Performance Modeling Tools," *Parallel Computing,* vol. 56, no. C, pp. 18-33, 2016.
40. L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM,* vol. 33, no. 8, pp. 103-111, 1990.
41. M. Amaris, D. Cordeiro, A. Goldman and R. Y. de Camargo, "A Simple BSP-based Model to Predict Execution Time in GPU Applications," in *22nd International Conference on High Performance Computing (HiPC)*, Bangalore, India, 2015.
42. S. Hong and H. Kim, "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness," in *Proceedings of the 36th annual international symposium on Computer architecture*, Austin, TX, USA, 2099.
43. S. Hong and H. Kim, "An integrated GPU power and performance model," in *Proceedings of the 37th annual international symposium on Computer architecture*, Saint-Malo, France, 2010.
44. S. S. Baghsorkhi, M. Delahaye, S. J. PAtel, W. D. Gropp and W.-m. W. Hwu, "An Adaptive Performance Modeling Tool for GPU Architectures," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Bangalore, India, 2010.
45. Y. Zhang and J. D. Owens, "A Quantitative Performance Analysis Model for GPU Architectures," in *17th International Symposium on High Performance Computer Architecture*, San Antonio, TX, USA, 2011.
46. A. Kerr, G. Diamos and S. Yalamanchili, "A Characterization and Analysis of PTX Kernels," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization*, Austin, TX, USA, 2009.

47. A. Kerr, G. Diamos and S. Yalamanchili, "Modeling GPU-CPU Workloads and Systems," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, Pittsburgh, Pennsylvania, USA, 2010.
48. J.-C. Huang, J. H. Lee, H. Kim and H.-H. S. Lee, "GPUMech: GPU Performance Modeling Technique Based on Interval Analysis," in *47th Annual International Symposium on Microarchitecture*, Cambridge, United Kingdom, 2014.
49. V. Volkov, "Understanding Latency Hiding on GPUs," Berkeley, 2016.
50. M. Amaris, R. Y. de Camargo, M. Dyab, A. Goldman and D. Trystram, "A Comparison of GPU Execution Time Prediction using Machine Learning and Analytical Modeling," in *15th International Symposium on Network Computing and Applications (NCA)*, Cambridge, MA, USA, 2016.
51. NVIDIA Corporation, "CUDA Occupancy Calculator," December 2016. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls. [Accessed May 2017].
52. Y. Liang, M. T. Satria, K. Rupnow and D. Chen, "An Accurate GPU Performance Model for Effective Control Flow Divergence Optimization," in *26th International Parallel and Distributed Processing Symposium*, Shanghai, China, 2012.
53. A. K. Parakh, M. Balakrishnan and K. Paul, "Performance Estimation of GPUs with Cache," in *26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, Shanghai, China, 2012.
54. M. Andersch, J. Lucas, M. Alvarez-Mesa and B. Juurlink, "Analyzing GPGPU Pipeline Latency," Fiuggi, Italy, 2014.
55. NVIDIA Corporation, "CUDA Binary Utilities," 20 March 2017. [Online]. Available: http://docs.nvidia.com/cuda/cuda-binary-utilities/. [Accessed April 2017].
56. Micron Technology, "GDDR5 SGRAM Introduction," 2014.
57. M. Harris, "An Easy Introduction to CUDA C and C++," 31 October 2012. [Online]. Available: https://devblogs.nvidia.com/parallelforall/easy-introduction-cuda-c-and-c/. [Accessed April 2017].
58. N. Bombieri, F. Busato, F. Fummi and M. Scala, "MIPP: A Microbenchmark Suite for Performance, Power, and Energy Consumption Characterization of GPU architectures," in *11th Symposium on Industrial Embedded Systems (SIES)*, Krakow, Poland, 2016.
59. H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi and A. Moshovos, "Demystifying GPU Microarchitecture through microbenchmarking," in *International Symposium on Performance Analysis of Systems & Software (ISPASS)*, White Plains, NY, USA, 2010.
60. NVIDIA Corporation, "nvidia-smi - NVIDIA System Management Interface," 26 July 2016. [Online]. Available: http://developer.download.nvidia.com/compute/DCGM/docs/nvidia-smi-367.38.pdf. [Accessed June 2017].

