The Pipeline Performance Model: a generic executable performance model for GPUs

1st Jan G. Cornelis
imec, Kapeldreef 75, B-3001 Leuven, Belgium
Electronics and Informatics (ETRO) Dept.
Vrije Universiteit Brussel (VUB)
Pleinlaan 2, B-1050 Brussels, Belgium
jgcornel@vub.be

2nd Jan Lemeire
imec, Kapeldreef 75, B-3001 Leuven, Belgium
Electronics and Informatics (ETRO) Dept.
Industrial Sciences (INDI) Dept.
Vrije Universiteit Brussel (VUB)
Pleinlaan 2, B-1050 Brussels, Belgium
jan.lemeire@vub.be

***Abstract*—This paper presents the pipeline performance model, a generic GPU performance model, which helps understand the performance of GPU code by using a code representation that is very close to the source code. The code is represented by a graph in which the nodes correspond to the source code instructions and the edges to data dependences between them. Furthermore, each node is enhanced with two latencies that characterize the instruction's time behavior on the GPU. This graph, together with a simple characterization of the GPU and the execution configuration, is used by a simulator to mimic the execution of the code. We validate the model on the micro-benchmarks used to determine the latencies and on a matrix multiplication kernel, both on an NVIDIA Fermi and an NVIDIA Pascal GPU. Initial results show that the simulated times follow the measured times, with acceptable errors, for a wide occupancy range. We argue that to achieve better accuracies it is necessary to further refine the model to take into account the complexity of memory access and warp scheduling, especially for more recent GPUs.**

***Index Terms*—GPU, pipeline, performance, model, latencies.**

I. Introduction

In the last decade the use of GPUs to speed up parts of programs that must process massive amounts of data, has become increasingly popular thanks to development frameworks like CUDA [1] and OpenCL [2]. Often, however, programmers do not understand the performance of the code they have written, failing to know where the bottlenecks are and whether their code can be made faster still.

A lot of work has been carried out around GPU performance modeling and analysis. Graphical models like the roofline model [3] and its derivatives [4], [5], [6], take into account inefficiencies but do not model non-ideal latency hiding: they implicitly assume that the computation and memory subsystems operate independently of each other. Analytical and quantitative methods [7], [8], [9], [10] take the interaction between both subsystems into account but they mix the interaction with inefficient execution making it difficult to interpret the obtained results. Furthermore, many of these methods are hard to reproduce due to the large number of parameters needed for the model as well as the need for specialized tools. Many models use simulators, for example [11] uses a simulator to verify the accuracy of their model, while [12] proposes an abstract timing model to accelerate simulation of code run on GPUs. Although their approach seems similar to ours, they operate on a lower level focusing on NVIDIA GPUs, through the use of PTX code and GPGPUSim [13].

The great number of existing models is a symptom of the complexity of GPU performance. Moreover, current models are often closely related to a specific GPU vendor or architecture and make it difficult to map performance loss to source code. The pipeline performance model, introduced in this paper, aims to be generic enough to be applicable to GPUs of different vendors and different generations. Furthermore, the pipeline performance model remains very close to the source code in order to make it easy to find bottlenecks in the implementation caused by patterns of inefficiency like those described in [14] and to quickly evaluate the effect of optimizations without applying them. The model is based on the fact that performance of code run on a GPU is for the greatest part determined by the latencies of the instructions that need to be executed and the amount of parallelism that is available to hide these latencies. The importance of latencies for performance has been stressed before in works like [15] and [16].

To attain its goals, the pipeline performance model represents the code by an instruction dependence graph that describes the dependences between instructions, while the hardware is taken into account by assigning an issue latency and a completion latency to every instruction of the graph. These latencies describe the time behavior of the instruction on the GPU. The graph, together with a simple characterization of the GPU and the execution configuration, is used by a simulator to mimic the time behavior of the execution of the code. Because the pipeline performance model separately models the source code and the GPU it is possible to quickly perform new simulations: the instruction latencies are most of the time independent of the code, while the instruction dependence graph is independent of the GPU on which the code is executed.

