Preprint accepted for Journal of Parallel and Distributed Computing
Volume 173, March 2023, Pages 32-47

Analysis of the Analytical Performance Models for GPUs and Extracting the Underlying Pipeline Model.

Jan Lemeire[a,b,*], Jan G. Cornelis[b], Elias Konstantinidis[c]

[a]*Dept. of Industrial Sciences (INDI), Vrije Universiteit Brussel (VUB), Pleinlaan 2, B-1050 Brussels, Belgium*
[b]*Dept. of Electronics and Informatics (ETRO), Vrije Universiteit Brussel (VUB), Pleinlaan 2, B-1050 Brussels, Belgium*
[c]*Dept. of Informatics and Telecommunications, National and Kapodistrian University of Athens, Greece*

Abstract

This work presents an in-depth study of the analytical models for the performance estimation of GPUs. We show that the models' analytical equations can be derived from a pipeline analogy that models each GPU subsystem as an abstract pipeline. We call this the Pipeline model. All the equations are reformulated based on generic pipeline characteristics, namely throughput and latency. Our analysis shows equivalences between models and reveals substantial problems with some of the equations. Rather than relying on equations, the Pipeline model is then used to simulate the behavior of kernel executions based on the same hardware parameters as the analytical models. The simplicity of the model and relying on simulation mean that this approach needs less assumptions, is more comprehensive and is more flexible. More performance aspects can be taken into consideration. The different models are compared and evaluated empirically with 14 kernels of the Rodinia benchmark suite with varying occupancy. The Pipeline model gives an average MAPE of 24, while the average MAPE values of the other models lie between 27 and 136.

Keywords: GPU Computing, GPU architecture, Analytical Performance Models

1. Introduction

Although GPUs offer tremendous potential performance, the current implementations of non-trivial algorithms only reach a small portion of the peak performance. In particular, the efficiency of their execution is degraded by many performance limiters. A performance model is often used to predict and understand the performance and efficiency of GPU programs. Several models have been proposed in the last decade, for overviews see Lopez-Novoa et al. (2015) and Madougou et al. (2016). This paper focuses on the *analytical models*, from which we extract the essential concepts on which their equations are—often implicitly—based. These concepts are used to construct the novel Pipeline model, which provides a unifying framework in which existing and new modeling research can be situated.

The models try to estimate the performance of the execution of code, called **kernel**, by a great number of threads on massively parallel hardware. An instance of a kernel execution is called a **kernel thread** or work item. Processors consist of a number of **cores**, which for the greatest part operate independently from each other. On NVIDIA GPUs, cores are referred to as streaming multiprocessors (and a CUDA core is a processing unit within the multiprocessor), while AMD uses the OpenCL term Compute Unit. The basic instruction stream is the **hardware thread** or alternatively **warp**, which is also the term used by NVIDIA. AMD uses the term **wavefront**. NVIDIA and AMD GPUs execute in Single Instruction Multiple Thread (SIMT) fashion in which multiple kernel threads (32 or 64) share the same instruction counter and consequently operate in lockstep. Because the hardware thread is the scheduling unit, the instructions modeled in the following are warp instructions rather than kernel thread instructions.

The proposed model uses the *pipeline analogy* to model the subsystems of a GPU core. A pipeline consists of multiple stages each performing a part of the execution of an instruction or memory transaction. A pipeline is characterized by its throughput and latency, respectively by the rate at which instructions can be executed and the time that it takes to complete an instruction. The inverse of the throughput we denote with λ (Cycles Per Instruction; CPI) and the end-to-end latency with Λ. Simple pipelines can issue an instruction each cycle and the latency equals the number of pipeline stages. In our model, these are abstract parameters that should not correspond to whole numbers of cycles nor should they be constants. They also capture inefficient execution of instructions, such as cache misses or serialization of concurrent memory transactions due to bank conflicts.

The memory subsystem can also be modelled as a pipeline. A memory instruction initiates a memory transaction in the memory subsystem. Each memory transaction can also characterized by its *throughput* (the bandwidth) and by its completion latency. These parameters are not constants but can vary to take the various performance aspects into account such as caching, uncoalesced access or memory contention.

A major consequence of pipeline execution is the need for sufficient independent instructions that can be executed concurrently to keep the pipeline busy. The amount of independent instructions within a thread is called Instruction Level Par-

*Corresponding author
Email address: jan.lemeire@vub.be (Jan Lemeire)

allelism (ILP). Furthermore, instructions of different threads are independent except for barrier instructions. Hence, hardware threads (warps in GPUs) that run concurrently can fill the pipeline with their independent instructions. The concurrent execution of warps is referred to as Warp Level Parallelism (WLP) or Thread Level Parallelism (TLP). We define *occupancy* as the number of hardware threads or *warps* that are executed concurrently on a GPU for a specific kernel and configuration. This is different than some of the literature where occupancy is defined as the *ratio* of concurrent warps to the maximum number of concurrent warps (which depends on the GPU architecture).

ILP multiplied by WLP is the *instruction occupancy*. For a single pipeline with parameters λ and Λ and with an instruction occupancy of 1, the throughput is λ/Λ times the maximal throughput of the GPU. The instruction occupancy should be at least Λ/λ to attain the maximal throughput. The plot of the throughput as a function of the occupancy is called the *occupancy roofline* (Volkov, 2018). When the occupancy is smaller than the ridge point, GPU kernel executions do not attain the 'roof'. They are then called latency-bound kernels because the performance is bounded by latencies which cannot be fully hidden by concurrent instructions. Consequently, the pipeline analogy naturally incorporates the effect of imperfect latency hiding.

We will show that most analytical models are based on this analogy by demonstrating that the equations of these models can be extracted from the pipeline analogy given the appropriate assumptions and simplifications. The Pipeline model can be used to analyse the performance. However, instead of relying on equations, the timing behavior of the execution of a GPU kernel is mimicked by a simulation of the execution of the instructions on the abstract pipelines for which the inverse of throughput is used as an issue latency. The advantage of simulation is that we can avoid complex equations, which hamper understandability and extensibility. A Java implementation of microbenchmarks and Pipeline model is available at `www.gpuperformance.org`.

This paper claims the following contributions:

- First in-depth analysis of current analytical models: interpretation of the sometimes complex equations, extraction of the underlying model, expression of the equations in terms of general pipeline characteristics, revelation of errors and problems, and proofs of equivalences between models.
- Definition of the Pipeline model which can be used for simulating GPU execution. Simulation allows the quantitative and qualitative analysis of more performance aspects than the analytical models.
- Empirical validation of the models with the Rodinia benchmark suite. We are the first to do this by varying the occupancy, while until now the kernels have only been tested with the given configuration and occupancy.

In the next section we analyze the existing analytical models in terms of the pipeline analogy. From this analysis, the novel Pipeline model is defined in section 3. Section 4 discusses the components of the model. The models are then compared in section 5. Finally, section 6 empirically validates the Pipeline model.

Table 1: Model classes captured by the proposed Pipeline Model and the models present in the class. The main model of each class is shown in bold.

Model classes	Member models
Roofline model	**Williams et al. (2009)**, Boat Hull model (Nugteren and Corporaal, 2012), Konstantinidis and Cotronis (2015, 2017)
Volkov's model	**Volkov (2016)**, Transit (Li et al., 2015), X-model (Li et al., 2016)
MWP-CWP	**Hong and Kim (2009)**, GPUPerf (Sim et al., 2012)
WFG Model	**Baghsorkhi et al. (2010)**
GPUMech	**Huang et al. (2014)**, MDM (Wang et al., 2019)

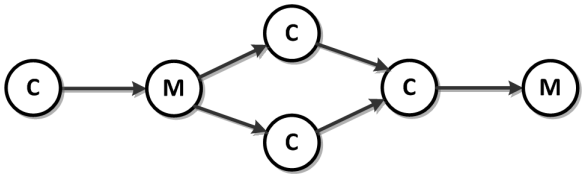

Figure 1: Example Kernel with 4 computational instructions (C) and 2 memory instructions (M). The edges represent the dependencies.

2. Analysis of the Analytical Models

The analytical models can be categorized into five main classes which are shown Table 1. Each will be rephrased in terms of the pipeline analogy and the parameters defined in Table 2. For demonstration purpose, we will apply each model to the simple GPU kernel shown in Fig. 1. The example kernel contains four computational instructions and two memory transactions. The edges of the graph represent the dependencies. The parameters used for the example are shown in the second column of Table 2. Note that the communication performance is not specified in Bytes, but is given in terms of *warp memory instructions per cycle*.
All of the models estimate the performance of a single core which is then multiplied with the number of cores of the GPU. The total number of threads is assumed to be evenly divided among the cores.

2.1. The Roofline Model

The Roofline model (Williams et al., 2009) gives a bound for the performance that can be attained for some program. It considers the throughput of the compute and memory subsystems. One of them will bound the overall performance, depending on the Compute Intensity (CI) of the GPU kernel. The CI is expressed as computational operations per byte, but for simplicity we will define it here as the amount of computations per memory operation. This results in a roofline-shaped graph of the computational throughput as a function of the compute intensity. Programs whose CI is to the left of the ridge are memory

Table 2: Parameters used to discuss the models and the values of the example.

Symbol	Value	Name
Hardware Parameters		
IPC_{comp}	1	computation instruction throughput
λ_{comp}	1	computational CPI = $1/IPC_{\text{comp}}$
Λ_{comp}	4	latency of computational instructions
BW	0.5	memory bandwidth = IPC_{mem}
λ_{mem}	2	memory CPI = $1/BW$
Λ_{mem}	6	latency of memory instructions
$\lambda_{\text{mem}}^{\text{instr}}$	1	memory instruction CPI (Section 2.4.1)
Software Characterization		
α_{comp}	4	number of computational instructions
α_{mem}	2	number of memory instructions
CI	2	Compute intensity = $\alpha_{\text{comp}}/\alpha_{\text{mem}}$
Execution Configuration		
ω	-	occupancy
Performance of a kernel running on a GPU core		
Λ_{app}	-	cycles of 1 warp executed in isolation
IPC_{app}	-	IPC of application
WPC_{app}	-	Warp throughput
CPR_{app}	-	Cycles Per Run (1 run = ω warps)

bound, while those whose CI is to the right of the ridge are compute bound.

$$IPC_{\text{app}}^{\text{roof}} = \min(IPC_{\text{comp}}, BW * CI) \tag{1}$$

Discussion. The Roofline model considers two subsystems (compute and memory) that are assumed to operate independently. All of the instructions can completely overlap and all latencies are hidden. The performance is defined as a throughput. The runtime can be calculated by counting the number of compute instructions or bytes to be transferred from memory. By considering the inverse of the throughputs, the ridge point is $\lambda_{\text{mem}}/\lambda_{\text{comp}}$, which is two for our example. Since the CI is also two, the performance of the example is exactly at the ridge point. $IPC_{\text{app}}^{\text{roof}}$ is one which gives a warp throughput WPC_{app} of 1/4.

Related models. The Boat Hull Model (Nugteren and Corporaal, 2012) is a related approach which considers application classes for better precision. The Roofline model can be extended to a mix of instructions of different types with different throughputs (Konstantinidis and Cotronis, 2015, 2017).

2.2. *Volkov's model*

The model developed by Volkov (2016) estimates the warp throughput for both the throughput bound and latency bound case, resulting in an *occupancy roofline* for the *whole* kernel. The throughput bound runtime of one warp is calculated by taking the maximum of the times needed by each subsystem to execute this warp. The time needed by a subsystem is obtained by adding the throughput CPIs for all the instructions executed on that subsystem. This is similar to the approach of the roofline model. For the latency bound case, the time needed to execute a

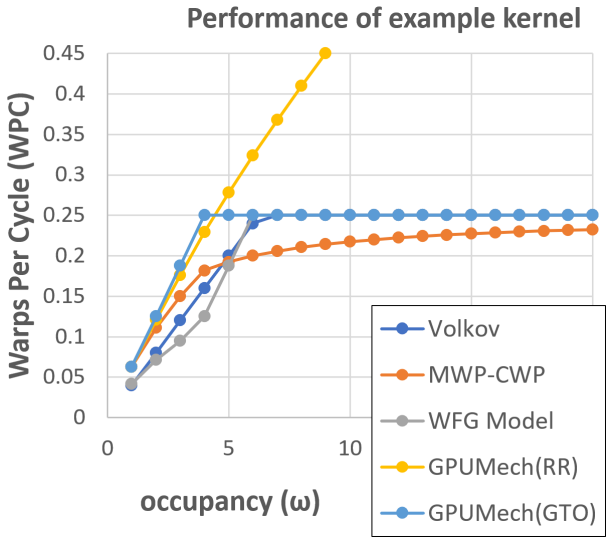

Figure 2: The performance estimation of the example kernel for the different models.

single warp is mostly estimated with latencies, while also considering overlap when there are independent instructions and the GPU has the capacity for ILP within a single thread. Then, the instruction's CPI is considered. This results in:

$$\Lambda_{\text{app}} = \sum_{i \in I} \Lambda_i^{\text{dep}} + \sum_{j \in J} \lambda_j^{\text{indep}} \tag{2}$$

where set J contains the instructions that can run in parallel with other instructions, which are contained in set I. For the warp throughput, the time is divided by the occupancy and the inverse is taken. This gives the left-hand side of the roof.

$$WPC^{\text{volkov}} = \min(WPC^{\text{roof}}, \omega/\Lambda_{\text{app}}) \tag{3}$$

with $WPC^{\text{roof}} = IPC_{\text{app}}^{\text{roof}}/\alpha^{\text{roof}}$ and α^{roof} the number of instructions executed on the bounding subsystem per warp.

