2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS) | 979-8-3503-1958-3/23/$31.00 ©2023 IEEE | DOI: 10.1109/QRS60937.2023.00079

Modeling of GPGPU architectures for performance analysis of CUDA programs

Francesco Terrosi[1,*], Francesco Mariotti[1], Paolo Lollini[1], and Andrea Bondavalli[1]
[1]Università degli Studi di Firenze, Firenze, Italia
francesco.terrosi@unifi.it, francesco.mariotti@unifi.it, paolo.lollini@unifi.it, andrea.bondavalli@unifi.it
*corresponding author

Abstract—Graphics Processing Units (GPUs), originally developed for computer graphics, are now commonly used to accelerate parallel applications. Given that GPUs are designed to be as efficient as possible, evaluating their performance is crucial. This problem has been tackled in the last years by researchers that started to propose solutions such as analytical models and digital simulators, which are, however, often complex to use and/or to adapt to the needs of the user. Thanks to its high flexibility, model-based analysis is widely used to evaluate systems' properties, including performance. Researchers started working on developing GPU models that can represent both their architecture and the software in execution, but they often use strong assumptions that undermine their usability. In this work we develop a Stochastic Activity Network model to evaluate the performance of CUDA applications running on NVIDIA GPUs. The model takes as input a representation of the program's instruction, parsed from the CUDA SASS assembly file, and a list of parameters to offer configurability to the user. We tune our model to match the architecture of two different NVIDIA GPUs and simulate the execution of a CUDA program. We then compare the results with those obtained from the execution of the program over the real GPUs.

Keywords-
performance;modeling;analysis;GPU;CUDA;SAN

1. INTRODUCTION

Graphics Processing Units (GPUs) were initially developed to accelerate computations in computer graphics applications like 3D modeling, computer animation and videogames. However, over the years, they started to be used in other application fields. Thanks to their architectural design, GPUs provide great performance when dealing with problems that have a high level of parallelism in the computation. Many examples of the usage of GPGPUs can be found in domains like Machine Learning, Cryptography and Scientific Computing. The term General-Purpose Graphics Processing Units (GPGPUs) denotes the use of GPUs in applications that would be otherwise executed on CPUs. The structure and functioning of GPUs are quite complex with respect to CPUs, and they can change from one vendor to another. NVIDIA proposed the Compute Unified Device Architecture (CUDA) [1], a parallel computing platform to develop programs that can be executed on NVIDIA GPUs.

Performance evaluation of GPGPUs has gained much importance in the last years, due to the increasing use of these devices in computer systems coming from very different domains: from the "classic" data elaboration where GPUs are used to accelerate long computations on local data, to real-time systems in which GPUs are used to, e.g., elaborate very quickly images coming from a camera sensor [2], [3]. Both these application fields may benefit from a performance analysis activity before buying (or developing) a GPU, e.g., for the first case one could compare different architectural designs to estimate the execution time of these designs on the reference workload and then choose the less costly option; in the second case, system designers may also benefit from a prior performance analysis to check whether the target GPU is compliant with the system's timing requirements. Suppose that a GPU is going to be used as an image processor that receives data from a sensor camera at a fixed rate, e.g., 60 frames per second. A GPU model would allow one to understand if the target device can, e.g., process every frame received by the camera at the given rate. This problem has been tackled during the last years but the proposed solutions, although some of them are promising, are: i) complex to use and/or to adapt to the needs of the user (e.g., digital simulators, where understanding the model architecture may be time-consuming), or ii) hiding implementation details that may undermine the representation accuracy of the GPU [4].

In this work we present a stochastic model based on Stochastic Activity Networks (SANs) [5] that can be used to carry out performability analysis of CUDA programs executed on NVIDIA GPGPUs. Thanks to its high configurability, this model can represent different architectures, as it is possible to specify parameters such as the number of multiprocessors or the size of memories. The model takes as input the list of program's instructions, parsed from the CUDA SASS assembly file, and analyses the performances of the execution of a program on the GPU. The main objective of this model is to provide performance measures, with particular focus on the number of clock cycles required by a specific kernel to run on a specific GPU. The main metrics that can be derived from our model are the total number of clock cycles required to execute the application, the number of active registers at any point of program's execution and the distribution of memory accesses to detect potential bottlenecks. We compare our results with the information provided by the NVIDIA Nsight profiler [6], a tool that allows to monitor the execution of CUDA programs over physical GPUs and to provide insights on the usage of their components.

979-8-3503-1958-3/23/$31.00 ©2023 IEEE
DOI 10.1109/QRS60937.2023.00079

The rest of the paper is organized as follows: in Section II we provide some background information about GPUs' architectures and discuss related works. In Section III we illustrate our modeling approach, describing what we want to represent in our model. In Section IV we give a detailed description of our stochastic model. In Section V we apply our approach to a concrete case study, modeling two GPU architectures and simulating the execution of a CUDA program. We compare the results obtained from the simulation with the execution of the program over the real GPUs. Finally, in Section VI we draw the conclusions and discuss possible future developments.
The model is available at [7].

2. Background

2.1. GPU Architecture

GPUs are designed to perform well in parallel tasks. This is achieved with redundancy of computational and memory resources. As can be seen from Fig. 1, a GPU is made of several Streaming Multiprocessors (SM in the figure), which are the computational units of the device. Shared among the SMs there is an L2 Cache memory and, possibly, an off-chip DRAM [9]. Internally, each SM is made of several Processors, which contain hardware resources to execute the instructions of a given program.
Fig. 2 shows the architecture of an SM and a processor. An SM is essentially just a container for Processors (P in the figure), which are the actual computational units of a GPU. In its simplest form, internally, each processor has several ALUs (CUDA cores in the NVIDIA terminology) and a Register File. An L1 Cache is shared among all the Processors inside an SM. For execution, threads are split in groups of 32, called warps, which are the atomic execution unit for GPUs, independent from each other. Warps are created and dispatched to the various SMs, trying to optimize and equally distribute the workload. Each SM assigns warps to their internal CUDA cores [8].
To facilitate the development of parallel applications running on GPGPUs, Nvidia proposed the CUDA programming model [1], an "extension" of C++. The code to be run on the GPU must be defined in functions denoted by CUDA's specific keywords. The set of instructions that will run on the GPU is called *kernel*. A kernel is executed as a grid of blocks of threads. The number of blocks and the number of threads

