Memory-Aware Latency Prediction Model for Concurrent Kernels in Partitionable GPUs: Simulations and Experiments

Masola Alessio, Capodieci Nicola, Cavicchioli Roberto, Sanudo Olmedo Ignacio, and Rouxel Benjamin

firstname.lastname@unimore.it
University of Modena and Reggio Emilia, Department of Physics, Informatics and Mathematics, Italy.

Abstract. The current trend in recently released Graphic Processing Units (GPUs) is to exploit transistor scaling at the architectural level, hence, larger and larger GPUs in every new chip generation are released. Architecturally, this implies that the clusters count of parallel processing elements embedded within a single GPU die is constantly increasing, posing novel and interesting research challenges for performance engineering in latency-sensitive scenarios. A single GPU kernel is now likely not to scale linearly when dispatched in a GPU that features a larger cluster count. This is either due to VRAM bandwidth acting as a bottleneck or due to the inability of the kernel to saturate the massively parallel compute power available in these novel architectures. In this context, novel scheduling approaches might be derived if we consider the GPU as a partitionable compute engine in which multiple concurrent kernels can be scheduled in non-overlapping sets of clusters. While such an approach is very effective in improving the GPU overall utilization, it poses significant challenges in estimating kernel execution time latencies when kernels are dispatched to variable-sized GPU partitions. Moreover, memory interference within co-running kernels is a mandatory aspect to consider. In this work, we derive a practical yet fairly accurate memory-aware latency estimation model for co-running GPU kernels.

Keywords: Latency Prediction · Concurrent Kernels · GPGPU-Simulator · Partitionable GPU.

1 Introduction

In the context of cyber physical systems (CPS) and related applications, in order to cope with data and compute intensive task-sets, processing platforms usually feature multi-core CPUs able to work in concert with massively parallel compute accelerators. A nowadays very common architectural solution when dealing with compute accelerators is represented by General-Purpose Graphics Processing Units (GP-GPUs or simply GPUs), as are often chosen on the account of their impressive performance per watt ratio, high availability, and the presence of fairly mature ecosystems of APIs, libraries and development kits aiming to simplify their usability. As applications complexity grows and their compute demands increase over time, novel GPU architectural blueprints must keep up with such a trend. In this sense, GPU vendors' road-maps suggest that transistor scaling enables GPU performance to continue to rise, due to higher GPU frequencies and growth in die size [21]. This latter point translates into the increasing count of clusters of parallel processing elements embedded within a single GPU die. In Layman's terms, we can say that *larger* GPUs are released generation after generation. However, compute kernels running alone are therefore not always able to exploit this new degree of parallelism offered by novel GPUs [3,18]. As a consequence of this, system engineers are researching mechanisms and methodologies to dispatch multiple kernels onto the GPU within overlapping time windows, hence assigning a variable number of GPU cores and memory resources to different kernels according to their timing and latency requirements. GPU resources to kernels assignment can be achieved in many ways: one of such mechanism has been termed *Multi-Kernel Execution* in which groups of threads belonging to different kernels might be distributed to all the GPU's compute clusters. This solution implies that blocks of threads of different kernels share the same clusters within overlapping time windows. Despite being proven to be very effective in increasing overall throughput [24], it is very difficult to control individual kernel latencies in such a scenario [17]. This is due to the fact that GPU threads are not only forced to compete for compute resources (i.e. internal cluster schedulers, ALU pipelines etc...) but also for the GPU cluster's cache hierarchy and local scratch-pad memories.

On the other hand, spatial kernel partitioning can be considered: in such a paradigm, the GPU's compute clusters are divided within non-overlapping partitions so that one or more clusters are assigned to individual kernels. In this way, the GPU last level cache (LLC, L2) is the first and most important contention point when multiple kernels are executing within overlapping time windows, which allows the system engineer to focus on a single memory contention point, once the scaling of kernels execution time as a function of assigned GPU partitions is known. Kernel execution time scaling on variable compute cluster partitions is not trivial to derive.