The resulting occupancy roofline for the example is shown in Fig. 2. $\Lambda_{\text{app}} = 25$ which gives a WPC of 1/25 for one warp and which reaches the peak of 1/4 for seven concurrent warps.

To model memory contention, Volkov uses a contention function which smoothens the curve around the ridge point (Volkov, 2016, Sec. 6.4). This is not modeled because it is not a generic solution, but based on the regression of a function on data of specific streaming kernels that employ perfectly-aligned memory access. Later we will refer to several generic methods that model memory contention.

Discussion. Volkov takes into account the necessity of sufficient concurrent threads (the occupancy) to keep all subsystems busy (such that they overlap) and hide the pipeline latencies. If there is only 1 hardware thread active and no ILP, the runtime of a warp is equal to the sum of all pipeline latencies. The execution of additional threads running concurrently is assumed to be *completely hidden behind the first thread* until the maximal throughput is achieved. Consequently, the model gives an upper bound of the performance under limited occupancy.

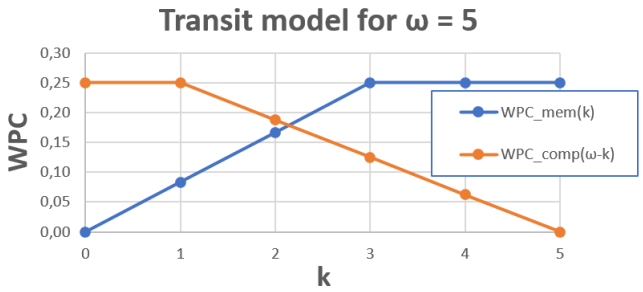

Figure 3: The Transit model applied to the example for $\omega = 5$. The computational roofline WPC_{comp} is reversed and its origin is placed at $k = \omega = 5$.

2.2.1. Equivalence of Volkov and Transit

Like Volkov's model, the Transit model (Li et al., 2015) provides a global view on the performance while taking the occupancy into account. Here, we will show that they are equivalent.

Transit represents the hardware by a computational and a memory subsystem. Both are characterized by an occupancy roofline which is measured separately with an appropriate microbenchmark. The software is characterized by its arithmetic intensity, the number of memory operations and the total number of threads executing concurrently i.e. the occupancy ω. The software imposes a service demand on the computational subsystem which in turn imposes a service demand on the memory access subsystem. The number of concurrent warps is divided among both subsystems: a warp is either computing or waiting for a memory request to complete. In the steady state of the system, memory service demand $g(x)$ and memory service supply $f(k)$ are equal and correspond to the warp throughput. This can be shown graphically based on the two occupancy rooflines: The memory service demand $g(x)$ and the memory service supply $f(k)$ are put into a single graph but with the axes reversed and the origin of f is placed at ω of the x-axis of g (because $x + k = \omega$). The point at which the two plots cross gives the throughput of the considered workload. This also determines whether the code is latency bound, compute bound or memory bound. For our example and $\omega = 5$, the two occupancy curves are shown in Fig. 3.

Theorem 1. *The Volkov and Transit models generate the same occupancy roofline if not considering ILP.*

Proof. We start with the two occupancy rooflines:

$$IPC_{\text{comp}}(x) = \min(\frac{1}{\lambda_{\text{comp}}}, \frac{x}{\Lambda_{\text{comp}}}) \tag{4}$$

$$IPC_{\text{mem}}(k) = \min(\frac{1}{\lambda_{\text{mem}}}, \frac{k}{\Lambda_{\text{mem}}}) \tag{5}$$

For throughput bound code the crossing point of the two occupancy plots will lie 'on' the roof of the occupancy plot of the bounding subsystem. This is equivalent to the Volkov model, where the throughput bound performance is determined by the most bounding subsystem. For the occupancy bound case ($x < \Lambda_{comp}/\lambda_{comp}$ and $k < \Lambda_{mem}/\lambda_{mem}$) we express the performance as warps per cycle :

$$WPC_{\text{comp}}(x) = \frac{x}{\Lambda_{\text{comp}}.\alpha_{\text{comp}}} \tag{6}$$

$$WPC_{\text{mem}}(k) = \frac{k}{\Lambda_{\text{mem}}.\alpha_{\text{mem}}} \tag{7}$$

Supply must match demand. For a given occupancy, k of the ω warps are busy with memory access, the others with computations:

$$WPC_{\text{mem}}(k) = WPC_{\text{comp}}(\omega - k) \tag{8}$$

$$\Leftrightarrow \frac{k}{\Lambda_{\text{comp}}.\alpha_{\text{comp}}} = \frac{\omega - k}{\Lambda_{\text{mem}}.\alpha_{\text{mem}}} \tag{9}$$

$$\Leftrightarrow k.(\frac{CI}{\Lambda_{\text{mem}}} + \frac{1}{\alpha_{\text{comp}}}) = \frac{\omega}{\Lambda_{\text{comp}}} \tag{10}$$

$$\Leftrightarrow k = \frac{\omega}{1 + CI.\Lambda_{\text{comp}}/\Lambda_{\text{mem}}} \tag{11}$$

By replacing k in Eq. 7, we get:

$$WPC_{\text{mem}}(k) = \frac{\omega}{(1 + CI.\Lambda_{\text{comp}}/\Lambda_{\text{mem}}).\Lambda_{\text{mem}}.\alpha_{\text{mem}}} \tag{12}$$

$$= \frac{\omega}{\Lambda_{\text{mem}}.\alpha_{\text{mem}} + \Lambda_{\text{comp}}.\alpha_{\text{comp}}} \tag{13}$$

$$= \frac{\omega}{\Lambda_{\text{app}}} = WPC(\omega) = WPC^{\text{volkov}}(\omega) \tag{14}$$

which is exactly the same as given by Volkov's model. □

2.3. The MWP-CWP Model

In contrast to the models discussed so far which are based on a global view, the following two models take the details of instruction sequence and execution into account by extracting equations that either (a) model the execution (the MWP-CWP model) or (b) consider the exact sequence of instructions (the WFG model, discussed in the next section).

The MWP-CWP Model (Hong and Kim, 2009), approximates the software by a code consisting of P compute periods made up of Q instructions followed by a memory request. The number of compute periods is determined by the number of memory requests, thus $P = \alpha_{\text{mem}}$. Because compute periods are assumed to be equal, it follows that $Q = CI$.

Analytical equations for the timing behavior are derived by considering three distinct cases which are based on the concepts of MWP and CWP.

2.3.1. Interpretation of MWP and CWP

Memory Warp Parallelism (MWP) is the number of warps that can access memory simultaneously. Therefore, it is :

$$MWP = \Lambda_{\text{mem}}.BW = \Lambda_{\text{mem}}/\lambda_{\text{mem}} \tag{15}$$

Meanwhile, Compute Warp Parallelism (CWP) is defined as the number of warps that can be executed during the memory waiting time of one warp plus one to include the warp that is waiting for memory:

$$CWP = \frac{\Lambda_{\text{mem}}}{CI.\lambda_{\text{comp}}} + 1 \tag{16}$$

When $\omega \geq MWP$ and $\omega \geq CWP$, the occupancy is large enough to achieve maximal throughput. Then, according to the model, CWP and MWP must be compared. Let us derive the meaning of this comparison:

$$MWP > CWP \tag{17}$$

$$\Leftrightarrow \Lambda_{\text{mem}}/\lambda_{\text{mem}} > \frac{\Lambda_{\text{mem}}}{CI.\lambda_{\text{comp}}} + 1 \tag{18}$$

$$\Leftrightarrow (\Lambda_{\text{mem}} - \lambda_{\text{mem}}).CI > \frac{\Lambda_{\text{mem}}.\lambda_{\text{mem}}}{\lambda_{\text{comp}}} \tag{19}$$

$$\Leftrightarrow CI > \frac{\lambda_{\text{mem}}}{\lambda_{\text{comp}}.(1 - \lambda_{\text{mem}}/\Lambda_{\text{mem}})} \tag{20}$$

According to the roofline model, a program's CI must be greater than $\lambda_{\text{mem}}/\lambda_{\text{comp}}$ for it to be compute bound. This is nearly the same condition as Equation (20). Here, CI has to be a little bigger, due to the fact that we do not assume perfect overlap of computations and memory: the memory request can only be initiated when the computations have finished. An additional warp is needed to keep the computational pipeline busy, hence the +1 in the definition of CWP (Eq. 16).

Hong calculates for the three cases the runtime of ω warps which we call a *run*.

The memory-bound case: $MWP < \min(\omega, CWP)$.
First, we map the model's concepts onto the pipeline concepts:

$$Mem\_p = \Lambda_{\text{mem}} \tag{21}$$

$$Comp\_p = CI.\lambda_{\text{comp}} \tag{22}$$

$$Mem\_cycles = \alpha_{\text{mem}}.\Lambda_{\text{mem}} \tag{23}$$

$$Comp\_cycles = \alpha_{\text{mem}}.CI.\lambda_{\text{comp}} \tag{24}$$

$$N = \omega \tag{25}$$

Eq. 1 of (Hong and Kim, 2009) for the runtime of ω warps in Cycles Per Run (CPR) can now be written as:

$$CPR = \alpha_{\text{mem}}.\omega.\lambda_{\text{mem}} + CI.\lambda_{\text{comp}}.MWP \tag{26}$$

The first term is the memory access time at peak performance and similar to the roofline model. The second term is the computation pipeline fill time, see Hong and Kim (2009, Fig. 6).

The compute-bound case: $CWP < \min(\omega, MWP)$.
Eq. 3 of Hong and Kim (2009) results in (see Hong and Kim (2009, Fig. 5)):

$$\begin{aligned} CPR &= Comp\_cycles.\omega + Mem\_p \\ &= \alpha_{\text{comp}}.\lambda_{\text{comp}}.\omega + \Lambda_{\text{mem}} \end{aligned} \tag{27}$$

The runtime is now the computation time (first term) plus the memory pipeline drain time (second term).

The occupancy-bound case: $\omega \leq \min(MWP, CWP)$.
For this case MWP and CWP are set to ω. Eq. 4 of Hong and Kim (2009) results in (see (Hong and Kim, 2009, Fig. 8)):

$$\begin{aligned} CPR &= (Mem\_cycles + Comp\_cycles) + Comp\_p.(\omega - 1) \\ &= (\alpha_{\text{mem}}.\Lambda_{\text{mem}} + \alpha_{\text{comp}}.\lambda_{\text{comp}}) + CI.\lambda_{\text{comp}}.(\omega - 1) \\ &= \Lambda^*_{\text{app}} + PipelineFillTime \end{aligned} \tag{28}$$

The MWP-CWP model only takes the throughput of computations into account and it ignores their latency. Therefore, we denote the execution of a single warp by Λ^*_{app}.

We conclude that the MWP-CWP model adds the pipeline fill/drain time to the estimation of Volkov. Note that in the case of barriers, a similar pipeline drain time is taken into account for each barrier (Hong and Kim, 2009, Fig. 11).

2.3.2. Example

The example kernel is approximated by two periods of two instructions followed by a memory instruction. The kernel is memory bound since MWP = 3 which is smaller than CWP = 6/2+1 = 4. The performance in function of occupancy is shown in Fig. 2. For low occupancies, the performance is higher than Volkov's model because the computational latencies are not taken into account.

2.3.3. Discussion.

Consider the assumptions of the model about the code's execution. The timing behavior of instructions and memory transactions in the subsystems is stated explicitly: when they 'occupy' the subsystem – by their throughput – and how long it takes before a dependent instruction or memory transaction can start – their latency. More specifically, the computational instructions of a warp keep the computational subsystem busy for $CI.\lambda_{\text{comp}}$ cycles, after which a memory request is started. The memory instructions result in a transaction handled by a separate memory subsystem.