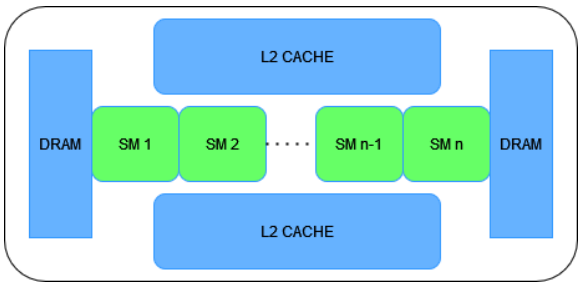

Figure 1: GPU Architecture

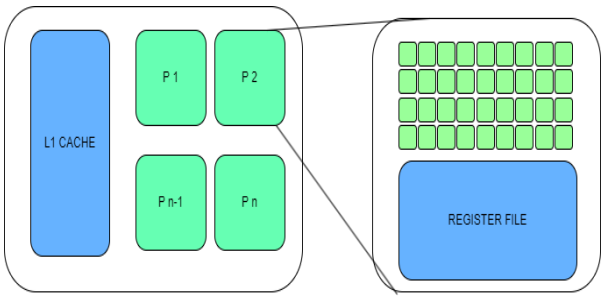

Figure 2: Streaming Multiprocessor and Processor

per block are decided by the programmer and can be optimized as needed. Inside the GPU, each kernel's block is assigned to one Streaming Multiprocessor, which will then sub-divide the block and assign these portions to its Processors, where threads are split into warps, representing the smallest unit of execution observable by the user.

2.2. Related Works

Performance analysis of applications running on GPUs gained attention due to the increasing use of GPUs to execute complex programs, such as Neural Networks [10]. There are many different approaches towards performance evaluation of hardware components, among which we recall benchmarks, models, and simulators. Each of these methods uses different levels of abstraction and has its strengths and weaknesses [11]. Benchmarking is an effective and trustworthy method to estimate performance. A benchmark is made of a set of reference programs specifically developed to stimulate specific areas of a target device. It requires an implementation of the target device either in software or hardware, hence it is generally applicable only in the later stages of development. One of the most appealing features of benchmarks is that they can also be executed on a digital representation of a device, but the trustworthiness of the results will then also strictly depend on how the digital representation was implemented [12].
Performance evaluation by simulation is one of the most used approaches, as they allow to estimate the performance of a hardware component or even a fully developed system based on an abstract representation of its internal components. Simulators provide powerful tools to represent and simulate hardware components in a way that tries to match how the real hardware device will operate, e.g., using hardware description languages. These simulators provide good measurement accuracy, but the codebase is usually complex and the execution time of such simulations is slow, even for simple programs [13].
GPGPU-Sim [14] is one of the reference simulators for GPUs. It is implemented in C++ and is organized as a combination of two simulators: a functional simulator and a timing simulator. The former is necessary to "execute" the instruction of the program, while the latter computes the time (in clock cycles) required by the GPU to execute an instruction. It is configurable via a configuration file to simulate the

architecture of many NVIDIA GPUs. Another simulator that was proposed in literature is FlexGrip [15]. This simulator is implemented using VHDL, a hardware description language, and currently implements the NVIDIA G80 GPU architecture. Contrary to GPGPU-Sim, FlexGrip can't be configured to simulate different hardware architectures, as it requires manual modification of the software codebase to represent the novel architecture.

Model-based evaluation [16] has been widely used to carry out performability analyses of complex systems in different domains. Petri Nets [17] and their extensions are one of the most common formalisms used to verify that certain properties are satisfied in the analysed system. SANs are one of the most used Petri Nets' extensions.

There have been proposals to use Petri Nets to evaluate the performances of GPUs and other parallel architectures on given workloads. In [18], Colored Petri Nets are used to model and simulate a concurrent application executed on a heterogeneous multicore platform. The model is also used to evaluate the implemented task scheduling policies.

The authors of [19] make use of Stochastic Petri Nets to estimate the job execution time of an implementation on GPUs of the map-reduce algorithm. However, their focus is on modeling the whole map-reduce process, and they do not detail the execution of the program's instructions on the GPU, nor they do consider the GPU's structure and the interactions between its components.

The authors of [20] use Colored Petri Nets to evaluate the performances of GPGPUs, using colors to represent different instruction types. They model the behavior of two different kernels on two separate GPU's architectures. However, the model is not generalizable to other GPUs, nor to other applications, as the behavior of the program is embedded in the net so, in order to evaluate different programs, the model should be manually modified.

In [21] bounded Petri nets are used to count the computational and memory operations performed on a GPU, based on the NVIDIA's CUDA programming model where memory is logically partitioned in shared, local, and global. To test the model, the authors use a pseudocode of an algorithm as input and compare the number of steps of the model's execution against the real kernel execution time on the GPU. We argue that the code used as input should be derived from the assembly source file for the specific GPU for a more realistic evaluation.

Summing up, the existing works mainly focus on specific programs running on specific GPUs on which the model is tailored. Moreover, the logical (CUDA) and physical (GPU) levels are mixed. We argue that memory latencies should be associated to the physical memory (L1, L2, DRAM) that will be accessed rather than to CUDA's logic memories.

Another problematic aspect is that the program is often represented as a list of pseudo-instructions defined by the authors, without considering the real Instruction Set Architecture (ISA).

To the best of our knowledge, our proposal is the first SAN-based model for GPUs. The use of SANs grants a high level of parameterization and configurability, which is discussed in the rest of the paper.

3. Model's Objective and Key Elements

In this section we describe the key elements of our model. Given that we are interested in counting the number of clock cycles required by a given kernel to execute on a GPU, we can discretize our timing model. This approach is consistent with the works discussed in Section 2.2. There is not an official reference concerning the number of clock cycles needed by each operation, but they can be derived from different sources such as benchmarks, knowledge of the GPU's architecture, or by presentations and comments by NVIDIA developers [22], [23], [24].

3.1. GPU Representation

From a programming perspective, the GPU is used as an accelerator for specific computations. A programmer must specify the number of blocks (also called Cooperative Thread Arrays, CTAs) and the number of threads per block before launching the kernel on the GPU. At hardware level, each block is assigned to a single SM, with a scheduling policy that tries to occupy the totality of available GPU's resources. Internally, the threads in each block are split in groups of 32 threads, called warps, which is the smallest execution unit for a GPU. It is important to keep in mind that, although all the threads belonging to the same kernel will execute the same program, i.e., the same instructions, they may diverge as result of branching. It is important then to trace which instructions are effectively executed by the program.