While ideally such a scaling function should be linear, i.e. the kernel's execution time decreases linearly as we linearly assign more GPU compute clusters, the GPU parallel capabilities might exceed the kernel degree of parallelism, hence reaching the theoretical performance peak without actually using the entire sets of GPU partitions. Moreover, memory bandwidth plays a crucial role, as highly memory-bound kernels cause L2 to VRAM memory bandwidth to act as a bottleneck so to further hinder the expected performance scaling. This latter aspect has been extensively studied and it is known as GPU roofline model [26,14].

In this work, we therefore aim to study the behaviour of GPU spatial partitioning when multiple kernels run concurrently in separated GPU partitions. More specifically, we present an intuitive, yet practical memory-interference aware performance prediction model that takes into account the individual kernels' features and memory behaviour. Compared to previous literature, our proposed methodology is able to provide reasonable accuracy when predicting kernels' execution time, and it is able to scale well when increasing the number of concurrent GPU kernels. This research is a preliminary study aimed at managing memory interference in partitions, with the future ultimate goal of assisting in the design of effective schedulers for multiple concurrent execution of kernels in a GPU.

2 Background

In this section, we briefly summarize the most important architectural characteristics of a GPU, the GPU programming model and the chosen simulation environment. As far as GPU terminology is concerned, without excessive loss of generality we will introduce and use throughout this paper the NVIDIA terminology in the context of the widely adopted CUDA API[1].

2.1 GPU Architecture

Even if the original purpose of GPUs was to accelerate graphic processing, in recent decades a GPU can be thought of as a massively parallel compute accelerator able to be deployed in a wide variety of general purpose scenarios [16]. Hardware-wise, the GPU execution model is a hybrid between a SIMT (Single Instruction Multiple Threads) and SIMD (Single Instruction Multiple Data) parallel compute engine, in which multiple instructions are executed by a large number of ALU (Arithmetical Logical Unit) pipelines in a lock-step fashion. There are evident similarities among GPUs released by different device vendors, as GPUs' ALU pipelines are always grouped into clusters of processing elements. Inside each processing cluster, both L1 cache and scratch-pad memories are shared among the ALU pipelines within the cluster. Just outside the cluster and shared among the other clusters, a last level cache (LLC) is present. This latter memory is connected to the rest of the system with an interconnection fabric. In case of a discrete GPU, this represents the connection between the LLC and the dedicated VRAM (Video Random Access Memory), or directly to system RAM in case of integrated GPUs commonly implemented in embedded System-on-Chips.

In NVIDIA terminology, ALU pipelines are named *CUDA cores* (Figure 1), which are grouped into computing clusters named *SM, Streaming Multiprocessors*. Inside an SM, the explicitly addressable scratch-pad memory is called *shared memory*. An L1 Cache is also present.

2.2 Programming Model and Scheduling

Software-wise, implementing a GPU application means tackling a heterogeneous programming problem. This is because the host CPU has the duty to submit copy and compute commands to the GPU and this is achieved with specific APIs designed to simplify access to the GPU compute acceleration

[1] https://docs.nvidia.com/cuda/

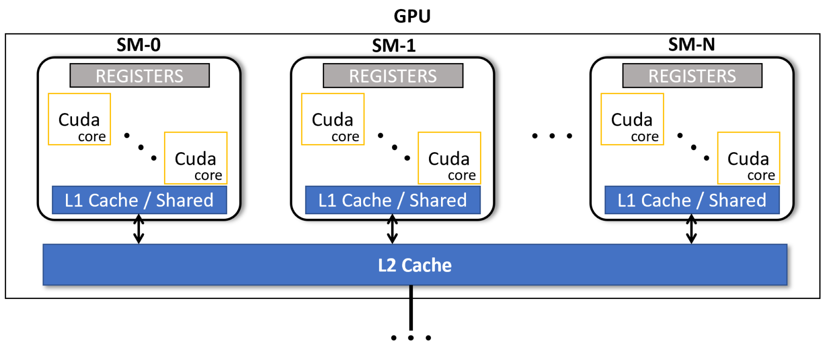


Fig. 1: GPU architectures are composed of a set of SMs with L2 cache shared among each SM.

capabilities [5]. Examples of such APIs are the NVIDIA CUDA (Compute Unified Device Architecture), which is a proprietary API, and OpenCL[2] which, in contrast, is an open-standard.