The remainder of this paper is organized as follows: in Section II the necessary background to understand the performance model is given. Section III presents the model. A number of experiments to validate the model and their results are presented in Section IV. Finally, Section V concludes the paper and looks forward to future work.

II. Background

We briefly explain the architecture of a typical GPU and look at how it relates to code execution. Most of the terms used here and in the description of the model are borrowed from the OpenCL definition of a generic accelerator device. For more details we refer to [17], [18] and [19].

A GPU consists of a number of compute units made up of dedicated processing elements, a great amount of registers and an L1 cache. The memory access subsystem connects the GPU to an external RAM memory, often using an L2 cache. Threads are executed in SIMT fashion: the same instruction unit is shared by a fixed number of threads. On NVIDIA GPUs groups of 32 threads execute in lockstep and are called warps; on AMD GPUs groups of 64 threads execute in lockstep and are called wave fronts. We use the term warp for both. For optimal performance SIMT execution must be taken into account given that it influences memory access and can cause serialization of code due to branching.

The code that must be executed by one thread is specified in a kernel function, both in CUDA and OpenCL. A kernel is executed by a number of threads organized in work groups as determined in the host program. Threads of the same work group can collaborate by performing barrier synchronization and using part of the L1 cache as scratch pad memory. Work groups are assigned to compute units where they remain until their execution completes. Because the number of work groups that can execute at the same time on a compute unit is limited, most work groups must wait for others to complete before they can start executing. The size and number of work groups that need to be processed, together with the number of work groups that can be processed concurrently is referred to as the execution configuration of the kernel.

The number of work groups that can execute concurrently depends on their resource needs. The occupancy of a kernel is defined as the number of threads that can be executed at the same time relative to the maximum number. The conventional advice is to increase occupancy to obtain better performance, but there are many additional factors like the amount of ILP and MLP in the kernel and the latencies of the instructions.

III. The Model

A. Philosophy

Most modern processors and memory subsystems operate in a pipelined fashion. A simple pipeline is determined by two latencies: the issue latency and the completion latency. The former is the minimal time between the issue of an instruction and the issue of another independent instruction. The latter is the time between an instruction issue and the moment its result becomes available. We will refer to these latencies as λ and Λ respectively. For a simple pipeline λ is one cycle and Λ in cycles is equal to the number of pipeline stages.

The time needed to process a number of instructions depends on the presence of sufficient independent instructions. For example, the time a pipeline needs to process W threads of N dependent instructions is given by:

$$t = \begin{cases} N\Lambda + (W-1)\lambda \; if \; W \leq \frac{\Lambda}{\lambda} \\ \Lambda + (NW-1)\lambda \; if \; W \geq \frac{\Lambda}{\lambda} \end{cases} \tag{1}$$

In the presence of insufficient threads and for sufficiently large values of N and $\frac{\Lambda}{\lambda}$ the run time is dominated by the completion latency. Given sufficient threads, the pipeline operates at its peak performance delivering one instruction every λ cycles.

The above concepts are extended to model a compute unit as a pipeline that processes warp instructions. A warp is represented by an instruction dependence graph in which the nodes represent its instructions and the edges the dependences between them. To simulate the timing behavior of the code on a GPU, the nodes are labeled with their instruction and completion latency on that GPU. The following subsections discuss the components of the model in more detail.

B. Latencies

The latencies of the model are not necessarily equal to those of the hardware for a number of reasons. First, because threads execute in SIMT fashion, we use the warp instruction as the basic scheduling unit. Therefore, the instruction rate derived from λ must be multiplied by the warp size to obtain the maximal instruction rate. On NVIDIA GPUs, $\lambda = 1$ corresponds to 32 instructions per cycle per compute unit, on AMD GPUs to 64. Secondly, to keep our model generic, we consider instructions found in the OpenCL or CUDA kernels. Often, such instructions are translated to many hardware instructions. Finally, the same instructions may have different latencies in different kernels. A typical example is a memory instruction for which the latencies depend on the memory access pattern, caching and contention between warps.