2.3.4. Memory Departure Delay.

For memory instructions, the MWP-CWP model defines a departure delay as the minimum departure distance between two consecutive memory transactions. Furthermore, the bandwidth limits the maximal number of warps that can access memory simultaneously (during one memory warp waiting period $mem\_p = \Lambda_{\text{mem}}$). In the Pipeline model, a second memory transaction can only start in the memory subsystem after the CPI (inverse of throughput) which we use as instruction latency. The CPI determines the time between consecutive starts of transactions in the subsystem. In our model, the MWP-CWP's departure delay is taken into account by the issue limit (which will be described in more detail later on).

This difference is shown in Fig. 4. In the MWP-CWP model, with three concurrent memory transactions, the bandwidth is attained such that the fourth memory transaction must wait until the first transaction finishes. The Pipeline model uses the CPI as an 'issue latency' in the simulation such that the second and third memory transactions start later. Note that according to the MWP-CWP model the first three transactions finish at a rate larger than the bandwidth. By only considering the CPI, the second term of Eq. 26 would change into:

$$PipelineFillTime = CI.\lambda_{\text{comp}} + (MWP - 1).\lambda_{\text{mem}} \tag{29}$$

Note that the latencies of the MWP-CWP model take inefficiencies into account, such as uncoalesced memory access. They should be considered as abstract values capturing the total behavior of an instruction.

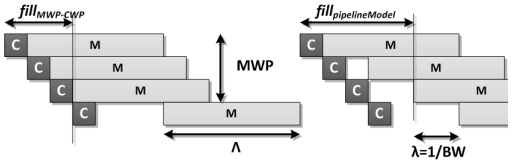

Figure 4: The difference between the MWP-CWP model (left) and the Pipeline model (right).

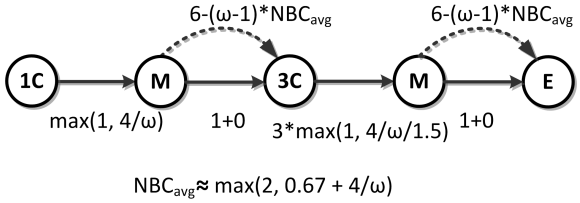

Figure 5: The WFG for the example kernel with the latencies for each edge as a function of the occupancy.

2.4. The WFG Model

The WFG model (Baghsorkhi et al., 2010) models the software in detail with a *Work Flow Graph (WFG)*. A node of this graph is either a block of computational instructions, a memory transaction or a barrier synchronization point. There are two types of edges: first, transition arcs represent the order of instruction execution (branching is possible, but not considered in this paper); and second, data-dependence arcs represent the data dependence of one node on another as a result of a memory transaction. Thus, contrary to previous models, this model takes into account the actual sequence of instructions and the dependencies among them.

2.4.1. Discussion of equations

A latency is attributed to each edge:

- *Transition arcs leaving computational instructions* (Baghsorkhi et al., 2010, Eq. 3): the CPI plus part of the completion latency that cannot be hidden by independent instructions (WLP or ILP). (Baghsorkhi et al., 2010, Eq. 3). If there is sufficient parallelism, then the pipeline latency is completely hidden and only the CPI should be considered. The equation for the latency of edge i is similar to Volkov's model:

$$\lambda^{\text{trans,i}}_{\text{comp}} = \alpha^{i}_{\text{comp}}.\max(\lambda_{\text{comp}}, \frac{\Lambda_{\text{comp}}}{ILP^{i}.\omega}) \quad (30)$$

  with α^{i}_{comp} the number of computations of the node and ILP^{i} the ILP of the block of code.

- *Transition arcs leaving a memory operation* (Baghsorkhi et al., 2010, Eq. 4): memory instructions impose a CPI ($\lambda^{\text{instr}}_{\text{mem}} = \lambda_{\text{comp}}$) plus the latency that is not covered by the total number of compute cycles CYC_{comp} of the kernel:

$$CYC_{\text{comp}} = \sum_{i} \lambda^{\text{trans,i}}_{\text{comp}}.\alpha^{i}_{\text{comp}} + \lambda^{\text{instr}}_{\text{mem}}.\alpha_{\text{mem}} \quad (31)$$

$$= \bar{\lambda}^{\text{trans}}_{\text{comp}}.\alpha_{\text{comp}} + \lambda^{\text{instr}}_{\text{mem}}.\alpha_{\text{mem}} \quad (32)$$

  where $\bar{\lambda}^{\text{trans}}_{\text{comp}}$ is a weighted average of the transition computational latencies and $\alpha_{\text{comp}} = \sum_i \alpha^{i}_{\text{comp}}$. To compute the memory's CPI that is hidden, all compute cycles are considered to be available for hiding:

$$\lambda^{\text{trans}}_{\text{mem}} = \lambda^{\text{instr}}_{\text{mem}} + \max(0, \frac{CYC_{\text{mem}} - CYC_{\text{comp}}}{\alpha_{\text{mem}}}) \quad (33)$$

$$= \lambda^{\text{instr}}_{\text{mem}} + \max(0, \frac{\alpha_{\text{mem}}/BW - (\bar{\lambda}^{\text{trans}}_{\text{comp}}.\alpha_{\text{comp}} + \lambda^{\text{instr}}_{\text{mem}}.\alpha_{\text{mem}})}{\alpha_{\text{mem}}})$$

$$= \lambda^{\text{instr}}_{\text{mem}} + \max(0, \lambda_{\text{mem}} - \bar{\lambda}^{\text{trans}}_{\text{comp}}.CI - \lambda^{\text{instr}}_{\text{mem}})$$

$$= \max(\lambda^{\text{instr}}_{\text{mem}}, \lambda_{\text{mem}} - \bar{\lambda}^{\text{trans}}_{\text{comp}}.CI) \quad (34)$$

  This equation is equivalent to having memory transactions being issued on a separate memory subsystem that overlap with latencies from the computational subsystem, but not with other memory latencies. Meanwhile, the memory instruction's CPI $\lambda^{\text{instr}}_{\text{mem}}$ cannot overlap with computational latencies. This assumes that the memory instruction is handled on the same subsystem as computations.

- *Dependency arcs leaving a memory operation*: the memory latency may not completely overlap with compute cycles of other concurrent warps. The equation that computes the number of non overlapped cycles, $Latency_{exposed}$, uses the total number of compute cycles divided by the number of memory operations and barriers plus one (Baghsorkhi et al., 2010, Eq. 5 and Eq. 6). The equations (given in the next subsection) express that the compute cycles are evenly 'distributed' among the memory instructions (resulting in an average). This expresses that dependent instructions have to wait for the latency to be finished. In the next subsection, we will discuss a problem with the given equations.

The WFG for the example kernel with the latencies for each edge is shown in Fig. 5.

Once the latencies are established, the graph can be reduced (flattened) by combining successive nodes. The tree is traversed and the latencies are summed. For memory nodes, the maximum latency of both edges leaving the node is taken. This gives the (average) runtime of a warp (CPW) and when multiplied with ω it gives the CPR.

Note that the WFG model also considers *work groups*. Warps are executed in work groups. The WFG model assumes that warps of the same work group proceed relatively synchronously, while warps of different work groups run asynchronously. The number of concurrent warps is based on a combination of both and is expressed as WLP_{effect}. In this paper, we will not consider this and simply assume $WLP_{\text{effect}} = \omega$.

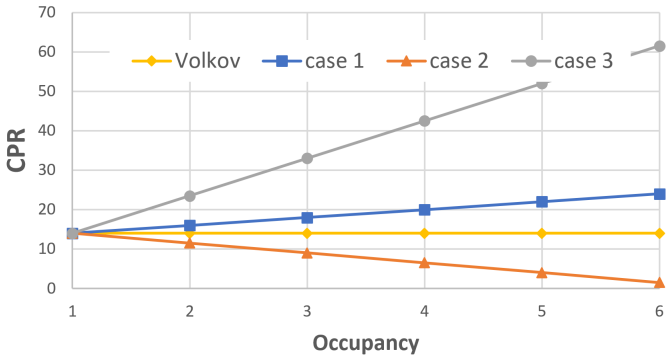

Figure 6: Examples showing the problems with the WFG model for the occupancy-bound case.

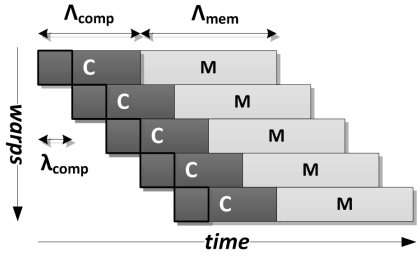

Figure 7: Overlap of λ_{comp} of the succeeding warps with the latencies of computations and memory.

2.4.2. *Problems with the equations*

The latency of the memory dependency arcs is calculated as the portion of the GPU memory latency that is not covered by interleaving execution of different warps. The equations of the WFG model are (Baghsorkhi et al., 2010, Eq. 5 and Eq. 6):

$$NBC_{\text{avg}} = \frac{CYC_{\text{comp}}}{\alpha_{\text{mem}} + \alpha_{\text{sync}} + 1} \tag{35}$$

$$\Lambda_{\text{exposed}} = \Lambda_{\text{mem}} - (\omega - 1).NBC_{\text{avg}} \tag{36}$$

where α_{sync} is the number of barrier synchronization points.

Volkov reported serious deviations of the WFG model with his results (Volkov, 2016). These deviations can be reproduced with a simple example. Consider that $\alpha_{\text{mem}} = 1$ and $ILP = 1$. Assume that for the computations, it is still occupancy-bound (Eq. 30) and that for the memory, the dependency arc has a larger latency than that of the transition arc ($\Lambda_{\text{exposed}} > \lambda^{\text{trans}}_{\text{mem}}$). When we neglect the small $\lambda^{\text{instr}}_{\text{mem}}$, the performance is (we omit index i):

$$CPW = \lambda^{\text{trans}}_{\text{comp}} + \Lambda_{\text{exposed}} \tag{37}$$

$$= \frac{\alpha_{\text{comp}}.\Lambda_{\text{comp}}}{\omega} + \Lambda_{\text{mem}} - (\omega - 1).\frac{\alpha_{\text{comp}}.\Lambda_{\text{comp}}}{2.\omega} \tag{38}$$

$$CPR = CPW.\omega \tag{39}$$

$$= \alpha_{\text{comp}}.\Lambda_{\text{comp}} + \Lambda_{\text{mem}}.\omega - (\omega - 1).\alpha_{\text{comp}}.\Lambda_{\text{comp}}/2 \tag{40}$$

Fig. 6 shows the results for three different parameter pairs. Volkov's model gives the same CPR for all pairs and all occupancies, namely 14. For case 1, $\alpha_{\text{comp}}.\Lambda_{\text{comp}} = 8$ and $\Lambda_{\text{mem}} = 6$ which is similar to the example kernel. Case 2 is compute-intensive, $\alpha_{\text{comp}}.\Lambda_{\text{comp}} = 11$ and $\Lambda_{\text{mem}} = 3$. The CPR drops for higher occupancies. This is impossible, the time for executing two or more warps cannot be lower than that for one. Meanwhile, case 3 has a high memory latency, $\alpha_{\text{comp}}.\Lambda_{\text{comp}} = 3$ and $\Lambda_{\text{mem}} = 11$, which results in a fast-increasing CPR. We first analyze the latter.

2.4.3. *First error in Eq. 6 of the WFG model*

Eq. 36 is trivially correct for one warp, but for multiple warps the average warp latency is multiplied by the number of warps ω to estimate the total runtime. However, for $\omega > 1$, the calculated $\Lambda_{exposed}$ must be taken into account only once, because the exposed latency of the first warp (Eq. 36) hides the memory latency of subsequent warps. When using Eq. 6 of Baghsorkhi et al. (2010), the exposed memory latency is multiplied by ω resulting in an overestimation of the runtime as long as the latency is not completely hidden. Therefore, we have to divide the exposed latency by the WLP, ω:

$$\Lambda_{\text{exposed}} = \frac{\Lambda_{\text{mem}} - (\omega - 1).NBC_{\text{avg}}}{\omega} \tag{41}$$

Note also that in Eq. 30 (Baghsorkhi et al., 2010, Eq. 3), the pipeline latencies of the computations are divided by ω.

2.4.4. *Second error in Eq. 6 of the WFG model*

Eq. 5 of Baghsorkhi et al. (2010) calculates the amount of computation cycles NBC_{avg} that can hide memory latencies. As can be seen in Eq. 35, *all* computation cycles are used to hide memory latencies but these latencies include those that already hide the computational latencies. The latter is already taken into account in Eq. 30. Consequently, it is possible for the runtime of a few warps to become shorter than that of one warp. In the example, one sees that the performance curve of Fig. 2 is slightly above Volkov's roofline for $\omega = 3$ or 4. This effect becomes more apparent with a higher CI.