In GPU APIs such as CUDA and OpenCL, commands submitted from the CPU to the GPU are related to the execution of the parallel programs onto the GPU's processing clusters. These GPU runnables are commonly referred to as *kernels*. A GPU application, therefore, has to manage data allocation, buffer movements from the CPU-only visible to the GPU-only visible memory areas (and vice versa) and kernels' dispatch calls.

The programmer has very little margin to influence the scheduling mechanisms of a GPU application. Kernels are dispatched through parallel queues of commands, that in CUDA terminology are called *streams*. In NVIDIA GPUs, commands enqueued in the same stream are executed following a FIFO order. Multiple kernels belonging to different streams might execute in parallel if the occupancy of a kernel within a stream does not saturate the compute capability of every SM. Blocks of threads of each kernel tend to be distributed among **all** the sets of SMs in a round-robin fashion: in this way, if a kernel does not saturate all the memory (i.e. shared memory and register) or compute resources (i.e. CUDA cores) of the SMs, another kernel of a different stream can execute in parallel [17].

It is important to note that the current CUDA APIs available on consumer-level GPUs do not allow programmers to control how kernels are assigned to *partitions* on NVIDIA GPUs. There is no publicly available method to specify which partitions should be used for a particular kernel in the CUDA programming model. As a result, the baseline CUDA stream scheduler will enforce resource sharing among blocks of threads belonging to different kernels within a single SM, which can impact the predictability of execution latencies.

The term *partition* refers to a selected subset of SMs that are, in our case, contiguous, meaning that they are ordered by their IDs. For example, if there is a GPU with 30 SMs numbered from 0 to 29, a contiguous partition of 10 SMs could include SMs with IDs ranging from 0 to 9, while a non-contiguous partition is one that includes SMs with IDs that are not ordered.

Workarounds for research and experimental purposes have been proposed for enabling inter-SM resource partitioning rather than the default behaviour (e.g. CUDA persistent threads [8]). Due to evident limitations on GPU partitioning present in these workarounds, detailed in Section 9, we implemented and discuss our memory-aware latency prediction model on partitionable GPUs on a cycle-accurate simulator (GPGPU-Sim).

2.3 Cycle Accurate Simulation Through GPGPU-Sim

In order to achieve cycle-level accuracy in simulating novel hardware architectures, we implemented and simulated our approach by means of the widely used GPGPU-Sim [13]. GPGPU-Sim is a cycle-level simulator capable of modelling arbitrary GPU architectures and executing computing workloads written using widely known APIs such as CUDA or OpenCL. As demonstrated by the research group that develops and maintains GPGPU-Sim, this simulator allows us to analyze the majority of architectures available on the market, somehow compensating for the scarce amount of documentation on low-level architectural details that are usually provided by manufacturers.

[2] https://www.khronos.org/opencl/

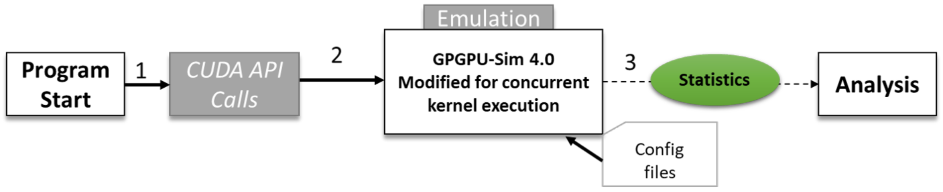


Fig. 2: Workflow from host to GPGPU-Sim.