With these considerations we identified the following entities to be modelled (for additional details we defer to [1]):

- WARP – It is the atomic unit of execution in GPUs, as the GPU scheduler does not schedule single threads. A warp contains the list of instructions of the kernel parsed from the SASS file. Each warp is assigned to one Processor and warps assigned to different Processors will execute in parallel, while warps assigned to the same Processor must execute sequentially. The number of warps that must be executed sequentially is one of the most impacting parameters that may slow down the kernel execution time.
- REGISTER FILE – The register file is the memory in which threads store data during the computation. Each instruction executed by a warp requires several registers to be performed. This number can be mapped based on knowledge of the ISA or by using the NVIDIA profiler. Registers are assigned per-thread at the beginning of the kernel execution and allow for fast data-retrieval. Given that the number of registers is limited, knowing how many registers are used by an application may be used to guide SW/HW optimization.
- STREAMING MULTIPROCESSOR – The Streaming Multiprocessor is internally composed of many

Processors. Warps are distributed among SMs to optimize the workload.

- PROCESSOR – In GPUs, each Processor contains the hardware resources to perform the given computation, such as ALUs and Load/Store Units. The Warp Scheduler inside each Processor selects instructions from active warps and schedules them one at a time, ensuring that each warp is executed sequentially.
- CUDA MEMORIES (Local, Global, Shared) – CUDA adopts a logic memory model to ease programmers from optimizing their code for optimal memory usage. Unfortunately, there is not direct correspondence between CUDA's logic memories and a GPU physical memory, e.g., data that resides in "Global Memory" may actually be retrieved from the L1 Cache of a Processor instead of the DRAM, which is commonly addressed as "Global Memory" in computer science.
- PHYSICAL MEMORIES (L1/L2 Cache, DRAM) – Different memories in the memory hierarchy are built for different purposes and with different technologies. This means that each of these memories will have different time accesses, in terms of clock cycles. We model these components to tie read and write operations to physical memories rather than CUDA's logic memories.

3.2. Program Representation

To represent a program in our model, we take advantage of the SASS source file that can be obtained during the compilation of a program. We decided to use SASS code instead of PTX, because SASS code is tied to the specific GPU architecture. PTX, instead, is a high-level language that is architecture-independent, and it is translated by the compiler to SASS code. SASS code can be easily obtained using compiler's utilities [26]. Thanks to the use of the NVIDIA Nsight profiler, it is also possible to know how many times a specific instruction was executed. This is extremely useful to provide a realistic and faithful representation of the program's execution, to handle situations such as branch divergency, and it can also be obtained from static code analysis as shown in [20], [21]. Each SASS instruction is associated to an identifier representing the kind of operation, e.g., ALU operation or Load/Store. This identifier is used to enable the right sub-model that simulates the execution of that specific instruction. We developed a tool that parses SASS files and produces the initialization code for the model. When we parse the file to represent the program's instructions inside our model, we also track the activation and deactivation of registers in the Register File. This allows to understand the total number of live registers and hence the register file's occupancy level at each instruction.

The parser works as follows: every line of the SASS file is a CUDA instruction in the form <INST, REGS> where INST is the instruction's opcode, e.g., ADD, and REGS is the number of registers used by that instruction. To produce a code representation readable by the model, we assign an integer identifier to the following ISA instructions, which we report in Table 1.

Table 1: List of SASS instruction's ID in the model

Instruction	ID in our model
Load Local	0
Store Local	1
Load Global	2
Store Global	3
Integer Arithmetic	4
Float Arithmetic	5
Synchronization Barrier	6

Operations from ID 0 to ID 3 encode memory operations in CUDA logic memories. Operations with ID 4 and 5 represent the execution of arithmetic/logic instructions on the processors' ALUs. Given that we want to represent the exact sequence of instructions as they would be executed on a real GPU, we order them by associating an index to each, so, the first instruction *executed* by the program will be that at index 0, the second instruction at index 1 and so on. The number of registers activated or deactivated when executing an instruction depends on the kind of instruction and the index of such instruction, i.e., it depends on the "past" and "future" of the program's execution. E.g., an instruction may reuse registers that were activated by a previous instruction; hence the number of active registers would remain the same. We leverage this information to associate to each instruction's index the number of registers activated or deactivated, thus allowing to count the number of registers in use at any point of the program's execution.

3.3. Model Parameters

The model has some parameters to allow for configurability. These parameters can be tuned based on the GPU's architecture and the program's properties. We defined a set of global parameters for hardware configuration, such as the number of Processors and memory latencies.

Some of the most important parameters of the model are:

- Number of Streaming Multiprocessor (*nsm*) used.
- Number of Processors present in each SM in the GPU's architecture under study (*nprocessors*).
- Number of Warp per Processor (*nwarps*). This is one of the most important parameters, as it is the most influent on the final execution time. In fact, warps in different processors will execute in parallel, while warps executed on the same processor must be executed sequentially/interleaved.
- L1/L2/DRAM access probabilities. These probabilities guide the access to physical memories, given a logic (local, global) memory access.

4. Model Implementation Using SANs

The developed model was implemented in Möbius [25], a free tool that allows to develop stochastic models using different formalisms, including SANs. Among their benefits, SANs allow to instantiate "special" places, called Extended Places, which can be initialized with typed tokens, where types can be

numbers (short, integer, float) or even data structures such as arrays. We exploit this feature to initialize the WARP and REGISTER_FILE sub-models, allowing us to order the instructions in a sequence that reflects the execution of the kernel on the GPU.
In this section we give a detailed description of the model. However, for an in-depth view, we defer to the repository available at [7].

4.1. Atomic Models

In our model we represent the following entities:

- WARP;
- EXECUTION UNIT;
- MEMORY;
- L1-L2_CACHE / DRAM;
- REGISTER_FILE.

Warp

The WARP sub-model (Fig. 3) represents a group of 32 threads executing on a single Processor inside an SM. The model contains the following places:

- WARP: extended place that is initialized with the list of instructions' IDs parsed by the SASS file (see Section 3.2), represented by an array.
- INSTRUCTION_READY: it represents the *ready* state of a warp, i.e., the state in which the warp can send instructions to the execution unit.
- INST_COUNTER: it contains the index of the next instruction to be executed.
- REGISTERS_FILL: its marking is set to 1 when the warp must wait for the number of registers in use to be updated before executing an instruction.
- SCHEDULER: when an instruction is dispatched, it contains the next instruction type to be executed.

The INST_SELECT input gate is enabled if there are instructions to be executed, i.e., the instructions' counter has not reached the total number of program's instructions, and if no other instruction is being executed on the processor to which the warp is assigned. The input function of INST_SELECT increases the instructions' counter and decreases the INSTRUCTION_READY tokens. The INST_ISSUE output gate puts the ID of the next instruction to be executed into the SCHEDULER extended place (selected from the list of instructions' IDs contained in WARP) and set the marking of REGISTERS_FILL to 1. This is done to activate the needed registers before executing the instruction.