As shown in Fig. 7, the λ_{comp} of warps 2 and 3 are overlapping with the computational latency of warp 1. Only warps 4 and 5 are overlapping with the memory latencies. One should subtract the overlap from computational latencies. We therefore propose the following equation for replacing (Baghsorkhi et al., 2010, Eq. 6):

$$\Lambda_{\text{exposed}} = \frac{\Lambda_{\text{mem}}}{\omega} - \max(0, \frac{\omega - 1}{\omega}.NBC_{\text{avg}} - \frac{\alpha_i.\Lambda_{\text{comp}}}{\omega.ILP^i}) \tag{42}$$

where the last term refers to the preceding computation transition arc.

This equation assumes that the memory instruction depends on the results of the computations. In case of independence, the computational cycles can indeed completely overlap with the memory cycles. But in that case one should also overlap them for a single warp, which is not the case because then $(\omega - 1) = 0$ in Eq. 36 and no overlap is considered.

2.5. *GPUMech*

GPUMech (Huang et al., 2014) uses interval analysis to estimate the amount of latency hiding that occurs. Interval analysis

is based on establishing steady state performance and then subtracting performance loss due to miss events (Eyerman et al., 2009).

The most representative warp is divided into intervals each ending with a dependency with a large latency which results in a stall period. The model proposes equations to estimate how many instructions of concurrent warps are issued during the stall periods depending on the scheduler: Round-Robin (RR) or Greedy-Then-Oldest (GTO). The resulting performance is then calculated with the number of non-overlapping instructions.
When implementing this part of the model, we did not consider latencies of computational instructions and applied the same corrections as Volkov (2016, p. 105), who confirmed the typos with the primary author: Eq. 7 calculates IPC instead of CPI, while in Eq. 15 and Eq. 16 max is used in place of min and vice versa.
Second, GPUMech models resource contention related to the MSHRs and the DRAM bandwidth by estimating the additional cycles which are then added to the global CPI. This memory access analysis is necessary to obtain the correct latencies of memory instructions. Since the evaluation of the models happen under the assumption that the correct latencies have been obtained, this part of the model is not implemented.

The results for both schedulers are shown in Fig. 2 and are similar to those reported by Volkov (2016, Fig. 7.9): with RR, the performance goes through the roof; while with GTO, the performance converges to Volkov's the roof. We now analyze both in more detail.

2.5.1. Problems with the Round-Robin equations.

Strangely enough, the equations of GPUMech result in a different estimation than shown by the given example (Huang et al., 2014, Fig. 8). The example in the paper shows an interval with 3 instructions and 6 stall cycles. Then the issue probability is 0.333 (Eq. 9; 3 divided by (3 plus 6)), and for 4 warps the number of non-overlapped instructions is 2 according to Eq. 11. But Fig. 8 show 6 non-overlapped instructions.
The problem arises with the *issue probability*: it assumes that there will not always be an instruction scheduled in the waiting slots, rather, the Round Robin policy ensures that after the issue of an instruction of the representative warp, instructions of the remaining warps are scheduled. We therefore have to set the issue probability to 1.

This results in one overlapped instruction for each remaining warp, which is in concordance with the text. But the number of overlapped instructions should be limited to the number of stall cycles. Otherwise the IPC becomes higher than 1 (while an issue rate of 1 is assumed). As a result we propose the following equations to replace Eq. 11 (*overl* stands for *overlapped*):

$$\#overl\_insts_i = \min(stall\_cycles_i, \#warps - 1)$$
$$\#nonoverl\_insts_i = (\#warps - 1) \times \#interval\_insts_i - \#overl\_insts_i$$

2.5.2. Equivalence of GPUMech GTO With Volkov

The equations for the GTO scheduler also rely on the issue probability (Eq. 9). Its value, however, is most often close to 1. In that case, GPUMech is equivalent to Volkov's model when ignoring computational latencies. With an issue probability of 1, *all* instructions of the remaining warps are used for overlapping with the stall cycles (Eq. 12 and Eq. 14) until all stall cycles are overlapped (Eq. 16). To show that the issue probability is rarely below 1, we replace the sums in nominator and denominator by the average in Eq. 9 and combine it with Eq. 15:

$$\begin{aligned} &\#issue\_prob\_in\_stall_i < 1 \\ \Leftrightarrow &\frac{avg\_interval\_instr \times stall\_cycles_i}{avg\_interval\_instr + avg\_stall\_cycles} < 1 \\ \Leftrightarrow &stall\_cycles_i < 1 + \frac{avg\_stall\_cycles}{avg\_interval\_instr} \end{aligned}$$

This will happen for intervals with a latency smaller than the average latency divided by the average number instructions of an interval; thus, a small portion of the intervals. Moreover, With small latencies, there is not much performance to lose, so that the overall performance will remain close to that of Volkov's model. Since computational latencies are not considered, the performance curve is slightly above that of Volkov in Fig. 2.

2.6. Extracting the underlying model

The analysis shows that all analytical models can be derived from an abstract pipeline analogy.

The Roofline model and Volkov's occupancy roofline provide an upper bound but do not capture detailed behavior. They do not take into account the interplay of the different threads and subsystems which gives rise to additional inefficiencies (mainly idling). The MWP-CWP model is based on a detailed execution profile, while the WFG model uses a detailed description of the kernel. However, both are limited because they strive to capture the behavior with analytical equations. To achieve this, the MWP-CWP model approximates the software, while the WFG model 'smears out' the compute cycles to estimate the latency hiding. Based on the detailed work flow graph, it is possible to determine exactly which compute cycles overlap with memory latencies. The Pipeline model does this by combining the execution profile of the MWP-CWP model with the detailed software description of the WFG model. The equations of GPUMech are based on two scheduling policies: RR and GTO. The scheduler is explicitly added as a parameter to the Pipeline model and will be used to simulate kernel execution.

3. The Pipeline Performance Model

This section defines the Pipeline model. Table 3 summarizes both the input parameters and the functions of the pipeline performance model.

3.1. Model Input

The input for the Pipeline model consists of 3 components: software, hardware and execution configuration.

Table 3: Pipeline performance model summary.

Common Definitions	
Instruction set	$\boldsymbol{I}$
instruction	i
Software Characterization	
IDG of each warp	IDG_j for $j : 1..\Omega$
Execution Configuration exe	
Group size	$\|\gamma\|$
Number of concurrent groups	γ
Number of concurrent warps	$\omega = \|\gamma\|.\gamma$
Total number of warps per core	Ω
Hardware Characterization	
Subsystem set	$\boldsymbol{S}$
Scheduler	σ
Issue Limit	IL
Context-independent mapper	$f : (i \in \boldsymbol{I}) \mapsto (s \in \boldsymbol{S}, \lambda, \Lambda)$
Context-dependent mapper	$g : (i \in \boldsymbol{I}, code, exe)$ $\mapsto (s \in \boldsymbol{S}, \lambda, \Lambda)$

3.1.1. Software

The instruction sequence of each hardware thread (warp) is modelled by an Instruction Dependence Graph (IDG), which is a directed acyclic graph in which each vertex of the graph corresponds to a single executed instruction. The edges between the vertices represent the def-use dependencies between the corresponding instructions, which can either be data dependencies or control dependencies. Instructions are elements from a global Instruction set $\boldsymbol{I}$. Instructions belong to one and only one of the following three categories. Memory instructions copy data from one memory level to another (e.g. RAM to registers). Computation instructions apply operations to register data. Barrier instructions synchronize hardware threads that belong to the same group.

In theory, every warp can follow a different execution path making it necessary to create a different *IDG* for every thread. However, in many cases all or most of the threads follow the same execution path, making it possible to use a single *IDG* for the simulation. Different strategies can be followed if there is divergence between the execution of threads.

3.1.2. Execution configuration

The kernel threads for which a software must be executed are organized in **groups**. A group is assigned in its entirety to one of the cores where it remains until it completes execution.

The execution configuration then consists of the following parameters:

Group size $|\gamma|$ The number of kernel threads of each group.

Number of concurrent groups γ The number of groups that can run concurrently on a single core. This is determined by the amount of resources each work group needs.

The occupancy (number of concurrent warps) ω is then the number of concurrent groups multiplied by the group size.

3.1.3. Hardware

A GPU core is characterized by the following elements:

Subsystem set $\boldsymbol{S}$ A set of symbols, one for each pipeline that models a subsystem.

Scheduler σ The scheduler determines which instruction is issued next given the groups currently present on a core and the state of the subsystems. An important parameter of the scheduler is the *issue limit*, which determines the maximum number of instructions that can be scheduled in one cycle.

The timing behavior of the execution of instructions is modeled by the two characteristics of a pipeline: throughput and end-to-end latency. For the first, we will use the inverse of throughput (IPC), which we denote by λ (CPI). The *latency* or Λ is the duration from the issue of an instruction to a subsystem until its result becomes available for dependent instructions.

In general, for computational instructions, the lambdas are constants and independent of the *context* of the instruction's execution. So, the following mapping can be defined for context-independent instructions:

$$f : (i \in \boldsymbol{I}) \mapsto (s \in \boldsymbol{S}, \lambda, \Lambda)$$

It defines a mapping from instruction type to subsystem, λ and Λ.

On the other hand, the timing behavior of instructions might depend on the context in which they execute. Typical examples are memory instructions that may be serviced faster or slower due to the effects of caching, memory access patterns or contention with other threads. All three factors are determined by a combination of the hardware (memory and cache configuration, ...), the software (memory indices, previous accesses, ...) and the execution configuration. To accommodate this phenomenon, the mapper f should be replaced by a mapper which takes the software characterization and execution configuration into account:

$$g : (i \in \boldsymbol{I}, code, exe) \mapsto (s \in \boldsymbol{S}, \lambda, \Lambda)$$

3.2. Model execution by simulation

At first, hardware-dependent information is added to the IDGs in two steps. The parameter mappers f and g are used to determine the appropriate subsystem and lambdas for each instruction. The interaction between software and hardware is then analyzed by simulating the timing behavior of the execution.

A detailed execution profile is generated by simulating the timing behavior of each warp with the IDG and the latencies. Examples are shown in Figs. 8 and 9. The throughput is simulated by using λ (CPI) as 'issue latency': the time between two consecutive issues. Ready instructions (for which the dependencies are satisfied) are issued when their subsystem is idle (i.e. λ of the previously issued instruction has transpired). The dependencies of an instruction are satisfied if the dependent instructions have terminated their completion latency. When multiple instructions are ready to be issued, the scheduler's policy decides which instruction is issued first.

The simulation returns the runtime expressed in processor cycles of a kernel on a single core. We refer the reader to Lemeire et al. (2016) for the detailed equations to compute global performance.

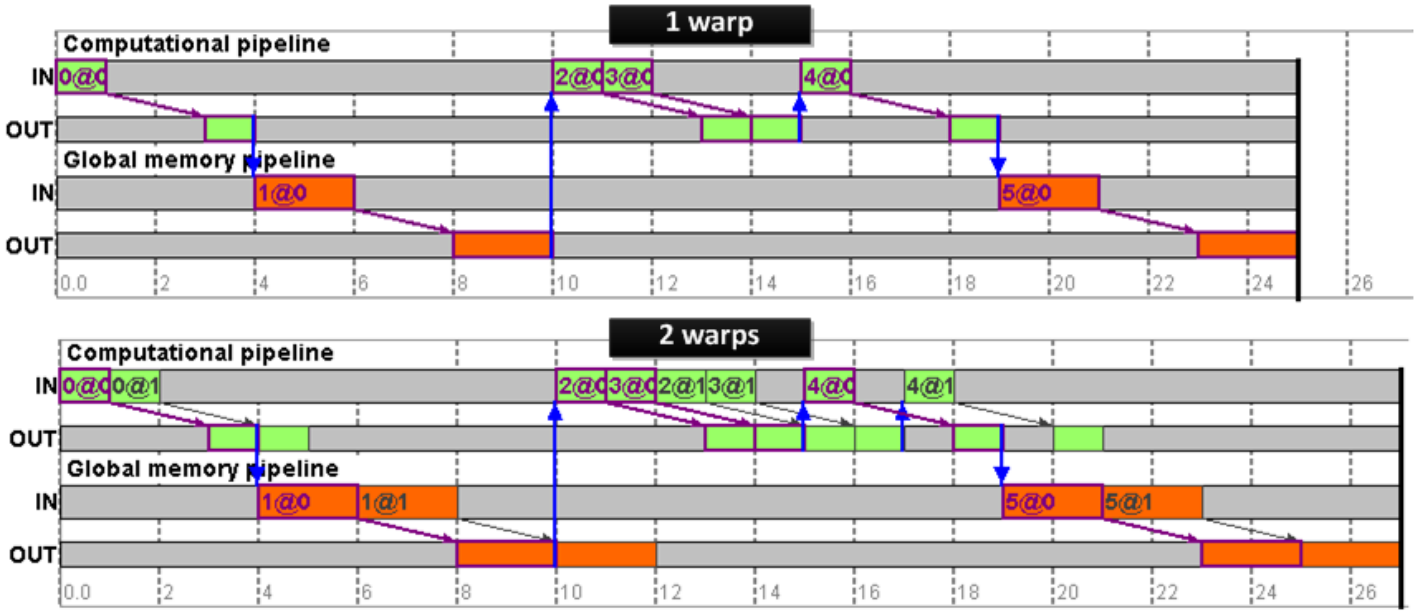

Figure 8: Execution profile of the example kernel and the example parameters, for an occupancy of 1 warp (top) and 2 warps (bottom). The computational and memory pipeline are shown. Each instruction 'occupies' the pipeline when entering it according to its throughput CPI. The latency determines when the instruction leaves the pipeline. Blue arrows indicate instructions that have to wait for others to finish because of a dependency.