In order to evaluate the correlation between the simulations and the real GPU hardware behaviour, GPGPU-Sim authors observed the Instruction Per Clock (IPC) measured through simulations and the average value obtained on real NVIDIA GPUs for a set of benchmark kernels. An impressive similarity index of 98.37% was reached. This value allows us to assume that the analysis performed with this simulator is reliable; it also enables us to modify some parts of the hardware characteristics in order to expose the limits of the existing GPU devices and to test architectural improvements for further optimization. In our work, we used GPGPU-Sim for timing analysis (emulation cycles) and measure the interference occurring at the L2 cache memory level during the concurrent execution of multiple kernels. The L2 is the point in the memory hierarchy in which we focus our attention, as this is the first contention point for GPU thread blocks of different kernels mapped in non-overlapping sets of SMs. In order to be able to define the mapping of concurrent kernels within user-defined SM partitions, we created an ad-hoc extension for GPGPU-Sim.

GPGPU-Sim produces a compiled library that allows the simulator to interpret the compiled instruction program set of CUDA device low-level source code (called PTX - Parallel Thread Execution) and simulates it within the desired hardware architecture (Figure 2). The emulated architecture is composed of many Single Instruction Multiple Thread (SIMT) cores that model multithreaded pipelined Single Instruction Multiple Data (SIMD) execution units. Such execution units are grouped within clusters that represent what NVIDIA calls Streaming Multiprocessor (SM). Each SM contains different ALU units, application specific co-processors, and the whole memory hierarchy and interface we described in the previous section. The emulated SMs are connected to the memory partition through an interconnection network. From the interconnection network, packets that contain operations to perform are injected into a SIMT FIFO (First Input First Output) queue that manages and redirects them to the corresponding SIMT SM instruction cache. The packets contain a memory response that could be servicing an instruction fetch miss or a memory pipeline (through a Load Store unit - LDST). Instructions loaded in the instruction cache, are then fetched, decoded, and the operations to perform are eventually executed by the entire core set.

3 Overview

Our methodology is illustrated in Figure 3. It details the steps taken to conduct an in-depth analysis to derive the different predictive models presented in this paper. In order to study the behaviour of a partitioned GPU, we define the concept of dynamic partition in a GPU so to be able to assign SMs to kernels and, in the following section, we describe the engineering effort that we put into GPGPU-Sim in order to modify its thread block scheduling algorithm to enable multiple concurrent kernels executions within partitioned sets of SMs.

With the necessary infrastructure in place to perform our study (Figure 3 blocks a and b), we proceed with two profiling phases: 1. isolation profiling and, 2. parallel kernel profiling.
In the isolation profiling phase, detailed in Section 5, we profile the kernels in a baseline scenario in which isolated kernels are potentially dispatched among the whole set of available SMs (default GPU behaviour). This allows us to collect a profile of each kernel's execution times and memory behaviour. Then, we observe how this profile evolves when the same kernel is mapped onto a subset of SMs, Figure 3 block c. In this way, we construct the completion latency and requested memory bandwidth (BW) vs SM partition size trend for each kernel. Such profile is then used to find a predictive model that allows the system engineer to derive the kernel behaviour (Figure 3 block d) in isolation, without the need to test all possible partition sizes for each specific kernel (Section 6).

In the parallel profiling phase (Section 7), we aim to find the interference effect of multiple kernels running concurrently. Therefore, we collect memory accesses of co-running kernels, in order to infer a predictive model able to derive the kernel completion latency including memory interference, Figure 3 blocks e and f.

We also present a method in Section 8 for predicting the L2 cache BW required by a single kernel executing in isolation on varying sets of SMs, Figure 3 block g.