LIST OF FIGURES

LIST OF TABLES

This page intentionally left blank.

SUPPLEMENT A – STRESZCZENIE ROZSZERZONE

Tytuł pracy: Modelowanie przetwarzania równoległego z wykorzystaniem GPU wraz z weryfikacją z wykorzystaniem platformy MERPSYS.

W niniejszej pracy dokonano oceny modelu wydajnościowego dla GPU, który bazując na prawie Little'a w analityczny sposób wyraża czas wykonania kernela jako zależny od ograniczeń wynikających z opóźnień i przepustowości, oraz osiągniętej zajętości zasobów. W celu wyboru modelu dokonano analizy rozwiązań z dwóch obszarów: symulatorów ogólnego przeznaczenia i modeli wydajnościowych dla GPU. W toku analizy zidentyfikowany został model spełniający wszystkie zakładane kryteria, takie jak: prostota, dokładność szacowanych wyników, trafność odwzorowania elementów składających się na aplikację równoległą wykonywaną na GPU oraz jasna dekompozycja modeli sprzętu i aplikacji.

Następnie model ten połączono z rezultatami kilku innych prac naukowych z tejże dziedziny, tak aby zawrzeć w pracy elementy niezbędne do jego parametryzacji, a w szczególności czasy opóźnień dla poszczególnych instrukcji kodu maszynowego wykonywanych na GPU. Dodano też równania szacujące czas przesyłania danych wraz z niezbędnymi parametrami i opisem metodologii wykorzystanej do ich pozyskania. Model włączono jako element do platformy MERPSYS, która jest symulatorem ogólnego przeznaczenia dla systemów równoległych i rozproszonych. Dla wszystkich równań opisujących model przygotowane zostały wyczerpujące opisy poparte przykładami, które wyjaśniają ich poszczególne składowe. Opracowane rozwiązanie pozwala użytkownikowi na utworzenie opisu aplikacji CUDA w edytorze MERPSYS z wykorzystaniem rozszerzonego języka Java, a następnie przeprowadzenie w wygodny sposób oceny wydajności dla różnorodnych konfiguracji uruchomieniowych. Do symulacji wykorzystać można różne urządzenia. Możliwe jest również przekroczenie rzeczywistych ograniczeń nakładanych przez sprzęt, przykładowo ilości pamięci dostępnej na hoście bądź urządzeniu, czy liczby połączonych ze sobą kart graficznych. Cały proces implementacyjny poczynając od napisania prostego kernela CUDA, jego kompilacji i dekompilacji, analizy kodu maszynowego, modyfikacji platformy MERPSYS i zdefiniowania w jej ramach modelu, został drobiazgowo opisany. Jednocześnie praca ta stanowi praktyczny opis wykorzystania i dostosowywania do swoich potrzeb platformy MERPSYS, tak więc może być wykorzystana jako samouczek przez osoby chcące zapoznać się z tym rozwiązaniem.

Dodatkowo opisanych zostało wiele aspektów związanych z przetwarzaniem na GPU, takich jak architektura i model programistyczny CUDA, hierarchia pamięci, przebieg wykonania kernela, wytwarzanie i analiza aplikacji CUDA, oraz rozwój kolejnych edycji GPU i różnic pomiędzy nimi. Opisy zawarte w pracy pozwalają czytelnikowi niezaznajomionemu z architekturą CUDA na opanowanie jej w sposób wystarczający do zrozumienia jak przebiega wykonanie programu oraz do stworzenia własnej, prostej aplikacji. W pracy znajdują się również liczne przypisy odsyłające do znacznie bardziej szczegółowych opisów poszczególnych zagadnień. Stanowi ona więc punkt odniesienia dla dalszej nauki i pogłębiania wiedzy w zakresie praktycznych zastosowań przetwarzania równoległego z wykorzystaniem GPU.