Execution Unit

The EXECUTION_UNIT sub-model (Fig. 4) represents the Processor inside an SM. As of now, our model allows to represent Integer and Floating Point ALU instructions, memory instructions and barrier synchronization. The model contains the following places:

- SCHEDULER, INSTRUCTION_READY and REGISTER_FILL: they have the same semantics as the analogous places of the WARP model.
- FLOAT_ALU and INT_ALU: they represent a floating point or integer instruction, respectively, executed by ALUs.
- READ and WRITE: they determine if a memory instruction is executed. We make use of extended places as we codify here the type of logical memory (CUDA memory) on which the operation will be performed.
- BARRIER: it represents a barrier synchronization operation.

The RETRIEVE_INSTRUCTION input gate is enabled when a new instruction is available in the SCHEDULER place and the REGISTER_FILL marking is 0, meaning that the registers have been activated. The gate's function sets the INSTRUCTION_READY marking to 0. Depending on the instruction's type, the DISPATCH output gate dispatches the instruction to the corresponding place according to the SCHEDULER marking (e.g., an integer operation is represented by a token in INT_ALU place). Once the operation is completed, the processor will be ready to execute another instruction (i.e., the marking of the INSTRUCTION_READY place is set to 1).

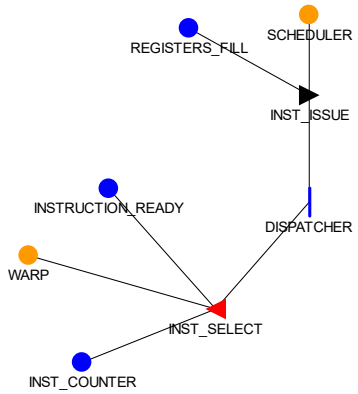

Figure 3: WARP atomic model

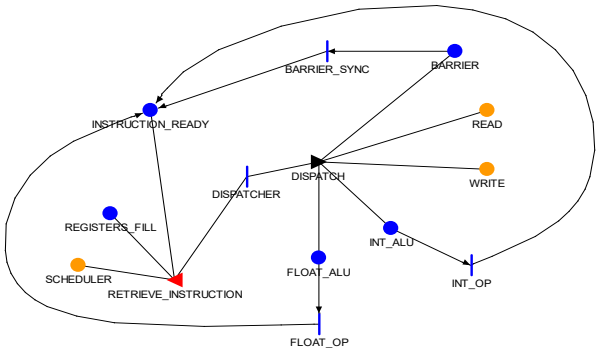

Figure 4: EXECUTION_UNIT atomic model

Memory

The MEMORY sub-model (Fig. 6) serves as the bridge between the CUDA programming model and the actual hardware architecture of a GPU. In fact, CUDA programs distinguish memory accesses between local/shared accesses and global accesses. However, this is not directly mapped to the physical memories of the GPU. In fact, a variable marked as local may be stored in the DRAM (off-chip memory) at any point of the program execution. Our model maps SASS memory instructions (e.g., Load Global) to physical memory accesses. The model contains the following places:

- READ and WRITE: they have the same meaning of the same places of the EXECUTION_UNIT model.
- READ_LOCAL and READ_GLOBAL: they represent the read operation executed on CUDA logic memory.
- READ_L1, READ_L2 and READ_DRAM: they represent the read operation executed on the actual physical memory of the device.
- WRITE_LOCAL and WRITE_GLOBAL: they represent the write operation executed on CUDA logic memory.
- WRITE _L1, WRITE _L2 and WRITE _DRAM: they represent the write operation executed on the actual physical memory of the device.

The READ_INPUT input gate is enabled when READ place contains a significant value, i.e., an identifier of the CUDA memory type. The READ_SELECT output gate determines on which CUDA logical memory the read operation is performed, according to the content of the place READ. LOCAL_READ_FROM and GLOBAL_READ_FROM activities have probabilities cases to map operations performed on CUDA logic memories to the physical memories of the GPU. Write operations are modelled in an analogous way.

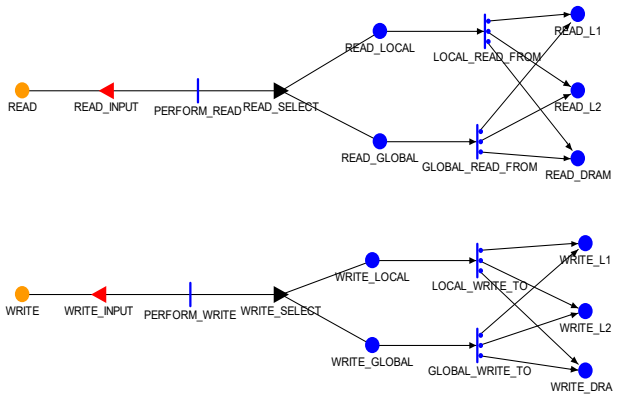

Figure. 5: MEMORY atomic model

Physical memories

The physical memory sub-models, L1_CACHE, L2_CACHE and DRAM, allow to represent hardware memories of the GPU, to count the accesses performed to each physical memory. As an example, in Fig. 5 we show the model of the DRAM memory (L1 and L2 cache models are similar). Each memory operation (represented by transitions PERFORM_READ and PERFORM_WRITE) is associated to a fixed value, representing the number of clock cycles needed to complete a transfer from the specific physical memory to the warp requesting those data. We use this information in the reward models. The completion of one of these operations enables the next instruction to be executed.

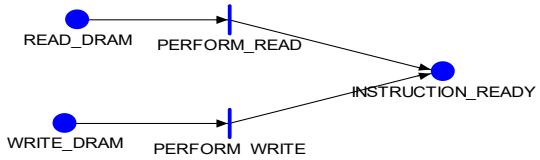

Figure 6: DRAM atomic model

Register file

Each instruction in a program needs to use a certain number of registers, allocated in the register file. By inspecting the SASS source file, it is possible to understand whether a register is activated by a given instruction, e.g., to store the result of a computation, will be used again or not after the instruction that required it. The REGISTER_FILE sub-model (Fig. 7) contains the following places:

- REGISTER_FILL: it has the same semantics as the analogous places in WARP and EXECUTION_UNIT atomic models.
- LIVE_REGISTERS: this extended place is initialized with the list of live registers activated or deactivated by each instruction obtained by the parser (see Section 3.2), represented with an array.
- INDEXES: it is used as an index for the array contained in the LIVE_REGISTERS place.
- ACTIVE_REGISTERS: it contains the current number of active registers. The number of registers in use by the kernel is updated before the execution of each instruction.