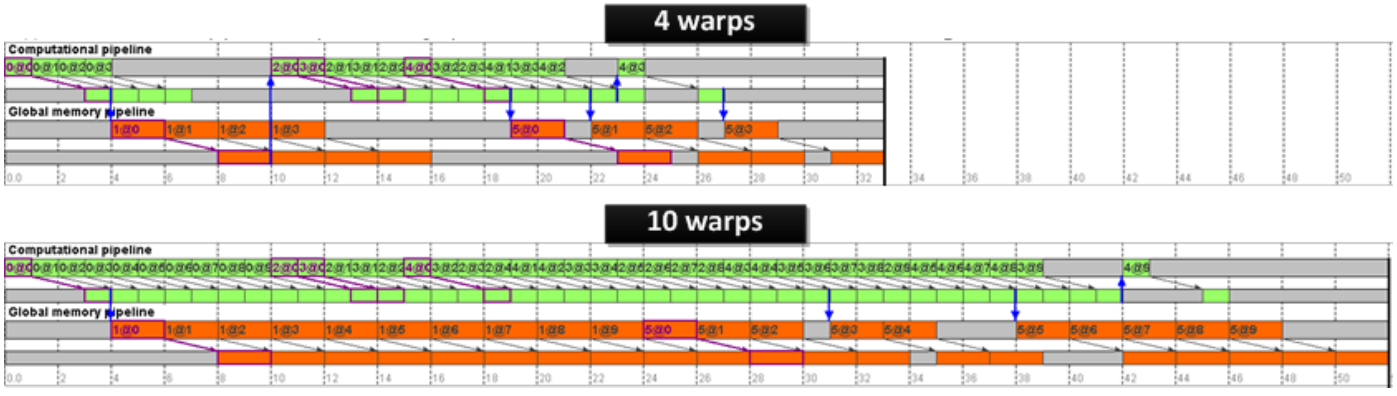

Figure 9: Execution profile of the example kernel and the example parameters, for an occupancy of 4 warp (top) and 10 warps (bottom). With 10 warps, the execution is almost at the peak performance (no idle periods except for the pipeline fill and drain).

4. Discussion

The above description formally defines the abstractions that together make up the Pipeline model. However, a number of questions that concern the implementation of this model were left unanswered. This section will discuss the most salient of them, and where appropriate it will mention how they are handled by existing models.

4.1. From WFG to IDG

The WFG of Baghsorkhi et al. (2010) defines the instruction execution order and the def-use dependencies between memory transactions and the computational instructions depending on the data. The dependencies among computational instructions in a basic block are expressed by the ILP factor, but the WFG does not describe how memory instructions depend on the outcome of computational instructions (the memory address or data). This gives rise to problems with the equations as discussed in Sec. 2.4.4. To overcome this problem, the IDG describes all def-use dependencies in the graph.

There are three differences between the WFG and the IDG of the Pipeline model. Fig. 1 shows the IDG of the example kernel. This is mainly motivated by allowing maximal detail without adding additional parameters. The first difference is that the IDG does not group computational instructions into a single node but maps each instruction onto a node. The second difference is that the IDG only contains edges for all def-use dependencies, not for the execution order. Not fixing the execution order when describing the GPU kernel is motivated by a possible reordering by either:

- The compiler, which means that the order is hardware dependent and might not be known when describing the kernel from the source code; or
- The GPU's instruction scheduler, which may issue out-of-order.

The scheduler σ of the model decides on the execution order.

The third difference is that the Pipeline model allows every warp to have a different IDG in the case of warp divergence. Meanwhile, the WFG model estimates the runtime based on a 'typical' warp by taking weighted averages in case of branch divergence.

4.2. Subsystems and Issue Limits

The Pipeline model considers each subsystem as a separate and independent pipeline. The timing behavior of an instruction is characterized by a throughput and a latency. All analytical models consider at least two subsystems: a computational and a memory subsystem. However, the architecture of modern GPUs is more complex than a simple pipeline. It contains several warp schedulers and dispatch units. A multiprocessor of the Nvidia Pascal architecture can issue four instructions per cycle on 128 CUDA cores, which we model with a λ of 0.25 cycles. The WFG model defines the CPI in the same way as $SIMD_{\text{work}}/SIMD_{\text{engine}}$ with $SIMD_{\text{work}}$ the warp size (number of kernel threads that are executed together in lock-step) and $SIMD_{\text{engine}}$ the number of scalar processors on a multiprocessor. Many GPUs have separate SFU units which can be modelled with a separate SFU subsystem besides the ALU subsystem for integer and floating point calculations. There are typically less SFU units, resulting in a lower throughput. Even in the presence of independent subsystems, the total number of instructions that can be issued per cycle is limited by the number of warp schedulers and dispatch units. This is modelled with an overall *issue limit* (expressed as IPC), which the scheduler takes into account. Also the Volkov model takes this limit into account. Given the number of executed instructions per warp, it is translated to a warp throughput that imposes an additional bound (Volkov, 2016). The issue limit can be regarded as a common *valve* as shown in Fig. 10.

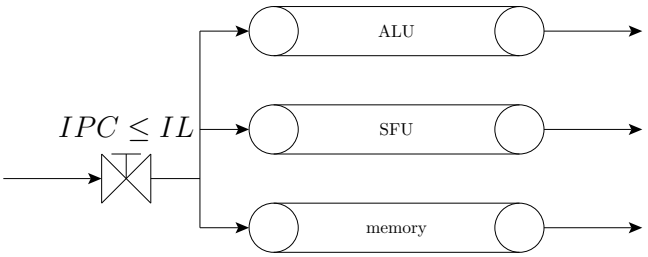

Figure 10: The Pipeline model: subsystems are modelled as pipelines and the issue limit as a valve.

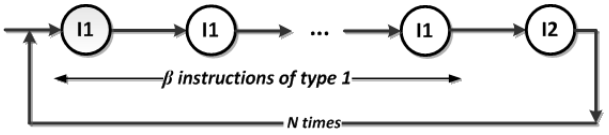

Figure 11: Instruction Dependency Graph (IDG) of the Instruction Mix kernel.

The subsystems and issue limits can be revealed empirically with a parameterized *Instruction Mix benchmark kernel* (Volkov, 2016). The kernel is a repetition of a sequence of β instructions of type 1 followed by one instruction of type 2 (where $\lambda_1 < \lambda_2$), as shown in Figure 11. The number of repetitions N is chosen large enough such that the kernel overhead can be neglected. The performance is then measured as a function of β at maximal occupancy. There are three possible situations:

1. The instructions are executed on the same subsystem (modelled by one pipeline). The performance (throughput of the instructions of type 1) is then:

$$IPC_{\text{instr1}}(1\ subsystem) = \frac{\beta}{\beta.\lambda_1 + \lambda_2} \tag{43}$$

2. The instructions are executed on different subsystems (modelled by two independent pipelines) without issue limit. The performance is then:

$$IPC_{\text{instr1}}(2\ subsystems) = \min(\frac{1}{\lambda_1}, \frac{1}{\lambda_2}) \tag{44}$$

3. The instructions are executed on different subsystems but there is an overall issue limit of the number of instructions

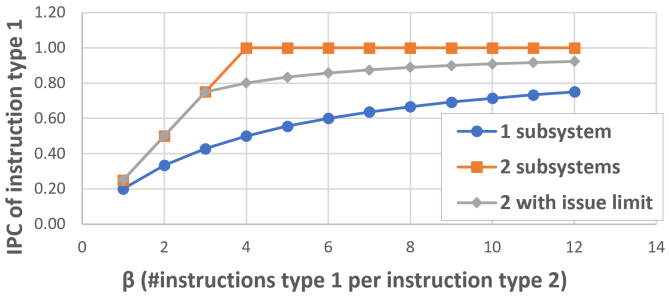

Figure 12: The theoretical performance of the Instruction Mix benchmark for different architectures with $\lambda_1 = 1$, $\lambda_2 = 4$ and $IL = 1$.

Table 4: The GPUs with their instruction parameters and Issue Limit (IL).

GPU	architecture	λ_{ALU}	λ_{SFU}	IL
NVIDIA Tesla C2050	Fermi	1	8	1
NVIDIA GeForce GTX 650 Ti	Kepler	0.25	1	4
NVIDIA Quadro K620	Maxwell	0.25	1	4
NVIDIA GeForce GTX 1060	Pascal	0.25	1	4
NVIDIA GeForce RTX 2070	Turing	0.5	2	2
AMD Radeon R9 380	Tonga	1	5	1

that can be issued within a cycle. The performance is then:

$$IPC_{\text{instr1}}(2 \textit{ with issue limit } IL) = \min(\frac{1}{\lambda_1}, \frac{1}{\lambda_2}, \frac{\beta}{\beta+1}.IL) \quad (45)$$

The three curves for some example parameter values are shown in Fig. 12.

To identify the best-fitting model of a GPU we performed experiments with a mix of β floating point instructions (ALU) with one special function instruction (SFU). Four different architectures were validated. Their latencies were measured with microbenchmarks (Lemeire et al., 2016) and are shown in Table 4. Fig. 13 shows the experimental results with the three theoretical curves. The results reveal that the Fermi and Kepler architectures can be modelled with a separate ALU and SFU subsystem, while considering an overall issue limit. The Tonga architecture is best modeled as a single subsystem for ALU and SFU instructions. For the Pascal and Turing it is however less clear. The differences of the real performance with the theoretical curves come from the higher complexity of the real architectures. Each multiprocessor contains multiple instruction buffers, warp schedulers and dispatch units, for which the generic and simple pipeline analogy can only provide an approximation. Kepler converges to an IPC of 4.3, which is higher than the expected maximum of 4. The Kepler architecture with 192 CUDA cores can issue six warp instructions per cycle, but this can only happen with ILP because there are only four warp schedulers, which is not present in the instruction mix kernel.

GPUPerf (Sim et al., 2012), an extension of the MWP-CWP model, also considers separate subsystems. It considers the ALU and SFU as two separate subsystems without issue limit and takes the maximal execution time of both (Eq. 44).