Latencies are determined using micro-benchmarks as explained in [20]. A kernel that consists of a great number of dependent instructions of the type under investigation is executed for a variety of occupancies. We expect the run time to have a boat-hull shape following Equation 1. The completion and issue latency are derived from the maximal and minimal run time respectively.

C. Instruction Dependence Graph

The Instruction Dependence Graph (IDG) represents data dependences between the instructions in the kernel. It is derived from a program dependence graph [21] by resolving all control dependences. Loops are resolved by replicating the nodes that make up one loop iteration N times, where N is the loop trip count. When doing so it is necessary to take into account loop unrolling, which may result in the removal of loop control instructions and an increase of ILP or MLP. Conditional statements are resolved by serializing their branches. Branches not taken by any thread of the warp are removed. Note that this may lead to the existence of different graphs for different warps. This possibility, however, is not yet explored in this work.

Fig. 1. A simple OpenCL kernel and its instruction dependence graph

```
__kernel void saxpy(
    float a,
    __global float * X,
    __global float * Y)
{
    int i = get_global_id(0);
    float x = X[i];
    float y = Y[i];
    float z = a * x + y;
    Y[i] = z;
}
```

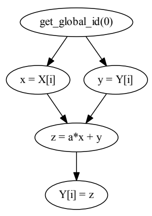

Figure 1 shows a simple kernel and its IDG. Given the appropriate latencies the IDG can be used to determine the minimum number of cycles to execute this kernel for one warp:

$$cycles = \Lambda_{index} + \Lambda_{mem} + \lambda_{mem} + \Lambda_{fmadd} + \lambda_{mem}$$

Note that in this equation all memory latencies are taken to be equal and that the warp is considered complete as soon as the final memory request has been issued. This principle is used to model the run time of a kernel on a GPU. However, because it is unfeasible to capture in analytical formulas the execution of many warps organized in groups with a certain occupancy, we use a simulator to mimic the timing behavior of the execution.

D. Timing Behavior Simulation

A simulator mimics the timing behavior of the kernel. This simulator takes as inputs the number of compute units P, the processor clock frequency f, the size and number of work groups that need to be processed N_{total}, the number of work groups that can be executed concurrently on a single compute unit M and an instruction dependence graph of the kernel with the latencies for the GPU under investigation. It is assumed that all work groups take more or less the same time to execute, allowing us to approximate the maximum number of work groups that need to be executed on a single compute unit: $N = \lceil \frac{N_{total}}{P} \rceil$.

The simulator will start with a set of M active work groups and schedule all instructions following the above scheme. While doing so, it keeps track of the number of cycles necessary. Of course, it takes into account the dependences between instructions, their issue latencies and also the maximum number of instructions that can be issued in one cycle. When a group completes, it is replaced by a new group if there are still groups waiting. This process is repeated until all groups have completed. The number of cycles is divided by f to attain the time in seconds.

IV. Validation

A. Setup

Algorithm 1 shows the pseudo-code of the simulator used to evaluate the model. This simulator treats a compute unit as consisting of different pipelines corresponding to the ALU, SFU, local memory and global memory subsystems and processes issue and completion events for each of them. It must be noted that this implementation does not take into account the maximum number of instructions that can be issued in one cycle. This is not a problem for the kernels analyzed in this paper, but it may cause an overly optimistic simulation for certain instruction mixes. Currently, we are developing a simulator that takes this limitation into account.

TABLE I
GPUs USED FOR THE VALIDATION

Architecture	Fermi	Pascal
Vendor	NVIDIA	NVIDIA
Name	Tesla C2050	GeForce GTX 1060 6GB
# compute units	14	10
# compute elements	448	1280
Base clock (MHz)	1150	1506