The READ_REGISTERS input gate is enabled if the marking of REGISTERS_FILL is equal to 1 and if the marking of INDEXES is less than the dimension of the array in LIVE_REGISTERS. When the input function of the gate is executed, the marking of REGISTERS_FILLS is set to 0 and the marking of INDEXES is increased. The execution of the ACTIVATE_REGISTERS output gate updates the current number of active registers according to the value found in LIVE_REGISTERS.

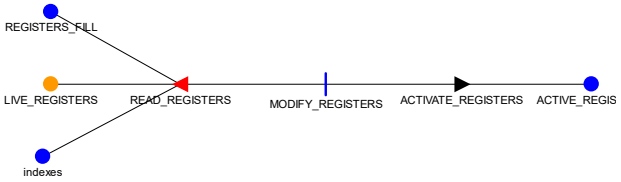

Figure 7: REGISTER_FILE atomic model

4.2. Composed Model

Möbius allows to design many atomic models, like those described in the previous section, and to combine them in a composed model. The model is built using JOIN and REP constructs, to match the architectural constraints imposed by the architecture of the GPU. The former is used to link together the different sub-models; the latter is needed to replicate the sub-models to match the target GPU architecture. The complete composed model is depicted in Fig. 8.

The final GPU model is built by linking together the sub-models (e.g., the DRAM and L2 Cache must be shared among all the sub-models, as they are shared among all the GPU's resources). In this model, places with the same name are shared between sub-models.

The initial marking of the model is declared in the WARP and REGISTER_FILE sub-models, with the first being initialized with the list of the program's instructions, and the latter with the list of live registers used or dropped at each instruction. These two atomic models are joined together, sharing the REGISTER_FILL place and replicated a number of times equal to the *nwarps* parameter.

These sub-models are then joined at Processor level with EXEC_UNIT and MEMORY sub-models, sharing READ, WRITE, INSTRUCTION_READY and SCHEDULER places. These sub-models are replicated according to the *nprocessors* parameter and joined with the L1_CACHE sub-model. Finally, the models are replicated according to the number of Streaming Multiprocessors (*nsm*) and joined with the L2_CACHE and DRAM atomic models.

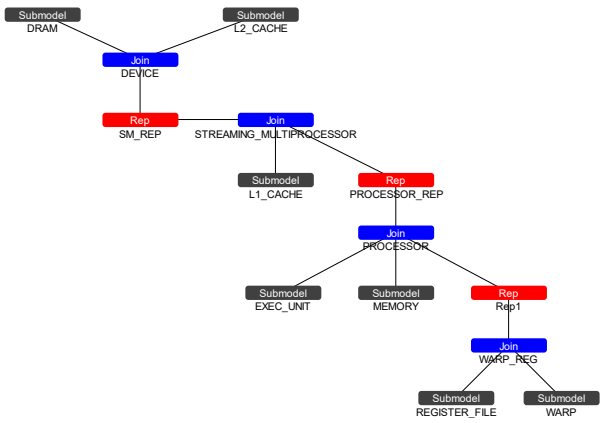

Figure 8: Composed model

4.3. Model's Setting

Setting of the parameters - To have a realistic representation of the CUDA programs running over GPUs, we tune some parameters of our model using the information obtained from the NVIDIA Nsight profiler. We derive the proportions of accesses to the different physical memories thanks to the proportion of cache hits and misses provided by the profiler. We assign these proportions ratios to the cases' probabilities of the activities. Looking at the MEMORY atomic model in Fig. 6, e.g., LOCAL_READ_FROM has three parameterized cases' probabilities, *local_read_l1*, *local_read_l2* and *local_read_dram*. Assign a probability of 0 to one of these cases, for both local and global memory operations, simulates the absence of the corresponding memory, e.g., to examine the impact of having or not having a local cache. Other parameters, such as the number of streaming multiprocessors (*nsm*), processors (*nprocessors*) and warps (*nwarps*) used by a program depend on the GPU's architecture and on the program's code.

Setting of Reward Variables - As explained in III, we are interested in a discrete measurement of the number of clock cycles necessary to execute a program. To accomplish this, we define some impulse reward variables assigned to specific activities of the atomic models, which have an impact on the measure of interest. In particular, the activities which can be used to count the clock cycles are reported in Table 2. The table reports the name of the GPU's operation, the corresponding activity's name, the sub-model to which it belongs, and the reward value representing the clock cycles needed to perform such operation. We recall that the exact cost in clock cycles is not known for every operation, so we had to collect values from different sources [22], [23], [24].

Model Initialization – The model is initialized as follows: the initial marking of the model is given by the marking of the WARP extended place in the WARP atomic model, and by the marking of the LIVE_REGISTERS extended place, in the REGISTER_FILE atomic model. The marking of these places consists in an array containing: i) the instructions IDs' list for the WARP atomic model, and ii) the number of registers activated or deactivated by each instruction for the REGISTER_FILE atomic model, as described in Section 3.2. E.g., a program that executes an arithmetic instruction and a store to global memory, would be represented in the WARP atomic model with the array: [4, 3], and in the REGISTER_FILE atomic model, with the array [+3, -3], if an integer operation requires 3 registers to be executed, i.e., operands registers and result register, that will be released after the store to global memory.

Example of model composition – To represent a real GPU we can compose the proposed models in the following way: the most important parameter that guides the models' simulation is *nwarps*, representing the number of warps that must be run sequentially on the GPU. This is obviously one of the most impacting parameters as warps executed in parallel will spend the same amount of time running in parallel. Hence, if one is interested only in estimating the number of clock cycles required to run an application, it is sufficient to properly set this parameter. This approach is consistent with other works on GPU stochastic modeling, discussed in Section 2.2 [20], [21]. For a resource-oriented analysis, it is also possible to estimate metrics for each hardware resource represented in our model. The number of SMs present in the architecture can be modified with the parameter *nsm*. Each SM has a fixed number of Processors which is tuned by the parameter *nprocessors*. For example, to model the FERMI GTX 480 GPU, with 15 SMs and 1 Processor each, is

sufficient to set the parameter *nsm* to 15 and *nprocessors* to 1. Instead, the parameter *nwarps* is computed depending on the number of hardware resources and on the number of threads generated by the kernel.

Table 2. GPU's operations representation in the model along with the reward variable associated to count the number of clock cycles needed by the operation.