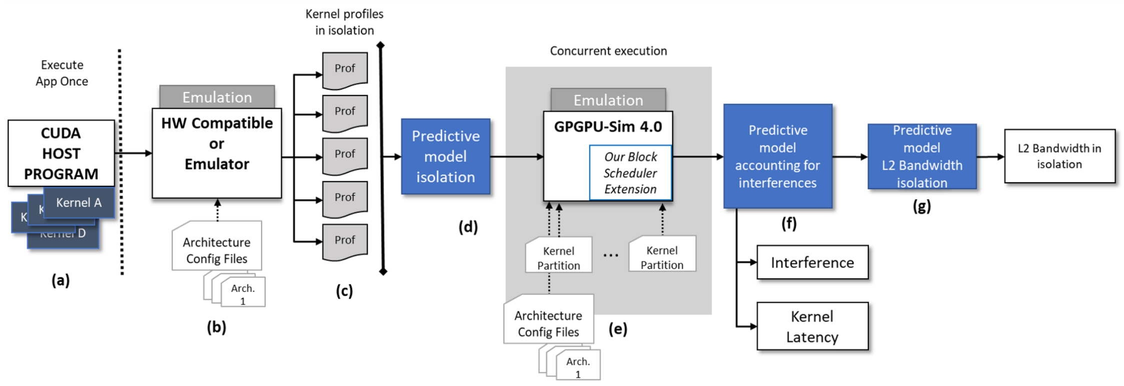

Execute App Once
CUDA HOST PROGRAM
Kernel A
Kernel D
(a)
Emulation
HW Compatible or Emulator
Architecture Config Files
Arch. 1
(b)
Kernel profiles in isolation
Prof
(c)
Predictive model isolation
(d)
Concurrent execution
Emulation
GPGPU-Sim 4.0
Our Block Scheduler Extension
Kernel Partition
Architecture Config Files
(e)
Predictive model accounting for interferences
(f)
Predictive model L2 Bandwidth isolation
(g)
L2 Bandwidth in isolation
Interference
Kernel Latency

Fig. 3: Methodology overview.

4 Simulation Settings and our GPGPU-Sim Extension

We chose to simulate an NVIDIA RTX 2060 as the target GPU for our experiments, since it is the most recent hardware that GPGPU-Sim Version 4.0.0 can emulate, and, also the one with the highest correlation index with respect to real hardware (99% [13]). Table 1 shows our configuration environment. In order to analyze our target scenarios, we modified GPGPU-Sim to enable the concept of mapping a GPU kernel to a subset of available SMs. Our modification of GPGPU-Sim implies a special configuration file given as input. During the simulator initialization phase, this configuration file is loaded by the `createSIMTCluster()` function, located in *gpu-sim.cc* source file.

Table 1: GPGPU-Sim configuration for a NVIDIA RTX 2060.

Name	NVIDIA RTX 2060
Clusters	30
SM per Cluster	1
Total SM	30
L1 Cache	4 banks, 64 KB per SM
L2 Cache	24 banks, 128 KB block, total 3MB
Machine ISA	sm_75
Nominal BW	348 GB/s

Our configuration file contains at each line: a kernel name and a range of integer values corresponding to SMs IDs. The kernel names, retrieved after the first execution, are used in the configuration file. The ranges associated to the kernels represent the partition on which the kernel will be mapped. We assume that such partitions are contiguous and un-fragmented: each kernel will be mapped to a partition composed of contiguous SMs, and the different partitions are prevented from overlapping. The mapping is therefore stored as a data structure visible by all emulated streaming multiprocessors. It implies having a static task-set with which static partitions can be defined. We extended GPGPU-Sim in order to dynamically create kernel-to-SM-partition mapping during the execution of the whole simulation.

From the CPU side code (host), multiple kernels are dispatched using multiple streams. The emulated CUDA runtime divides the work belonging to the different kernels in thread blocks, also known as

CTAs (Cooperative Thread Arrays) that each individual emulated SM is able to schedule. In the vanilla version of GPGPU-Sim, this is implemented by first selecting a kernel, then considering dispatching blocks belonging to that kernel, which follows the regular stream scheduling logic (see Section 2.2). More specifically, a method gets called for every SM instance (called SIMT cluster in GPGPU-Sim) to retrieve a kernel to execute. Once a kernel is selected, its CTAs are dispatched in a round-robin fashion among the SMs.

We modified the kernel selection method, which is called `select_kernel()` in order to restrict the eligible kernels among the ones that have been associated to the partition containing the SIMT cluster in which CTAs are going to be dispatched. This will effectively map thread blocks of specific kernels to pre-defined sets of SMs (Figure 4).