Algorithm 1 Timing Behavior Simulator

```
$waiting\ group\ count \leftarrow N - M$
$active\ group\ set \leftarrow \emptyset$
for $i = 0$ to $M$ do
    add a new group to $active\ group\ set$
end for
for all Pipeline do
    schedule an issue event at t = 0
end for
while $active\ group\ set$ is not empty do
    $event \leftarrow pop(event\_queue)$
    if is_issue(event) then
        I = look for instruction of the appropriate type
        if instruction is found then
            schedule a completion event at $t = t(event) + \Lambda(I)$
            schedule an issue event at $t = t(event) + \lambda(I)$
        else
            schedule an issue event at $t = t(event) + 1$
        end if
    else if is_completion(event) then
        $(group, warp, instruction) \leftarrow unpack(event)$
        complete(group, warp, instruction)
        if is_complete(group) then
            remove group from active groups
            if $waiting\ group\ count > 0$ then
                $waiting\ group\ count -= 1$
                add a new group to $active\ group\ set$
            end if
        end if
    end if
end while
```

The simulator was tested for a number of kernels on two NVIDIA GPUs of two different generations: a Tesla C2050 and a GeForce GTX 1060 6GB. Their characteristics are shown in Table I. In the remainder of this text we will refer to them by the names of their architecture: Fermi and Pascal.

To exhaustively test our model, which takes into account occupancy and group size, every kernel is run for a wide range of occupancies using different group size and count

TABLE II
MICRO-BENCHMARKS SIMULATIONS: LATENCIES, AVERAGE ERROR AND STANDARD DEVIATION

Benchmark	device	λ	Λ	μ_{err}	σ_{err}
$fadd$	Fermi	1	18	2.79	0.56
	Pascal	0.25	6	0.55	4.14
cos	Fermi	8	40	0.45	0.52
	Pascal	1	14	0.19	3.80
$loop$	Fermi	4	58	0.91	3.80
	Pascal	1.75	29	-1.91	2.25
$memory$	Fermi	23	521	3.74	4.18
	Pascal	12	378	11.59	7.19

combinations. Nevertheless, to limit somewhat the simulation space, group sizes were limited to powers of two greater than or equal to 32. For the Fermi GPU we explored 34 combinations for 18 different warp counts. For the Pascal GPU we explored 78 combinations for 43 warp counts.

B. Micro-benchmark Kernels

We start by simulating the benchmark kernels that are used to derive the latencies. Table II shows the latencies used for simulation and the average percentual error and standard deviation for a few characteristic benchmarks: floating point addition $fadd$, hardware accelerated cosine cos, loop overhead $loop$ and memory access $memory$. As can be seen the simulation is quite accurate for instructions that correspond to hardware instructions. Loop overhead is measured using a composed benchmark that executes a compute intensive loop, but in which the compiler is directed to refrain from loop unrolling using `#pragma unroll 1`. Finally, the memory benchmark tries to measure reading global memory with a perfectly aligned access pattern and without caching.

For the Pascal GPU the accuracy is not as good as for the Fermi GPU for a number of reasons. First, the Pascal GPU has a boost clock, which causes uncertainty about the frequency at which a kernel was run. Furthermore, fixing the frequency involves a very labour intensive setup as explained in [16]. Second, the effect of multiple warp schedulers, which is more important on the Pascal GPU, is not yet taken into account by our current simulator.

For both GPUs the simulation of the memory benchmark is less accurate. To understand why this is the case, it is useful to compare the real performance with the simulated performance. Figure 2 shows that memory access does not behave like a perfect pipeline: the maximum performance is reached at a higher occupancy than expected. According to [16] this is due to memory contention. Regardless of the cause, this phenomenon can be taken into account by making the latencies depend on the warp count.

Clearly, the differences between the measured and simulated run times reveal differences between our model and the real world behavior of a GPU. Therefore, their study can be used to get a more detailed understanding of what is happening on a particular GPU.