GPU OP	ACTIVITY NAME	SUBMODEL	REWARD
INT Operation	INT_OP	EXEC_UNIT	2
FLOAT Operation	FLOAT_OP	EXEC_UNIT	2
L1 Access	PERFORM_READ PERFORM_WRITE	L1_CACHE	12
L2 Access	PERFORM_READ PERFORM_WRITE	L2_CACHE	300
DRAM Access	PERFORM_READ PERFORM_WRITE	DRAM	600
Barrier	BARRIER_SYNC	EXEC_UNIT	3000

5. Case Study

In this section we describe the validation campaign performed to validate the model, along with the algorithm chosen and the reference architectures. We evaluate our model by simulation, until it reaches a terminating state. We ran each simulation with, respectively, 100 and 1000 minimum and maximum number of batches (simulation runs), with a 0.1 relative confidence interval and a 0.95 confidence level.

5.1. Target Algorithm and GPUs

To validate our model, we chose the most frequently used algorithm on GPUs, that is: General Matrix Multiplication (GEMM) solver. The code is shown in Fig. 9. We configured the model to match the architectures of the NVIDIA RTX 2080 GPU and the NVIDIA QUADRO 6000 RTX. The former has 46 SMs with 4 Processors each, with a maximum number of warps that can run in parallel equal to 46 x 4 = 184; the latter has instead 72 SMs with 4 Processors each, hence capable of running 72 x 4 = 288 warps in parallel. The launch parameters of the Kernel, i.e., the number of blocks and the number of threads per blocks, are needed to understand the level of parallelism of the application, which allows one to compute how many warps will be scheduled per processor.
The model can be initialized depending on the kind of analysis that one intends to perform as discussed in Section 4.3. We show here an example based on the GEMM 256 case. First of all, the number of SMs and Processors can be configured according to the considered architecture, e.g., for the NVIDIA RTX 2080, *nsm* equal to 46 and *nprocessors* equal to 4. GEMM 256 creates 2048 threads, that the GPU distributes among its SMs and Processors. Given that the RTX 2080 has 184 parallel processors, the number of warps that must be run sequentially (*nwarps*) is 12.
The GEMM algorithm used in this work was implemented from scratch. This was done mostly for easiness of use and of configuration of the size of the problem (matrix). Each matrix is initialized with random numbers.

```
__global__ void gemm(const int* a, const int* b,
long* c) {
    int row = blockIdx.x * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    for (int = 0; i < N; i++) {
        c[row * N + col] += a[row * N + i] *
                                b[i * N + col];

        }
}
```

Figure 9: General Matrix Multiplication (GEMM) code used. This version is developed to compute a*b=c, where a and b are matrices of fixed size NxN.

5.2. Results of the Simulations

The kernel was first profiled with matrices of Increasing size: 32x32, 64x64, 128x128, 256x256, 512x512. To validate our results, the number of clock cycles estimated by the model is then compared to the one measured on the real GPU using the NVIDIA Nsight Profiler.
Fig. 10 shows the simulated and real clock cycles spent by the kernel to perform GEMM on these matrices for RTX 2080 GPU. The figure report on the x-axis the number of clock cycles, while on the y-axis it is reported the size of the matrix. It is possible to notice that the results obtained by the model simulation are close to those profiled on the real GPU.
To better visualize the accuracy of our model, we plot in Fig. 11 the absolute value of the relative error of our model against the real value obtained by the NVIDIA Nsight Profiler.
The average error measurement is 6%, comparable to other works in literature, e.g., in [20], the average error measurement is 6% on the NVIDIA FERMI GTX 480 and 8% on the NVIDIA KEPLER K20m. If we measure the error from

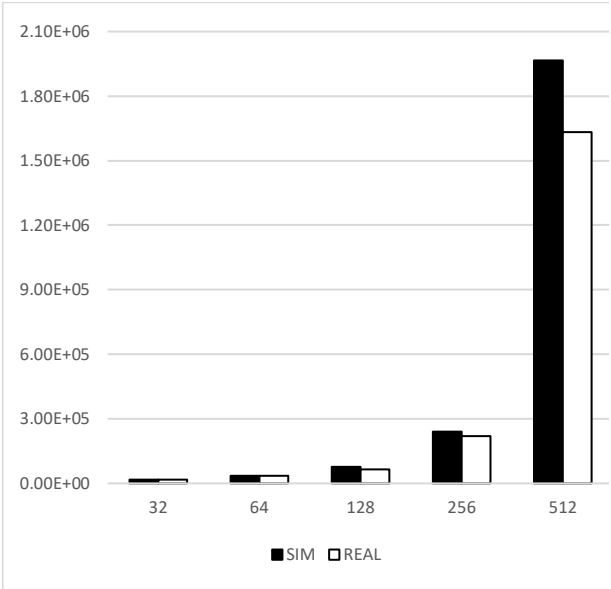

Figure 10: Clock cycles estimated by the model (black) against the clock cycle profile by the NVIDIA Nsight Profiler (white) for the NVIDIA RTX 2080. The y-axis reports the number of clock cycles, while the x-axis reports the size of the matrix.

32x32 to 256x256, obtain an average error of 0.0275. In the 512x512 case we have a huge increase in measurement error to 20%. We argue that the improvements made by manufacturers to boost GPUs performance also undermine the predictability of their performance, leaving room for measurement errors. This is pointed out also in [4], where some models are shown to fail at predicting performance of more recent GPUs.

Fig. 12 reports the simulation of the GEMM algorithm on the NVIDIA Quadro 6000 RTX. We plot the absolute relative error in Fig. 13. First, we can see that the model predicts with higher accuracy the case 512x512, with an error of 2% against the error of 20% previously observed for the RTX 2080. The model is much more accurate on this GPU, with a maximum error of 10% and an average error of 4%. Compared to the error measured with the NVIDIA RTX 2080, it has a slightly lower accuracy in the first four cases, but it doesn't show the anomaly observed previously in the 512x512 case, where the measured error was 20%. The measurement error for the 512x512 case on this GPU is, instead, 2%.