Przedstawiono również systematyczną metodologię pozwalającą na wyznaczenie charakterystyk kernela, a następnie wykorzystanie ich jako parametry wejściowe modelu. W pracy zawarto również wiele informacji związanych z analizą aplikacji CUDA, między innymi: technik pozwalających na pomiar czasu wykonania w sekundach i cyklach zegara, modyfikacji kodu w celu usprawnienia analizy z wykorzystaniem *profilera,* pomiaru czasu przesyłania danych, szczegółowej analizy kodu maszynowego i tworzenia grafu wykonania kernela, określania mocy obliczeniowej GPU na podstawie architektury i liczby jednostek wykonawczych, szacowania zajętości zasobów na podstawie kodu źródłowego, i wiele innych.

Model został oceniony z wykorzystaniem kerneli wykazujących różne cechy, w szczególności różniące się stosunki czasu obliczeń do czasu komunikacji, gdzie przez czas komunikacji rozumiemy dostępy do pamięci globalnej GPU. Wykorzystano również wiele konfiguracji uruchomieniowych, w których zmienne były: rozmiar bloku, zajętość zasobów GPU (ang. *occupancy*) i rozmiar danych wejściowych. Autor ocenia model jako bardzo dokładny dla kerneli ograniczonych przez przepustowość obliczeniową i dla realistycznych zadań obliczeniowych, czyli takich gdzie rozmiar danych jest liczony w gigabajtach, a zajętość zasobów oscyluje w granicach stu procent. Dla kerneli ograniczonych przez przepustowość pamięci i zdegenerowanych scenariuszy z nierealistycznie małymi rozmiarami danych wyniki nie były już tak dobre, jednak wciąż mieściły się w zadowalających ramach. Dodatkowo dowiedziono przenośność modelu pomiędzy dwoma urządzeniami z tej samej architektury, ale o różniących się mocach obliczeniowych.

SUPPLEMENT B – EXTENDED ABSTRACT

Title of thesis: Modeling parallel processing with GPU and verification using the MERPSYS platform.

In this work, we evaluate an analytical GPU performance model based on Little's law, that expresses the kernel execution time in terms of latency bound, throughput bound, and achieved occupancy. We then combine it with the results of several other research papers from the field, so that the work contains all the elements needed for its parametrization, most notably the latencies of various instructions executed on the GPU. We also introduce equations for data transfer time estimation, together with required parameters and description of the methodology used to obtain them. The model was incorporated into the MERPSYS framework, which is a general-purpose simulator for parallel and distributed systems.

The resulting solution enables the user to express a CUDA application in MERPSYS editor using an extended Java language and then conveniently evaluate its performance for various launch configurations using different hardware units. An additional benefit is a possibility to exceed actual limits imposed by the hardware, e.g. amount of operating memory available on the host or the device, or the number of interconnected GPUs. The entire implementation process beginning with writing a simple CUDA kernel, its compilation and decompilation, assembly analysis, modifications of the MERPSYS framework and definition of the model within it, was meticulously documented. Additionally, since this work includes a detailed description of the usage and adjustment process of the MERPSYS platform, it may be used as a tutorial for people interested in this solution.

We also explain numerous aspects related to the GPU computing such as CUDA architecture and programming model, memory hierarchy, kernel execution process, development and analysis of CUDA applications, as well as evolution of subsequent editions of the GPUs and differences between them. Descriptions are very detailed, hence readers unfamiliar with CUDA architecture can understand it on a level allowing for a development of their own simple applications. Furthermore, we provide a systematic methodology for extracting kernel characteristics, that are used as input parameters of the model. The paper additionally contains a vast amount of information related to the analysis of CUDA applications, naming a few: kernel execution time measurement in seconds and clock cycles, enhancing the code for improved analysis under a profiler, data transfer time measurement, analysis of a machine code and kernel execution graph preparation, determining GPU processing power based on its architecture, estimation of the occupancy based on the source code analysis, and more.

The model was evaluated using kernels representing different traits, most importantly varying computations to communication ratio, where by communication time we refer to accesses to the GPU's global memory. We also used a large variety of launch configurations. We found the model to be very accurate for computation bound kernels and realistic workloads, whilst for memory throughput bound kernels and degenerated scenarios the results were not as good, but still within acceptable limits. We have also proven its portability between two devices of the same hardware architecture but different processing power.