Listing 1. CUDA SDK matrix multiplication in OpenCL

```
#define BLOCK_SIZE 8

__kernel void
mmul_08(__global float* C,
        __global float* A,
        __global float* B,
        int WA,
        int WB,
        __local float* extra)
{
  __local float SA[BLOCK_SIZE][BLOCK_SIZE];
  __local float SB[BLOCK_SIZE][BLOCK_SIZE];

  int y = get_global_id(1);
  int x = get_global_id(0);
  int ly = get_local_id(1);
  int lx = get_local_id(0);
  int gy = get_group_id(1);
  int gx = get_group_id(0);

  int aIdx = y * WA + lx;
  int aDelta = BLOCK_SIZE;
  int aEnd = (y + 1) * WA;
  int bIdx = ly * WB + x;
  int bDelta = BLOCK_SIZE * WB;

  float sum = 0;
  for (; aIdx < aEnd; aIdx += aDelta, bIdx += bDelta) {
    SA[ly][lx] = A[aIdx];
    SB[ly][lx] = B[bIdx];
    barrier(CLK_LOCAL_MEM_FENCE);
    for (int k = 0; k < BLOCK_SIZE; ++k)
      sum += SA[ly][k] * SB[k][lx];
    barrier(CLK_LOCAL_MEM_FENCE);
  }

  C[y * WB + x] = sum;
}
```

C. Memory Matrix Multiplication

We now look at an OpenCL implementation of matrix multiplication from the CUDA SDK, which uses local memory (shared memory in CUDA terminology). Listing 1 shows the source code for 8x8 work groups. We use our model to simulate this kernel and a similar one for 16x16 work groups. The NVIDIA PTX code showed that for both kernels the inner loop was completely unrolled, while the outer loop was completely unrolled for the 8x8 case but only partially for the 16x16 case. To simplify the construction of the IDG we removed unrolling of the outer loop for both kernels using `#pragma unroll 1` with little effect on the run time. The unrolling of the inner loop introduces local memory level parallelism. The multiply-add instructions depend on two local memory reads and the previous multiply-add instruction for all but the first one. Note that only the loop and the writing of the result to global memory was included in the IDG.

Because the latencies of memory requests depend on their memory access pattern, caching and the occupancy of the kernel we wrote small benchmarks that mimic the data access of matrices A and B. In this case we derived a single issue and completion latency for the whole occupancy range and for each kind of access. Table III shows the latencies we used for simulation.

Both kernels were run on both GPUs to multiply two 1024x1024 matrices. Figure 3 compares the actual and simu-