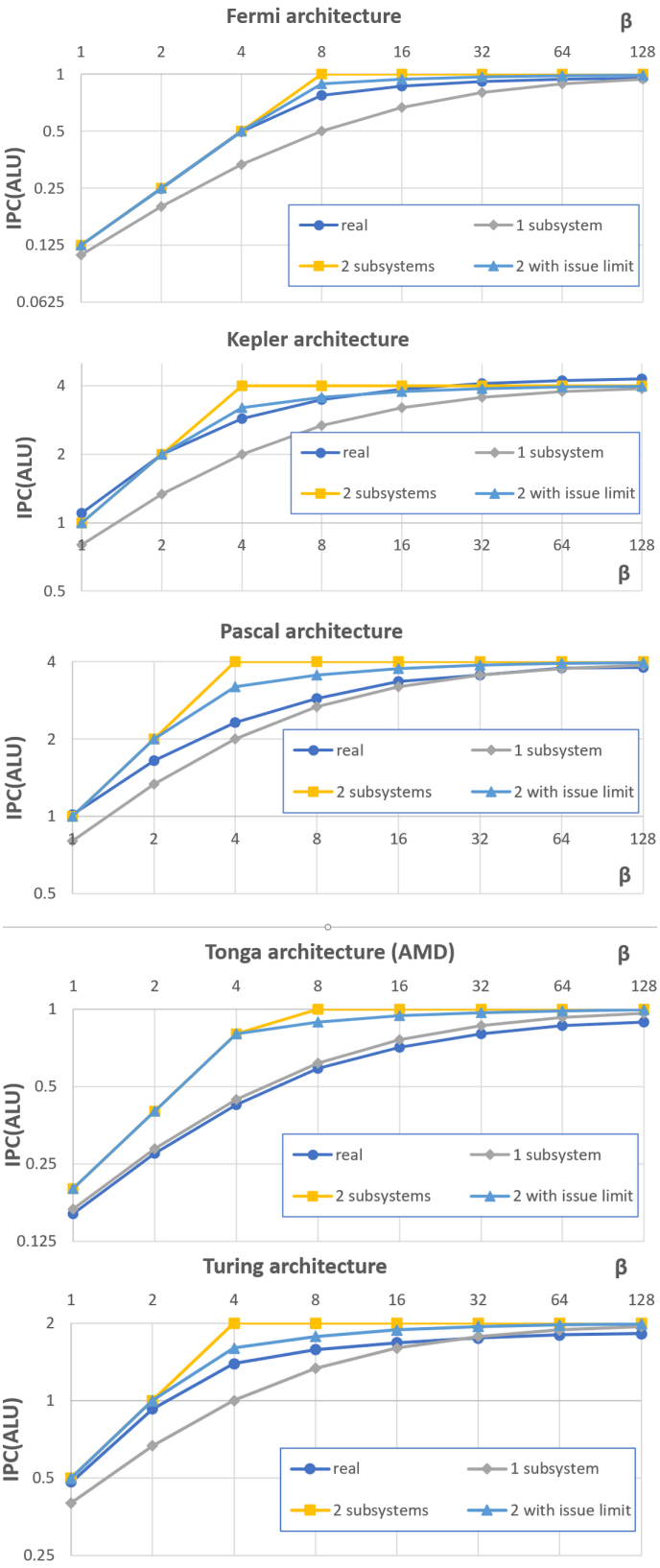

Figure 13: Experimental results ('real') of the Instruction Mix kernel and the 3 models with the GPU parameters of Table 4.

4.3. Instruction parameters λ and Λ

Because the basic scheduling unit is the hardware thread or warp, the CPI of an instruction depends on the ratio of the warp size and the number of processing elements to execute the given instruction. This ratio can be more or less than one, illustrating the abstract nature of the parameter. The throughput can often be derived from the device specifications but in general we have to rely on **microbenchmarks**, especially for the latencies (Lemeire et al., 2016)[1]. A microbenchmark estimates the value of a single characteristic of the hardware. Some instructions can be measured in isolation, while others (like a branch instruction) have to be mixed with other instructions. Next the latencies obtained from the experiments are rounded to the closest multiple of $\frac{1}{N}$ where N is the issue limit.

The lambdas for multiplication `mul` and division `div` instructions for different types are shown in Table 5: single precision and double precision floating point `f32` and `f64` and signed integers `s32`. This table also shows the lambdas for the hardware accelerated cosine `cos.approx.f32`, barrier instructions `bar.sync` and global memory and local memory access for four byte values: `ld.global.s32` and `ld.local.s32`. The lambdas are expressed in clock cycles. The λ of `mul.f32` and `cos.approx.f32` correspond to what is expected given the core ALU count N_{ALU} and the core SFU count N_{SFU}. This is not the case for the Maxwell GPU. The λ for simple floating point instructions on this GPU was measured to be 0.375 cycles rather than 0.25 cycles as would be expected given the architectural details (there are four warp schedulers per compute unit and 4×32 ALU processing elements). The measured value corresponds to an IPC of $\frac{8}{3}$ rather than the expected $\frac{8}{2}$. To obtain the latter IPC, it is necessary to introduce instruction level parallelism in the code. Until the date of this writing no explanation was found for this phenomenon.

A few additional observations are worth making. First, for all NVIDIA GPUs $\frac{\lambda_{div.f32}}{\lambda_{mul.f32}}$ is 3. For the AMD GPU it is 2.25. Secondly, $\frac{\lambda_{mul.f64}}{\lambda_{mul.f32}}$ is 2 for the Fermi GPU – one especially designed for computing –, while it is 16, 20, 32 and 38 for the Kepler, Maxwell, Pascal and Turing GPUs respectively. For the Tonga GPU it is 8. Furthermore, barrier synchronization has become relatively costlier compared with regular instructions on more recent GPUs. Finally, the global memory read CPI allow us to compute the obtained throughput:

$$throughput(GB/s) = N(\Pi) \times \frac{|\omega| \times 4}{\lambda_{RAM}} \times f_{clock}(GHz)$$

Table 6 compares the peak throughput with the theoretical bandwidth of the device. The theoretical bandwidth for the Fermi GPU assumes that error control and correction or ECC is disabled. ECC adds overhead and decreases the achieved throughput.

[1] In our previous work, we called λ the issue latency and Λ the completion latency, but these terms cause too much confusion.

4.4. Memory Analysis Modules for mapper g

Ultimately, we would like to have a model with a separate characterization of software and hardware such that combinations of hardware and software can easily be studied. The Pipeline model achieves this for context-independent instructions (when mapper f is used). Unfortunately, instructions that do depend on context break the separation of concerns. On GPUs, this happens mostly for memory instructions. Hardware, software and configuration should be considered together to determine the correct lambdas. Examples are memory bank conflicts, which depend on memory organization and the memory access pattern, and cache misses which depend on cache organization and memory access. This is represented in the Pipeline model by mapper g.

For memory instructions we can implement g as a *Memory Analysis Module* which estimates the lambdas and feeds them to the Pipeline model. For these lambdas, one can either consider average values based on aggregate numbers obtained from a profiler, or one can take the execution sequence and instruction dependencies into account. An example of the former is GPUPerf (Sim et al., 2012) (an extension of the MWP-CWP model), which uses the Average Memory Access Time (AMAT), while the latter is analyzed in detail in GPUMech and the related MDM (Wang et al., 2019).

5. Comparison with the analytical models

From the analysis of Sec. 2 we can deduce that the Pipeline model captures all aspects of the analytical models except for the difference with the MWP-CWP model discussed in Sec. 2.3.4 and shown in Fig. 4. But the Pipeline model is more flexible. For each model, Table 7 shows which performance aspects it takes into account. The Pipeline model combines all of them.

5.1. Comparison of performance

To compare the application of the models on the example kernel, we correct and extend the equations such that the same aspects are taken into account (see Table 7). The corrected models are indicated with an asterisk.

MWP-CWP* is the corrected version which takes the latencies for computational instructions into account by replacing Λ^*_{app} in Eq. 28 with Λ_{app} defined by Eq. 2. Now it is however less clear for which values of ω we are in the occupancy-bound case. To determine this, we take the maximal CPR value of the three cases. For our corrected version of the WFG model, the memory instruction CPI $\lambda^{\text{instr}}_{\text{mem}}$ is set to 0 and Eq. 42 is used instead of Eq. 36. GPMech(RR)* is the corrected version proposed in Sec. 2.5.1.

This results in the performance curves shown in Fig. 14. The figure clearly shows that Volkov's model provides an upper bound for occupancy-bound kernels[2]. It assumes maximal latency hiding. At the left-hand side of the ridge point, the first

[2] GPUMech ignores the computational latencies, hence the performance is a bit above Volkov's curve.

Table 5: The parameters CPI (λ) and latency (Λ) for selected instructions expressed in number of clock cycles.

	Fermi	Kepler	Maxwell	Pascal	Turing	Tonga
Instruction	λ / Λ	λ / Λ	λ / Λ	λ / Λ	λ / Λ	λ / Λ
cos.apx.f32	8 / 40	1 / 18	1 / 15	1 / 15	2/21	5 / 24
mul.f32	1 / 18	0.25 / 9	0.375 / 6	0.25 / 6	0.5/4	1 / 5.25
mul.f64	2 / 22	4 / 22	7.5 / 42	8 / 43	19/45	8 / 76
mul.s32	2 / 18	0.5 / 5	0.875 / 12.5	0.75 / 12	0.25 / 2	1 / 5.25
div.f32	3 / 45	0.75 / 28.5	1.125 / 20	0.75 / 18	1.5 / 12.5	2.25 / 14
div.f64	19 / 253	26 / 260	47 / 376	47 / 376	- / -	155 / 740
div.s32	20 / 200	3 / 96	7 / 105	5 / 100	5/65	24 / 192
bar.sync	2 / 40	0.75 / 24	4.5 / 125	2.25 / 70	1.5/17	7.5 / 150
ld.global.s32	23 / 475	7.5 / 300	18 / 440	12 / 345	18/450	42 / 136
ld.local.s32	2 / 28	1 / 28	1 / 28	1 / 25	2/32	2 / 60

Table 6: Comparison of achieved throughput and theoretical bandwidth.

Device	Throughput (GB/s)	Bandwidth (GB/s)	Efficiency
Fermi	89.6	144	62.2%
Kepler	70.5	86.4	81.5%
Maxwell	22.6	29	77.8%
Pascal	160.6	192	83.7%
Turing	288	361	80.5%
Tonga	172	176	97.7%

Table 7: The performance aspects taken into account by the models.

	Roofline	Volkov	MWP-CWP	WFG	GPUMech	Pipeline
Aspects considered in comparison of Sec. 5.1						
ILP	Y	Y	N	Y	N	Y
Λ_{comp}	N	Y	N	Y	N	Y
Λ_{mem}	N	Y	Y	Y	Y	Y
Other aspects						
barriers	N	N	Y	Y	N	Y
more than 2 subsystems	Y	Y	N	N	N	Y
workgroups	N	N	N	Y	N	Y
divergent warps	N	N	N	Y	Y	Y
issue limit	N	Y	Y	N	N	Y
scheduler	N	N	N	N	Y	Y

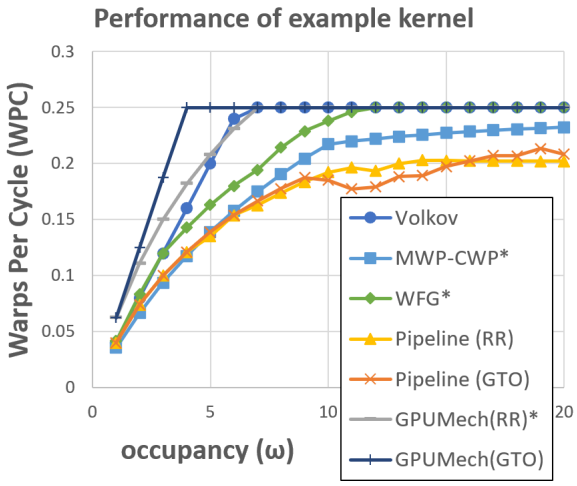

Figure 14: The performance estimations of the different models after corrections for taking the same aspect into account.

warp hides all the other warps. At the right-hand side of the ridge point, all latencies are hidden.

The WFG model expresses that successive warps cannot be hidden completely behind the first warp. The MWP-CWP model adds to Volkov's roofline the idling during the pipeline fill and drain. However, the impact of this gets negligible for complex kernels. MWP-CWP and Volkov assume that at all other moments there is a ready instruction present that can be scheduled (i.e. it assumes that an *ideal scheduling order* exists and that an omniscient scheduler uses the optimal scheduling).

In practice, schedulers use scheduling policies based on heuristics. The Pipeline model uses a real scheduler to generate the execution profile. It reveals the idling due to non-ideal scheduling; as a result, the occupancy-performance curve is further flattened.

5.2. Model Inputs

The Pipeline model is based on a combination of the parameters of the analytical models without introducing new ones. For their input some models rely on aggregate instruction counts (Roofline, Volkov, MWP-CWP), while others rely on a detailed dependency graph (WFG, GPUMech, Pipeline model). Except Roofline, all of the models consider throughput (or CPI, the inverse of throughput) and at least the latencies for memory instructions. For most computational instructions, the lambdas only depend on the GPU and can be estimated with microbenchmarks. Meanwhile, for memory transactions the lambdas also depend on the access pattern, cache utilization, spatial locality and resource contention. For this, each model employs a separate memory analysis as explained in Sec. 4.4. For our experiments, a memory analysis module is introduced in Sec. 6.3.

5.3. Understanding performance

Besides accurate prediction of runtime, a performance analysis should also yield insight into the aspects limiting the performance. This is more easy to achieve by the proposed abstract simulation than by the analytical equations. The generated execution profile (Figs. 8 and 9) can, for instance, be used to calculate the filling rate of each pipeline or the amount overlap (e.g. as defined by GPUMech). A detailed analysis can further reveal the causes of non-overlap.

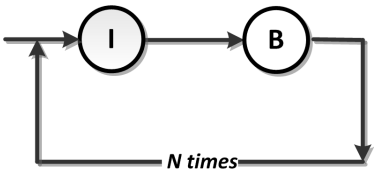

Figure 15: Instruction Dependency Graph of the Iterative Barrier Kernel. Node I stands for a computational instruction and node B for a barrier instruction.