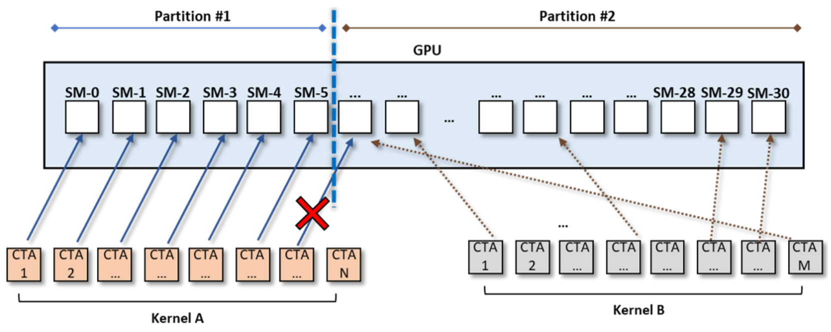


Fig. 4: Example of 2 kernels mapped onto 2 GPU partitions with our GPGPU-Sim extension.

We have written a host CUDA application that launches different CUDA kernels in different CUDA streams in order to perform our simulations. Such a benchmark application is used in combination with multiple settings for our added configuration files in order to be able to observe the behaviour of concurrently executing kernels in different partition sizes. We aim to understand how the performance of individual kernel scales when executed in varying partitions and by accounting for memory interference on shared memory hierarchy, i.e. the GPU L2 cache.

5 Memory Aware Performance Estimation

The initial profiling phase involves eight different kernels: these kernels have different computational and memory requirements and range from synthetic or very basic operations to significantly more complex kernels. Such kernels are either implemented by ourselves or taken from known benchmark suites. The group of kernels is composed of:

– *Vector add* (VADD) performs a vector addition.
– *Single Precision A X plus Y* (SAXPY) is a combination of scalar multiplication and vector addition.
– *Copy* (COPY) operates a copy between two device-side buffers.
– *Ray tracing* (RAYTRACE) performs a ray-tracing based shading on a 3D scene.
– *Direct X texture compression* (DXTC) is a texture compression algorithm.
– *Convolutional* (CONV) is a convolutional kernel over 2D matrices.
– *Matrix vector product and transpose* (MVT) realises operations on a matrix.
– *Path finder* (PF) is a path finding algorithm.

The kernels, their launch configurations, the working set size and other important information are listed in Table 2.

During this phase, we profiled the cycles required by each kernel to complete, the L2 memory BW, and the number of accesses to the GPU L2. These metrics were collected for each kernel individually while varying the SM partition size. The number of SMs assigned for each kernel across the different experiments is $\in [5, 10, 15, 20, 25, 30]$.

Table 2: Kernels used for profiling on an NVIDIA GeForce RTX 2060 emulated architecture.

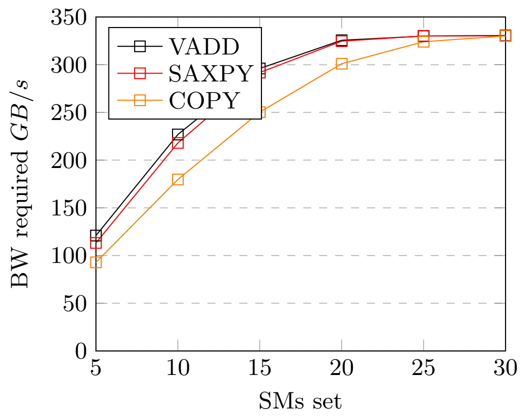
Kernel	**Input**	**Output**	**Shared Mem.**	**Launch Config.**	**KIA**	**Duration**	**Origin**
	MiB	MiB				ms	
VADD	40*2	40	0	(81920,1,1),(128,1,1)	0.056	0.379	Synthetic / In house implementation
SAXPY	40*2	40	0	(81920,1,1),(128,1,1)	0.059	0.379	Synthetic / In house implementation
COPY	40	40	0	(81920,1,1),(128,1,1)	0.064	0.271	Synthetic / In house implementation
RAYTRACE	-	1.5234	0	(104,80,1),(8,8,1)	3.575	2.621	In house Implementation
DXTC	0.003 + 0.5	0.25	1.5 Kb	(46,1,1),(512,1,1)	149.265	3.273	Cuda Samples
CONV	0.00001 + 40	40	128 B	(327680,1,1),(32,1,1)	0.291	1.089	In house Implementation
MVT	64 + 0.01*4	0.01*2	0	(64,1,1),(64,1,1)	0.015	2.878	Polybench
PF	0.5 + 255.5	0.5	2 Kb	(1024,1,1),(256,1,1)	0.475	0.337	Rodinia [6]