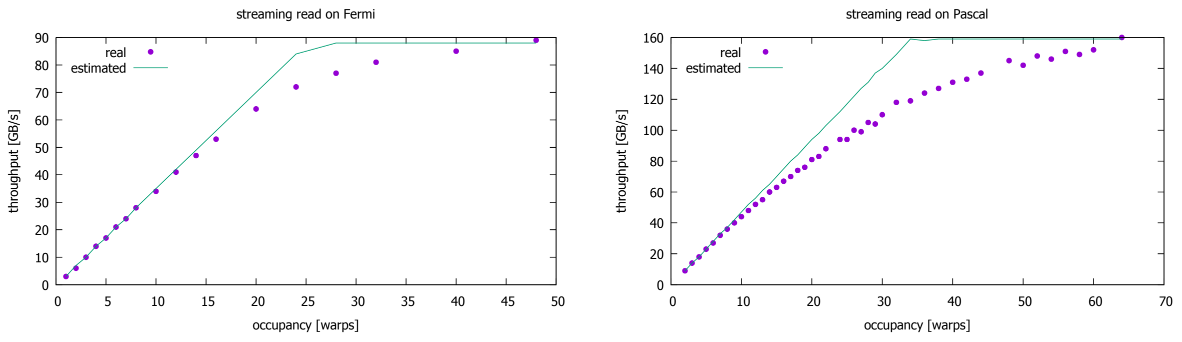

Fig. 2. Real and estimated performance of the streaming read benchmark

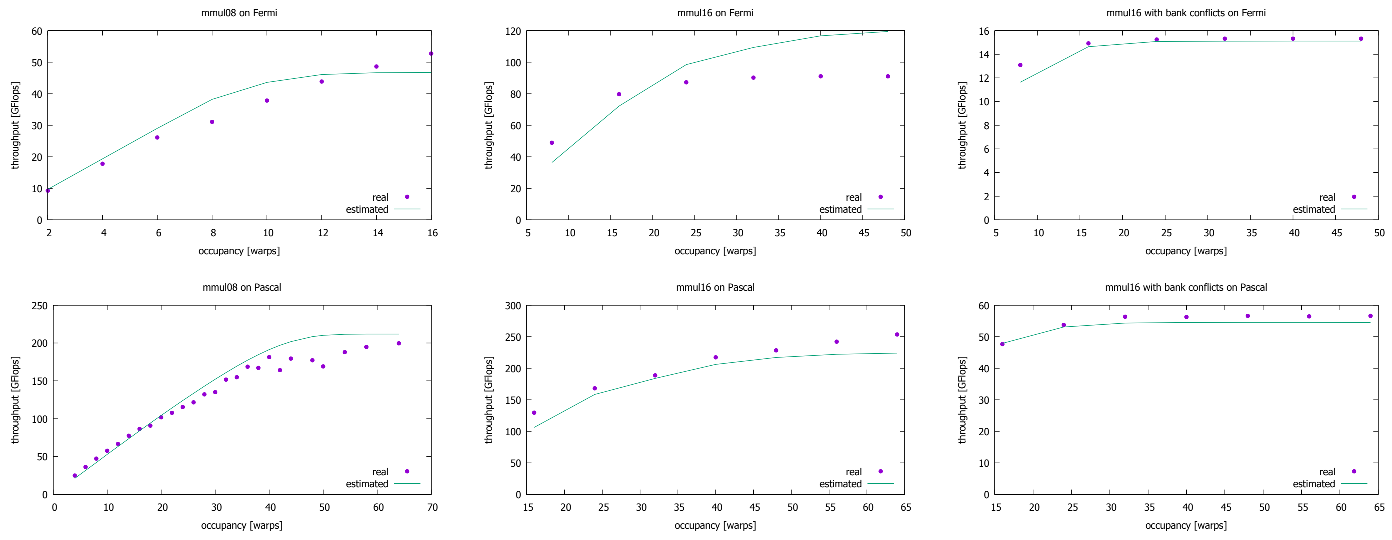

Fig. 3. Real and estimated performance of the matrix multiplication kernels

TABLE III
LATENCIES USED FOR THE MATRIX MULTIPLICATION SIMULATION

	Fermi		**Pascal**	
Operation	λ	Λ	λ	Λ
loop	4	58	1.75	29
local memory	2	47	1	30
matrix A 8x8	20	162	7	210
matrix B 8x8	45	479	8	409
matrix A 16x16	13	244	6	247
matrix B 16x16	17	561	10	444
barrier	3	40	2	70
fmadd	1	18	0.25	6

lated performance for each run. In this figure $mmul08$ and $mmul16$ refer to the matrix multiplication using 8x8 and 16x16 work groups respectively. We can see that for most runs our simulation predicts a performance that is quite close to the one that is actually achieved. For $mmul16$ on the Fermi our simulation overestimates the achieved performance. We believe the reason for this can be found in the latencies used for the global memory access.

Finally, to illustrate the capability of our model to quickly evaluate the impact of optimizations, we changed the local memory indices in the inner loop in order to introduce local memory bank conflicts. These occur when threads of the same warp read different elements in the same bank of local memory. This causes a decrease of local memory performance due to the need to serialize the access. In our model, this effect is taken into account by the association of the local memory access with higher latencies. For the Fermi GPU the latencies of the conflicting access were measured to be 16 and 250 cycles, while for the Pascal GPU they were measured to be 7 and 100 cycles. We replaced the original latencies by the new ones and reran the simulation. As can be seen in Figure 3 our simulation correctly takes into account the effect of the increased latency. As a matter of fact due to the fact that local memory has now become the bottleneck of the implementation the simulation is even more accurate.

V. Conclusion and Further Work

This paper introduced the pipeline performance model, a generic GPU performance model. It uses an instruction dependence graph of the code, which represents the dependences between the instructions and in which each instruction is characterized by its issue and completion latency. This representation together with the work configuration used to run the code and its occupancy is used to simulate the timing behavior of the code on the GPU under investigation.

The instruction latencies are determined with micro-benchmarks and can be reused for different kernels running on the same GPU. Similarly, the instruction dependence graph is derived from the OpenCL or CUDA source code and can be reused to simulate the same kernel on different GPUs using the appropriate latencies.