The model can be used to perform different kinds of analysis. We report here an analysis of the load distribution on the hardware units used in our GEMM implementation, i.e., ALUs, L1 and L2 Cache and DRAM. Fig. 14 shows the percentage of clock cycles spent by the program on each of these hardware units on the Quadro RTX 6000. We can see that when the dimension of the matrix is small, in the 32x32 and 64x64 cases, the bottleneck is caused by the L2 cache. This is consistent with GPU behavior; in fact, data transfers to and from the CPU must pass through the L2 cache [27], and, for small matrices, they may entirely reside in this memory. Thus, warps spend most of their time for data to be read from the L2 Cache (more than 70% of the whole execution time), process these data in few cycles (the time spent on ALUs is less than 2%), and then write these data to global memory. The benefits of having an L1 Cache can be observed becoming relevant starting from the 128x128 case. We can also observe

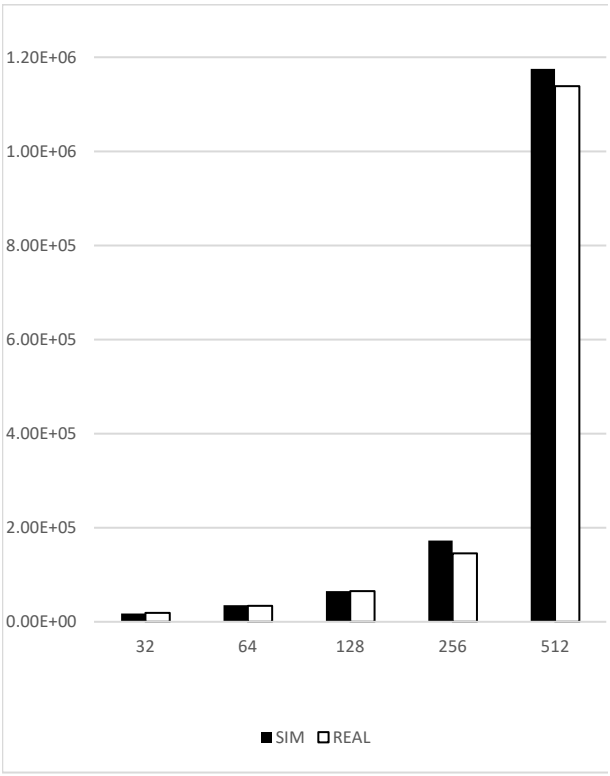

Figure 12: Clock cycles estimated by the model (black) against the clock cycle profile by the NVIDIA Nsight Profiler (white) for the NVIDIA Quadro 6000 RTX. The y-axis reports the number of clock cycles, while the x-axis reports the size of the matrix.

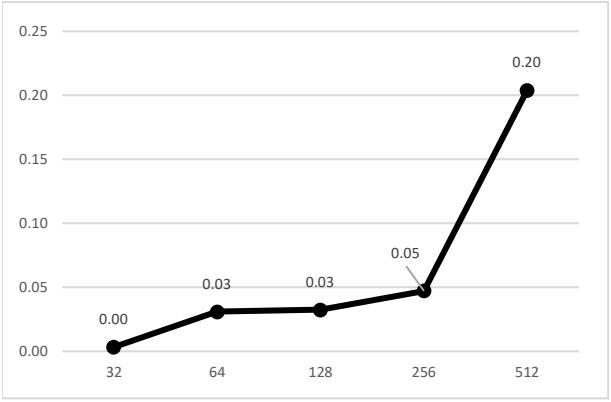

Figure 11: measurement error on the clock cycles estimated by our model w.r.t. those measured by the NVIDIA Nsight profiler on the NVIDIA RTX 2080 GPU.

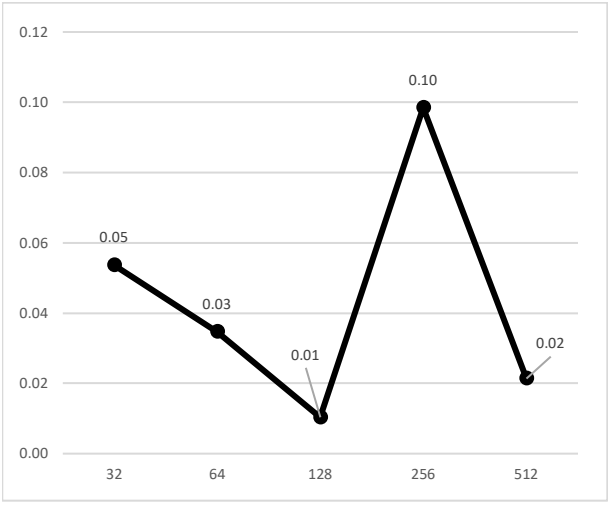

Figure 13: measurement error on the clock cycles estimated by our model w.r.t. those measured by the NVIDIA Nsight profiler on the NVIDIA Quadro 6000 RTX GPU.

that the load distribution of the 256x256 case is very similar to the 512x512 one.

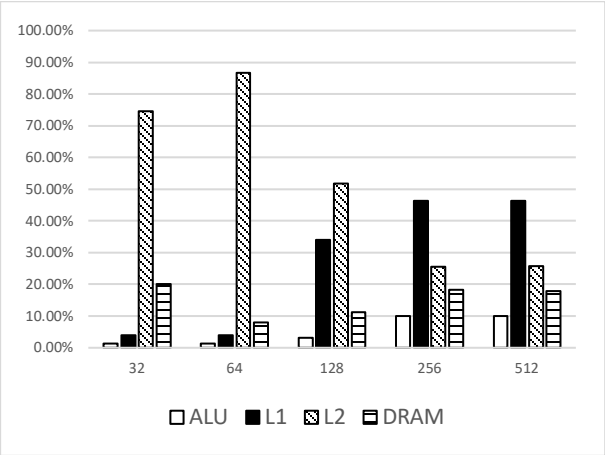

Figure 14: fraction of time spent on ALU, L1 and L2 Cache, and DRAM for the GEMM algorithm on the NVIDIA Quadro RTX 6000.

6. Final Discussion and Conclusion

In this work we developed a stochastic model using SANs to simulate the execution of CUDA programs running on NVIDIA GPUs. Other stochastic models discussed in literature rely on a custom program representation; we, instead, parse directly the CUDA SASS code file to obtain a faithful program representation as input for the model. In this way each instruction represented in our model is tied to a specific CUDA ISA instruction. We simulated a matrix multiplication algorithm on two NVIDIA GPUs: the RTX 2080 and the Quadro RTX 6000 to estimate the number of clock cycles required with matrices of increasing size. We also showed an example of how the model can be used to measure the percentage of clock cycles spent on the different functional units.

The results show good prediction accuracy in both our experiments, with the highest error in the case of the 512x512 GEMM Solver on the RTX 2080. Excluding this anomaly, the model is very accurate, with an average measurement error of 0.0275. The predictions for the Quadro RTX 6000 instead are better, with an average error of 4%. A decrease in accuracy as the workload size increases is also observable in [20], [21], to the best of our knowledge, the only works that try to model the internal structure of a GPU. However, the work in [21] does not focus on estimating the number of clock cycles required by a given application, hence we compare our results with those in [20].

The authors of [20] estimate the number of clock cycles on an implementation of the GEMM algorithm. They validated their model by simulating the NVIDIA FERMI GTX 480 and NVIDIA KEPLER K20m GPUs. An increase in measurement error as the workload gets larger is observed in their work too, however if we average the errors for all the experiments, we can see that their model has an error of 7% while our model has an error of 5%.