5.1 Kernels Memory Bandwidth Analysis

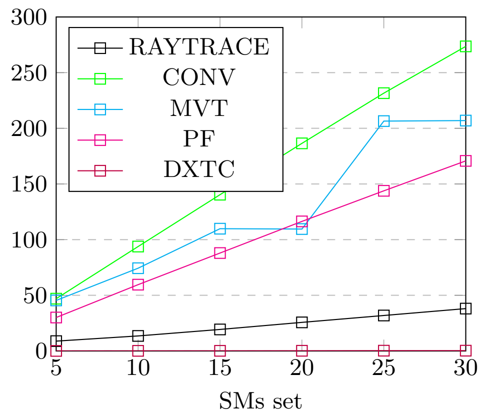
First, we observe how the requested memory BW changes for each kernel as we scale up the number of SM in which they are scheduled to be executed, shown in Figure 5. Trivially, access to the memory interface by different SMs occurs in parallel, as each SM is directly connected to different L2 banks. This implies that, as we scale down the number of SMs assigned to a kernel, its memory BW tends to decrease, as the parallelism in which the memory interface is accessed is also decreased.

Out of this first profiling phase of experiments, we can make an initial quantitative categorization of the maximum BW required by each kernel to access the L2 cache when dispatched on all the 30 SMs. We labelled the kernels in 3 different types:

- *Memory Intensive*: $\{VADD, SAXPY, COPY\}$ the BW demand ($\geq 70\%$) reaches a saturation point, and does not scale linearly, even if we increase the SM count, Figure 5a.
- *Hybrid*: $\{CONV, MVT, PF\}$ the BW demand ($[10\%, 70\%)$) scales linearly based on the number of SMs with average utilization of compute units, Figure 5b.
- *Computational*: $\{RAYTRACE, DXTC\}$, the BW utilization is low ($< 10\%$) while the computation requirement linearly increases with the number of SMs, Figure 5b.



(a) Memory intensive kernels



(b) Hybrid/compute kernels

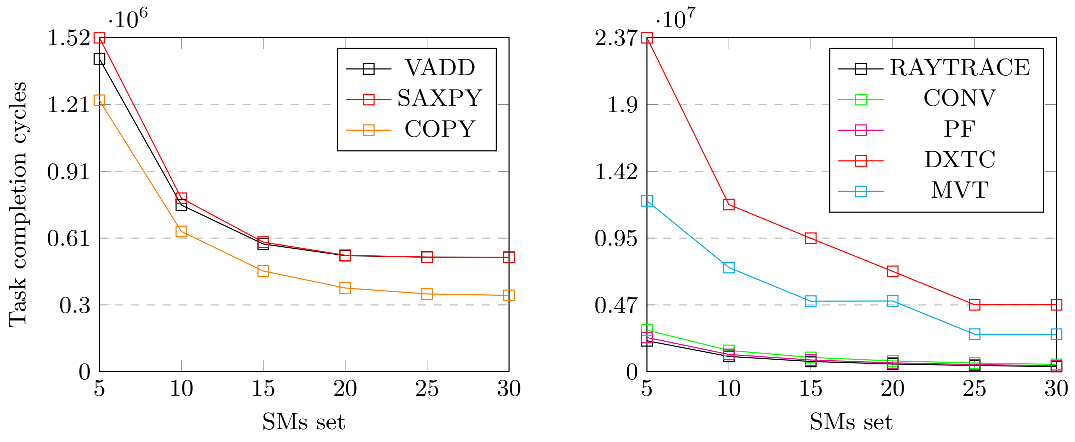
Fig. 5: Memory BW required by each kernel.