The model was tested on the micro-benchmarks used to determine the latencies and on a matrix multiplication kernel that uses local memory. All kernels were run for a wide occupancy range using various group sizes. Despite the differences between the real and simulated performance, we believe our model captures the main performance contributors. It was also showed that the greatest difficulty lies in using accurate latencies for memory instructions, which can be expected given the complexity of memory access. Finally, we showed that the model can be used to quickly asses whether a given instruction is the bottleneck by changing its latencies and rerunning the simulation. To validate this claim we increased the latencies of local memory access in the matrix multiplication kernel by introducing bank conflicts and showed that the simulation correctly predicts the deteriorated run time.

A lot of improvements to the model are foreseen. First, we want to automate the construction of the instruction dependence graph, which is currently a labour intensive process, using compiler technology. Secondly, we will further refine how we model memory instructions. For example, the effect of contention can be taken into account by letting the latencies be a function of the occupancy. Thirdly, we will update the simulator to take into account the instruction issue limit and test how our model performs on the instruction mix from [16], which is introduced as a performance model benchmark and for which many performance models fail to give good results.

References

[1] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *ACM Queue*, 6(2), 2008.

[2] John E. Stone, David Gohara, and Guochun Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science and Engineering*, 12(3), 2010.

[3] Samuel Williams, Andrew Waterman, and David A. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4), 2009.

[4] Haipeng Jia, Yunquan Zhang, Guoping Long, Jianliang Xu, Shengen Yan, and Yan Li. GPURoofline: a model for guiding performance optimizations on GPUs. In *European Conference on Parallel Processing (Euro-Par)*, 2012.

[5] Cedric Nugteren and Henk Corporaal. The boat hull model: enabling performance prediction for parallel computing prior to code development. In *Proceedings of the 9th conference on Computing Frontiers*, 2012.

[6] Elias Konstantinidis and Yiannis Cotronis. A practical performance model for compute and memory bound GPU kernels. In *23rd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2015.

[7] Sunpyo Hong and Hyesoon Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *36th International Symposium on Computer Architecture (ISCA)*, 2009.

[8] Sara S Baghsorkhi, Matthieu Delahaye, Sanjay J Patel, William D Gropp, and Wen-mei W Hwu. An adaptive performance modeling tool for GPU architectures. In *ACM Sigplan Notices*, volume 45, 2010.

[9] Jaewoong Sim, Aniruddha Dasgupta, Hyesoon Kim, and Richard Vuduc. A performance analysis framework for identifying potential benefits in GPGPU applications. In *ACM SIGPLAN Notices*, volume 47, 2012.

[10] Yao Zhang and John D Owens. A quantitative performance analysis model for GPU architectures. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2011.

[11] Jen-Cheng Huang, Joo Hwan Lee, Hyesoon Kim, and Hsien-Hsin S Lee. GPUMech: GPU performance modeling technique based on interval analysis. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

[12] Kishore Punniyamurthy, Behzad Boroujerdian, and Andreas Gerstlauer. GATSim: abstract timing simulation of GPUs. In *Proceedings of the Conference on Design, Automation & Test in Europe*, 2017.

[13] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009.

[14] Jan G Cornelis and Jan Lemeire. Benchmarks based on anti-parallel patterns for the evaluation of GPUs. In *PARCO*, 2011.

[15] Michael Andersch, Jan Lucas, Mauricio A LvLvarez-Mesa, and Ben Juurlink. On latency in GPU throughput microarchitectures. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2015.

[16] Vasily Volkov. *Understanding Latency Hiding on GPUs*. PhD thesis, EECS Department, University of California, Berkeley, 2016.

[17] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2017.

[18] NVIDIA Corporation. *CUDA C PROGRAMMING GUIDE*. NVIDIA Corporation, 2018.

[19] AMD Corporation. *AMD GRAPHICS CORES NEXT (GCN) ARCHITECTURE*. AMD Corporation, 2012.

[20] Jan Lemeire, Jan G Cornelis, and Laurent Segers. Microbenchmarks for GPU characteristics: The occupancy roofline and the pipeline model. In *24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, 2016.

[21] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3), 1987.