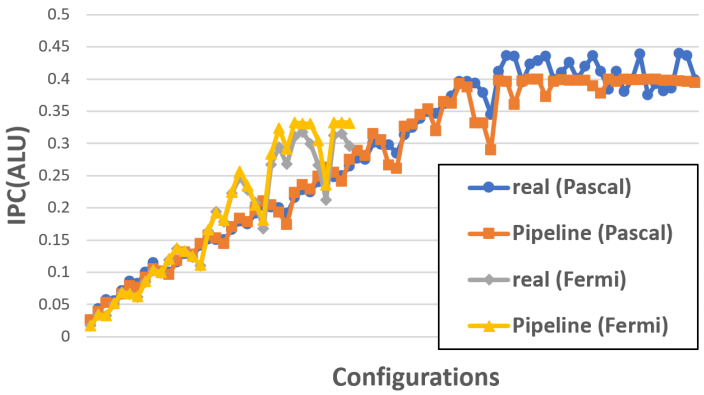

Figure 16: Performance of the iterative barrier kernel with different occupancies and work group sizes for the Fermi and Pascal GPU. Occupancy varies between 1 and the maximum (resp. 48 and 64), work group size varies between 32 and 1024. The results are first sorted on occupancy and then on work group size.

6. Empirical Validation

We validate the model on the NVIDIA GPUs from Table 4. The lambdas for the instructions that were used to carry out the simulations are shown in Table 5 (Lemeire et al., 2016).

6.1. Instruction Mix Kernel

First, we consider the instruction mix kernel from Section 4.2 with $\beta = 4$ (around the roofline ridge point) and varying occupancy. The mix is run 256 times in an unrolled loop to minimize the kernel overhead.

6.2. Iterative Barrier Kernel

Next, we consider a kernel that consists of a sequence of N dependent instructions each separated by a barrier instruction, as shown in Figure 15. N is chosen large enough such that the kernel overhead can be neglected. This kernel allows us to study the impact of the group size on the kernel's performance and the pipeline model's ability to take this into account. As can be seen in Figure 16 the performance is indeed highly influenced by the group size. This is also reflected by the results of the Pipeline model. For more recent GPUs there are additional factors that influence the performance that are not taken into account. Table 8 shows the performance estimation accuracy of the Pipeline model for both experiments. The Pipeline model is quite accurate, although it is less accurate for the more recent Pascal GPU.

Table 8: Pipeline model accuracy: Mean Absolute Percentage Error (MAPE) of the performance for the different experiments.

	Fermi	Kepler	Pascal
Instruction mix	6.7%	7.1%	20.5%
Barrier kernel	3.8%	3.9%	5.1%

Table 9: Kernels used for evaluation and occupancy range of experiments. The minimum and maximum occupancy is expressed in number of warps. The minimum occupancy corresponds to the size of a single work group. It is the minimum occupancy on the Quadro K620 because on the GeForce GTX 1060 6GB it is not possible to limit the number of concurrent work groups to one with local memory.

Kernel	**Benchmark Application**	**Occupancy Range**
Fan2	Gaussian Elimination	2 – 64
nw_kernel1	Needleman-Wunsch	1 – 32
kmeans_kernel_c	Kmeans	8 – 64
kmeans_swap	Kmeans	2 – 64
lud_perimeter	LUD Decomposition	1 – 32
lud_internal	LUD Decomposition	8 – 64
hotspot	Hotspot3D	8 – 64
opt_srad1	SRAD	8 – 64
opt_srad2	SRAD	8 – 64
bpnn_layerforward_ocl	Back Propagation	8 – 64
bpnn_adjust_weights_ocl	Back Propagation	8 – 64
compute_step_factor	CFD Solver	6 – 60
time_step	CFD Solver	6 – 60
compute_flux	CFD Solver	6 – 60

6.3. Rodinia Benchmark Suite

Finally, we evaluated the Pipeline model and existing performance models on 14 kernels from the well-known Rodinia benchmark suite (Che et al., 2009), which was also used by the authors of most analytical performance models. The choice of kernels was based on practical considerations such as the existence of an OpenCL version, a sufficiently large problem size and the relative independence of the execution path on the processed data. We executed each kernel under investigation on two different GPUs for a range of occupancies determined by the kernel's group size. For this, the code was adapted to run with different numbers of concurrent work groups by allocating local memory. Table 9 gives an overview of the kernels together with the benchmark application to which they belong. The same lambdas were used for all models (Table 5).

Memory lambdas

To determine the loop-trip counts and the memory access latencies we used statistics obtained from the NVIDIA profiler tool *nvprof*. The memory lambdas were determined as follows:

- For global memory access we compared the number of bytes accessed in DRAM $bytes_{DRAM}$ to the number of bytes accessed in global memory $bytes_{global}$. In the following we represent the ratio of both as $R = \frac{bytes_{DRAM}}{bytes_{global}}$. Furthermore, the DRAM and L2 lambdas are denoted by $\Lambda_{DRAM}, \lambda_{DRAM}$ and $\Lambda_{L2}, \lambda_{L2}$ respectively.
  - $R < 1$. We assume the fraction $1 - R$ of the bytes is accessed from L2 cache: With a issue probability of

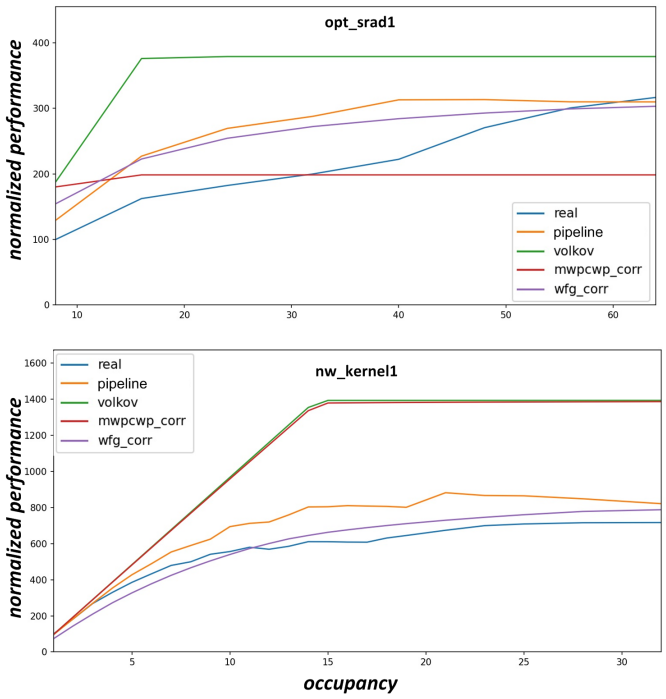

Figure 17: Experimental and model performance of kernels opt_srad1 and nw_kernel1 on the Quadro K620 (Maxwell). The performance (throughput) is normalized with respect to the real performance at minimal occupancy which is set to 100.

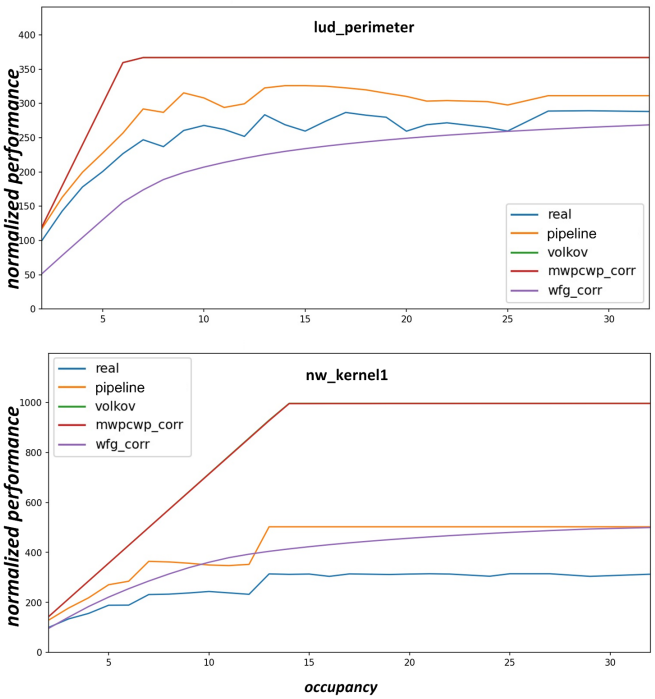

Figure 18: Experimental results of kernels lud_perimeter and nw_kernel1 on the GeForce GTX 1060 (Pascal). Note that for both kernels the curve for Volkov's model is not visible since it lies exactly on the mwpcwp_corr curve. The performance (throughput) is normalized with respect to the real performance at minimal occupancy which is set to 100.

1, *all* instructions of the remaining warps are used for overlapping with the stall cycles until all stall cycles are overlapped.

$$\begin{aligned} \Lambda_{global} &= R \times \Lambda_{DRAM} + (1-R) \times \Lambda_{L2} \\ \lambda_{global} &= R \times \lambda_{DRAM} + (1-R) \times \lambda_{L2} \end{aligned}$$

- $R = 1$.

$$\begin{aligned} \Lambda_{global} &= \Lambda_{DRAM} \\ \lambda_{global} &= \lambda_{DRAM} \end{aligned}$$

- $R > 1$. More bytes are accessed in DRAM than strictly necessary. We compute the lambdas as follows:

$$\begin{aligned} \Lambda_{global} &= \Lambda_{DRAM} + (R-1) \times \lambda_{DRAM} \\ \lambda_{global} &= R \times \lambda_{DRAM} \end{aligned}$$

- For local memory access we use the bank conflict degree D to estimate the lambdas as follows:

$$\begin{aligned} \Lambda'_{local} &= \Lambda_{local} + D \times \lambda_{local} \\ \lambda'_{local} &= (1+D) \times \lambda_{local} \end{aligned}$$

Results Some experimental results are shown in Figures 17 and 18. Volkov's model clearly gives an upper bound to the performance but is too optimistic. WFG_corr and Pipeline are closer to the true values. But especially for the Pascal GPU, the Pipeline model gives a shape very similar to the true occupancy curve, although there is a clear offset between both curves. This reflects that quantitatively Pipeline is not completely accurate, but that the model correctly captures the qualitative behavior of the GPUs. To quantify shape similarity, we derive a new metric from the MAPE metric with which the predictive accuracies are assessed.

The MAPE (Mean Absolute Percentual Error) of the performance is used to assess predictive accuracy:

$$MAPE^{model} = \frac{\sum_{occ} |WPC^{model}_{occ} - WPC^{real}_{occ}| / WPC^{real}_{occ}}{n} \times 100$$

where occ traverses all occupancy values, and with n the number of values. The result tables also show the average MAPE per model for all kernels.

The new metric $MAPE^{shape}$ measures how well the *shape* of the occupancy curve is modelled. It is calculated as follows:

$$MAPE^{model}_{shape} = \frac{\sum_{occ} |WPC^{model}_{occ} - WPC^{real}_{occ} - lf(occ)|}{WPC^{real}_{occ} \times n} \times 100$$

with $lf(occ)$ the best linear fit of all values $WPC^{model}_{occ} - WPC^{real}_{occ}$. The new metric measures the MAPE of the differences to the linear fit. This procedure reduces the original MAPE value: a constant *offset* and a constant drift of this offset is subtracted from the differences.

The results of the evaluation are shown in Tables 10 to 12 for the Quadro K620, the GeForce GTX 1060 and GeForce RTX 2070 respectively.

The Pipeline model obtains the best results, with Volkov's model a close second. The other models perform worse, unless we apply the proposed corrections for WFG and MWP_CWP. Then the accuracies of these models are between Pipeline and Volkov. $MAPE^{shape}$ gives values below 10% (and even better for the Pascal and Turing) which indicates that the errors are largely due to incorrect lambdas rather than an incorrect modeling of GPU behavior.

The total simulation time of the Pipeline model for all the results (226 experiments for each of the 3 devices) was about 3 minutes.

7. Conclusion

The Pipeline model is shown to be the underlying model of the analytical GPU performance models. It allows the interpretation and validation of their assumptions and their often complex analytical equations, based on simple concepts (i.e. pipelines, throughputs and latencies). Simulation is used to estimate the performance instead of equations. It provides a mental model of how threads are executed on GPUs and it offers the modeling community a clear description of the underlying concepts of current models. Moreover, the Pipeline model offers the flexibility to take more aspects into account, including scheduler, barrier synchronization, concurrent work groups and multiple subsystems. Empirical validation shows that the Pipeline model gives a reasonable accuracy on real kernels (average MAPE of 24%). On the other hand, the accuracy of the simplest model taking occupancy into account, Volkov's model, has an average MAPE of 52.9% which was hard to beat by the more complex analytical models.