We argue that, as stated in [4], the lack of knowledge of internal architectural details and possible hardware optimizations, which are often not disclosed by vendors, limit the accuracy of the results.

As pointed out in Section 2.2, it may be possible to obtain better results using tools such as, e.g., digital simulators, or benchmarking. However, these techniques are adopted to develop and/or validate prototypes of a final product, and comparing many different solutions in such a way is expensive and time-consuming. The purpose of model-based analysis is, instead, different. Its goal is to help system designers in comparing many technical solutions quickly and with minimum effort, usually at the early stages of system's development. Hence, the strength of the proposed model is its configurability, with the possibility of simulating many hardware configurations, to allow to quickly compare many different architectural solutions.

In fields such as that of safety-critical systems, designers are required not only to evaluate the performance of different technical solutions working in nominal conditions, they also must evaluate situations in which the system is working in degraded conditions. We argue that with minor modifications, the model proposed in this work could be an effective tool to simulate such situations.

As future work, our major interest is to extend the model to simulate: i) situations in which a hardware unit is failed *prior* to running a program and, ii) situations in which a hardware unit fails *during* the execution of a program. Other improvements to the model are to extend our SASS parser and to include novel functional units such as Tensor Cores.

References

[1] CUDA, https://developer.nvidia.com/cuda-toolkit - Last visited: 23-06-2023

[2] Bridges, Robert A., Neena Imam, and Tiffany M. Mintz. "Understanding GPU power: A survey of profiling, modeling, and simulation methods." ACM Computing Surveys (CSUR) 49.3 (2016): 1-27.

[3] Khairy, Mahmoud, et al. "Accel-Sim: An extensible simulation framework for validated GPU modeling." 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2020.

[4] Lopez-Novoa, Unai, Alexander Mendiburu, and Jose Miguel-Alonso. "A survey of performance modeling and simulation techniques for accelerator-based computing." IEEE Transactions on Parallel and Distributed Systems 26.1 (2014): 272-281.

[5] W. Sanders and J. Meyer, "Stochastic activity networks: formal definition and concepts," in Lectures on formal methods and performance analysis, ser. LNCS. Springer, 2002, vol. 2090, pp. 315–343.

[6] NVIDIA Nsight System - https://developer.nvidia.com/nsight-systems, visited: 27/04/2023

[7] https://github.com/FrancescoTerrosi/gpu-model/tree/validation

[8] Glaskowsky, Peter N. "NVIDIA's Fermi: the first complete GPU computing architecture." White paper 18 (2009).

[9] The History of the GPU - New Developments, Jon Peddie, DOI https://doi.org/10.1007/978-3-031-14047-1, Springer Cham, ISBN 978-3-031-14046-4, 03 January 2023

[10] Shi, Shaohuai & Chu, Xiaowen. (2017). Performance Modeling and Evaluation of Distributed Deep Learning Frameworks on GPUs.

[11] Lucas Jr, Henry. "Performance evaluation and monitoring." ACM Computing Surveys (CSUR) 3.3 (1971): 79-91.

[12] Dattakumar, R., and R. Jagadeesh. "A review of literature on benchmarking." Benchmarking: An International Journal 10.3 (2003): 176-209.

[13] Munteanu, Daniela, and J-L. Autran. "Modeling and simulation of single-event effects in digital devices and ICs." IEEE Transactions on Nuclear science 55.4 (2008): 1854-1878.

[14] Bakhoda, Ali, et al. "Analyzing CUDA workloads using a detailed GPU simulator." 2009 IEEE international symposium on performance analysis of systems and software. IEEE, 2009.

[15] Andryc, Kevin, Murtaza Merchant, and Russell Tessier. "FlexGrip: A soft GPGPU for FPGAs." 2013 International Conference on Field-Programmable Technology (FPT). IEEE, 2013.

[16] D. M. Nicol, W. H. Sanders, and K. S. Trivedi, "Model-based evaluation: From dependability to security," IEEE Trans.Depend. Sec. Comput., vol. 1, no. 1, pp. 48–65, Jan./Mar. 2004.

[17] Petri, Carl Adam (1962). Kommunikation mit Automaten (Ph. D. thesis). University of Bonn.

[18] I. D. Mironescu and L. Vinţan, "Coloured Petri Net modelling of task scheduling on a heterogeneous computational node," 2014 IEEE 10th International Conference on Intelligent Computer Communication and Processing (ICCP), Cluj-Napoca, Cluj, Romania, 2014, pp. 323-330, doi: 10.1109/ICCP.2014.6937016.

[19] Yang Hung, Sheng-Tzong Cheng, Chia-Mei Chen. 2015. Estimation of Job Execution Time in MapReduce Framework over GPU clusters. PESARO 2015, The Fifth International Conference on Performance, Safety and Robustness in Complex Systems and Applications, Barcelona, Spain, ISBN: 978-1-61208-401-5, Pages: 15 to 20

[20] Souley Madougou, Ana Lucia Varbanescu, and Cees de Laat. 2016. Using colored petri nets for GPGPU performance modeling. In Proceedings of the ACM International Conference on Computing Frontiers (CF '16). Association for Computing Machinery, New York, NY, USA, 240–249. https://doi.org/10.1145/2903150.2903167

[21] Gogolińska, A., Mikulski, Ł., Piątkowski, M. (2018). GPU Computations and Memory Access Model Based on Petri Nets. In: Koutny, M., Kristensen, L., Penczek, W. (eds) Transactions on Petri Nets and Other Models of Concurrency XIII. Lecture Notes in Computer Science(), vol 11090. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-662-58381-4_7

[22] NVIDIA Developer Forum - https://forums.developer.nvidia.com/, visited: 27/04/2023

[23] Mei, Xinxin, et al. "Benchmarking the memory hierarchy of modern GPUs." Network and Parallel Computing: 11th IFIP WG 10.3 International Conference, NPC 2014, Ilan, Taiwan, September 18-20, 2014. Proceedings 11. Springer Berlin Heidelberg, 2014.

[24] Wong, Henry, et al. "Demystifying GPU microarchitecture through microbenchmarking." 2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS). IEEE, 2010.

[25] Daly, David, et al. "Möbius: An extensible tool for performance and dependability modeling." Computer Performance Evaluation. Modelling Techniques and Tools: 11th International Conference, TOOLS 2000 Schaumburg, IL, USA, March 27–31, 2000 Proceedings 11. Springer Berlin Heidelberg, 2000.

[26] CUDA Binary Utilities, https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html - Last visited: 23-06-2023

[27] NVIDIA Developer Forum - https://forums.developer.nvidia.com/t/cudamemcpy-and-l2-cache/42817, visited: 11/07/2023