We conclude that Volkov's model can be used for a quick estimation of an upper bound of the performance, while simulation of the Pipeline model is needed for an improved accuracy and for a quantitative or qualitative analysis of the various performance aspects.

Our website `www.gpuperformance.org` offers a Java app for microbenchmarking GPUs. The GPU performance characteristics can then be added to and compared with the GPUs of our database. An implementation of the simulation of the Pipeline model is also available.

References

Baghsorkhi, S.S., Delahaye, M., Patel, S.J., Gropp, W.D., Hwu, W.W., 2010. An adaptive performance modeling tool for GPU architectures, in: 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP, pp. 105–114.

Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S.H., Skadron, K., 2009. Rodinia: A benchmark suite for heterogeneous computing, in: 2009 IEEE International Symposium on Workload Characterization (IISWC), pp. 44–54. doi:10.1109/IISWC.2009.5306797.

Eyerman, S., Eeckhout, L., Karkhanis, T., Smith, J.E., 2009. A mechanistic performance model for superscalar out-of-order processors. ACM Trans. Comput. Syst. 27, 3:1–3:37.

Hong, S., Kim, H., 2009. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness, in: 36th International Symposium on Computer Architecture, ISCA.

Huang, J., Lee, J.H., Kim, H., Lee, H.S., 2014. GPUMech: GPU performance modeling technique based on interval analysis, in: 47th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO, pp. 268–279.

Konstantinidis, E., Cotronis, Y., 2015. A practical performance model for compute and memory bound GPU kernels, in: 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP, pp. 651–658.

Konstantinidis, E., Cotronis, Y., 2017. A quantitative roofline model for GPU kernel performance estimation using micro-benchmarks and hardware metric profiling. J. Parallel Distrib. Comput. 107, 37–56.

Lemeire, J., Cornelis, J.G., Segers, L., 2016. Microbenchmarks for GPU characteristics: The occupancy roofline and the pipeline model, in: Conference on Parallel, Distributed, and Network-Based Processing, PDP, pp. 456–463.

Li, A., Song, S.L., Brugel, E., Kumar, A., Chavarría-Miranda, D.G., Corporaal, H., 2016. X: A comprehensive analytic model for parallel machines, in: 2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016, pp. 242–252.

Li, A., Tay, Y.C., Kumar, A., Corporaal, H., 2015. Transit: A visual analytical model for multithreaded machines, in: 24th International Symposium on High-Performance Parallel and Distributed Computing, HPDC, pp. 101–106.

Lopez-Novoa, U., Mendiburu, A., Miguel-Alonso, J., 2015. A survey of performance modeling and simulation techniques for accelerator-based computing. IEEE Trans. Parallel Distrib. Syst. 26, 272–281.

Madougou, S., Varbanescu, A.L., de Laat, C., van Nieuwpoort, R., 2016. The landscape of GPGPU performance modeling tools. Parallel Computing 56, 18–33.

Nugteren, C., Corporaal, H., 2012. The boat hull model: enabling performance prediction for parallel computing prior to code development, in: Proceedings of the 9th conference on Computing Frontiers.

Sim, J., Dasgupta, A., Kim, H., Vuduc, R.W., 2012. A performance analysis framework for identifying potential benefits in GPGPU applications, in: 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP, pp. 11–22.

Volkov, V., 2016. Understanding Latency Hiding on GPUs. Ph.D. thesis. University of California at Berkeley.

Volkov, V., 2018. A microbenchmark to study GPU performance models, in: 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP, pp. 421–422.

Wang, L., Jahre, M., Adileh, A., Wang, Z., Eeckhout, L., 2019. Modeling emerging memory-divergent gpu applications. IEEE Computer Architecture Letters 18, 95–98.

Williams, S., Waterman, A., Patterson, D.A., 2009. Roofline: an insightful visual performance model for multicore architectures. Commun. ACM 52, 65–76.

kernel	Pipeline	Volkov	MWP-CWP	WFG	MWP-CWP_corr	WFG_corr	gpumech_rr	gpumech_gto
Fan2	12 / 4.8	15 / 7.8	22 / 17.0	91 / 32.3	11 / 6.6	22 / 17.3	63 / 30.4	48 / 22.9
nw_kernel1	20 / 10.5	78 / 32.3	98 / 43.4	78 / 20.5	76 / 31.4	11 / 5.5	92 / 18.4	49 / 10.3
kmeans_kernel_c	49 / 8.8	49 / 9.8	59 / 4.2	96 / 9.9	49 / 9.8	57 / 4.9	46 / 9.3	25 / 7.7
kmeans_swap	51 / 15.8	52 / 16.8	51 / 13.4	87 / 16.8	52 / 16.8	52 / 13.1	212 / 16.6	212 / 16.6
lud_perimeter	9 / 4.9	19 / 11.6	18 / 11.0	83 / 25.3	19 / 11.6	32 / 10.3	51 / 23.4	17 / 7.1
lud_internal	36 / 11.1	63 / 22.4	47 / 17.1	92 / 6.4	50 / 17.0	21 / 6.5	54 / 3.4	18 / 3.1
hotspot	18 / 9.4	22 / 13.3	15 / 4.4	95 / 13.3	22 / 12.9	19 / 5.9	40 / 12.9	38 / 11.2
opt_srad1	28 / 15.0	72 / 24.2	40 / 15.3	74 / 5.7	28 / 3.7	26 / 9.3	243 / 19.7	48 / 13.4
opt_srad2	10 / 3.7	33 / 18.8	26 / 15.5	92 / 10.2	28 / 15.8	12 / 9.2	117 / 8.3	49 / 2.9
bpnn_layerforward_ocl	13 / 7.6	15 / 8.9	60 / 15.9	87 / 10.6	3 / 1.7	25 / 8.2	136 / 6.3	36 / 7.2
bpnn_adjust_weights_ocl	38 / 9.4	63 / 13.2	70 / 8.1	98 / 13.2	63 / 13.1	72 / 8.8	48 / 13.2	48 / 13.1
compute_step_factor	7 / 7.6	2 / 1.1	17 / 13.4	94 / 1.0	4 / 1.8	18 / 11.6	86 / 1.1	80 / 2.5
time_step	2 / 1.3	2 / 1.3	14 / 12.3	91 / 1.3	2 / 1.2	13 / 8.9	208 / 1.1	202 / 1.5
compute_flux	12 / 3.6	14 / 5.7	25 / 4.9	82 / 9.6	14 / 5.8	29 / 2.4	58 / 6.9	33 / 3.8
average	**22 / 8.1**	**36 / 13.4**	**40 / 13.9**	**88 / 12.6**	**30 / 10.7**	**29 / 8.7**	**104 / 12.2**	**65 / 8.8**

Table 10: *MAPE* and $MAPE^{shape}$ values (percentages) of the Rodinia kernels for the Quadro K620 (Maxwell)

kernel	Pipeline	Volkov	MWP-CWP	WFG	MWP-CWP_corr	WFG_corr	gpumech_rr	gpumech_gto
Fan2	28 / 4.9	30 / 6.7	27 / 10.9	92 / 19.9	24 / 5.2	21 / 10.1	75 / 18.4	59 / 14.1
nw_kernel1	54 / 10.6	184 / 46.6	243 / 51.3	70 / 13.3	184 / 46.4	40 / 7.9	200 / 9.7	102/ 13.7
kmeans_kernel_c	16 / 4.7	16 / 4.7	24 / 8.4	94 / 4.8	16 / 4.7	22 / 6.1	108 / 4.7	59 / 15.4
kmeans_swap	38 / 14.8	38 / 14.9	38 / 14.7	84 / 14.9	38 / 14.9	39 / 14.4	391 / 14.640	391/ 14.2
lud_perimeter	15 / 4.7	37 / 9.7	35 / 10.3	78 / 11.4	37 / 9.7	18 / 4.5	96 / 9.2	25 / 5.1
lud_internal	23 / 2.9	45 / 5.3	44 / 4.5	93 / 5.9	43 / 5.1	32 / 0.9	35 / 4.0	37 / 3.7
hotspot	20 / 5.7	20 / 5.9	27 / 2.4	96 / 6.0	20 / 5.9	32 / 1.4	27 / 5.7	23 / 4.6
opt_srad1	33 / 7.1	67 / 1.9	20 / 1.9	78 / 2.0	20 / 1.9	21 / 4.1	234 / 10.8	28 / 5.5
opt_srad2	28 / 2.6	54 / 10.9	43 / 9.1	93 / 5.5	47 / 8.9	13/ 4.9	147 / 3.7	60 / 0.8
bpnn_layerforward_ocl	3 / 2.7	27 / 12.6	55/ 1.9	88 / 3.3	16 / 7.1	26 / 5.6	138 / 1.6	25 / 2.5
bpnn_adjust_weights_ocl	40 / 6.7	66 / 6.7	68 / 4.5	98 / 6.8	66 / 6.7	70 / 4.0	42 / 6.7	42 / 6.6
compute_step_factor	12 / 1.6	14 / 0.4	16 / 10.2	93 / 0.4	11 / 0.7	17 / 9.5	108 / 0.6	98 / 2.1
time_step	4 / 0.18	5 / 0.18	9 / 7.8	90 / 0.22	5 / 0.15	9 / 6.2	235 / 0.54	228 / 1.8
compute_flux	7 / 3.1	9/ 4.3	12 / 7.5	80 / 4.3	8 / 4.2	15 / 1.4	74 / 2.3	29 / 2.1
average	**23 / 5.2**	**44 / 9.3**	**47 / 10.4**	**88 / 7.1**	**38 / 8.7**	**27 / 5.8**	**136 / 6.6**	**86 / 6.6**

Table 11: *MAPE* and $MAPE^{shape}$ values of the Rodinia kernels (percentages) for the GeForce GTX 1060 (Pascal).

kernel	Pipeline	Volkov	MWP-CWP	WFG	MWP-CWP_corr	WFG_corr	gpumech_rr	gpumech_gto
Fan2	24.5 / 7	47.2 / 12	38.9 / 21	40.8 / 8	88.7 / 23	26.5 / 15	61.5 / 23	57.1 / 20
nw_kernel1	11.6 / 5	339.8 / 18	486.8 / 99	339.1 / 18	47.2 / 9	123.3 / 13	277.9 / 10	273.9 / 12
kmeans_kernel_c	57.2 / 8	37.0 / 10	57.5 / 8	37.1 / 10	95.3 / 10	52.2 / 7	42.9 / 10	42.9 / 10
kmeans_swap	44.7 / 9	34.2 / 10	37.5 / 6	34.2 / 10	83.6 / 10	37.8 / 6	191.1 / 10	191.2 / 10
lud_perimeter	7.5 / 8	116.8 / 34	106.9 / 27	116.7 / 34	71.2 / 8	36.3 / 9	95.5 / 9	97.2 / 10
lud_internal	34.9 / 12	151.6 / 26	151.7 / 26	148.7 / 25	86.3 / 6	22.4 / 2	74.7 / 6	50.4 / 5
hotspot	21.1 / 4	29.4 / 9	30.2 / 7	28.5 / 8	94.8 / 9	37.6 / 7	29.6 / 8	29.4 / 8
opt_srad1	31.3 / 10	139.7 / 36	75.7 / 21	48.2 / 8	65.7 / 7	64.9 / 10	271.7 / 8	119.8 / 11
opt_srad2	12.4 / 0	37.8 / 7	28.9 / 5	31.4 / 7	90.8 / 5	7.4 / 1	68.0 / 4	50.6 / 1
bpnn_layerforward_ocl	17.8 / 5	17.3 / 5	94.8 / 19	7.1 / 3	82.6 / 6	22.3 / 4	106.8 / 4	71.0 / 8
bpnn_adjust_weights_ocl	47.9 / 8	62.4 / 11	73.9 / 8	62.7 / 11	98.2 / 11	76.5 / 10	69.6 / 11	69.5 / 11
compute_step_factor	5.8 / 3	31.3 / 10	23.6 / 7	24.6 / 9	92.5 / 11	23.1 / 6	33.5 / 11	33.3 / 10
time_step	22.0 / 6	44.0 / 11	13.5 / 4	43.0 / 11	86.8 / 11	4.9 / 4	105.8 / 11	106.2 / 11
compute_flux	49.8 / 5	13.2 / 4	43.4 / 9	13.6 / 4	83.1 / 10	39.2 / 5	28.1 / 10	42.4 / 7
average	**27.8 / 6**	**78.7 / 15**	**90.2 / 19**	**69.7 / 12**	**83.3 / 10**	**41.0 / 7**	**104.0 / 10**	**88.2 / 10**

Table 12: *MAPE* and $MAPE^{shape}$ values of the Rodinia kernels (percentages) for the GeForce RTX 2070 (Turing)

.