We performed multiple tests to understand whether there were any profiling errors on MVT, but we could not understand why MVT (Figure 5b) has an unexpected behaviour when the set of SMs is 20 and 30 having similar near values to the previous sets. Such strange behaviour could be driven by the simulated architecture or by the kernel behaviour.

We also note that the saturation point of the BW is 330 GB/s while the nominal BW of the architecture is 348 GB/s. This saturation point will be later referred to as the Effective Maximum Bandwidth (EM_{BW}).

5.2 Kernels Completion Latency Analysis

Second, we focus on how the latency completion time scales with the number of assigned SMs. The collected results are summarized in Figure 6 in which execution latencies are expressed in simulation cycles. As far as memory intensive kernels are concerned, we observe a decreasing exponential behaviour of the kernel latencies that tend to stabilize as soon as the memory saturation point shown in Figure 5 is reached. In other words, the expected behaviour of a linearly decreasing latency gets *interrupted* due to memory latencies being prevalent over instruction execution times. Such a behaviour is not exhibited in the other kernel type: while not being perfectly linear, a plateau is never reached in most of the tested kernels, instead, we observe an uninterrupted exponential decrease in the completion times.



(a) Memory intensive kernels (b) Hybrid/compute kernels

Fig. 6: Completion latencies analysis for each kernel (cycle analysis).

These in-depth analyses of BW and kernel latency shed some lights on the construction of further predictive models.

6 Predicting Latencies Depending on Assigned SMs

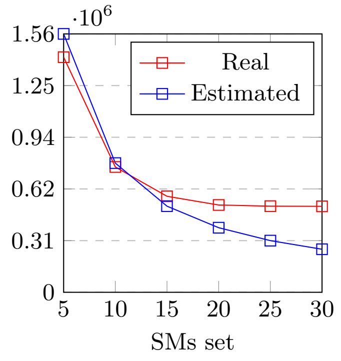
With the above initial results, it is trivial to understand that the number of SMs strongly influences both the memory BW and the kernel completion latency. More specifically, for non-memory intensive kernels, a linear demand on memory produces a proportional scaling on completion cycles. For memory intensive kernels, as memory requirement increases, the ability of the kernel to fully exploit the available parallelism of a larger SM count is jeopardized. Hence, our predictive model is the first to account for this kernel categorisation of memory/latency behaviour.

We first determine the BW utilisation $U_{bw}(k)$ of a kernel k, eq. 1, as the ratio between the maximum nominal L2 BW (BW_{max}), i.e. 348 GB/s with our hardware architecture, and the effective requested BW by the kernel when executed on the maximum available SMs N, noted as $BW_{isolated}(k, N)$.

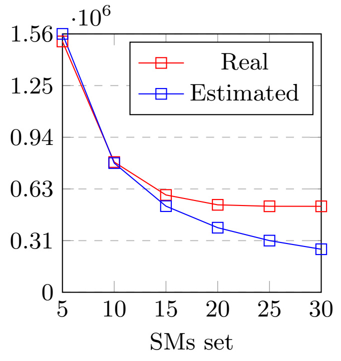
$$U_{bw}(k) = \frac{BW_{isolated}(k, N)}{BW_{max}} \tag{1}$$

Then, we define a *memory saturation memory* $Sat(U_{bw}(k))$, eq. 2, based on the BW utilisation $U_{bw}(k)$, eq. 1. By analyzing the equation (1), the more $U_{bw}(k)$ is closer to 1, the faster the memory BW saturation point is reached. Hence, the saturation function $Sat(U_{bw}(k))$ is able to capture the BW behaviour of all three kernel categories.

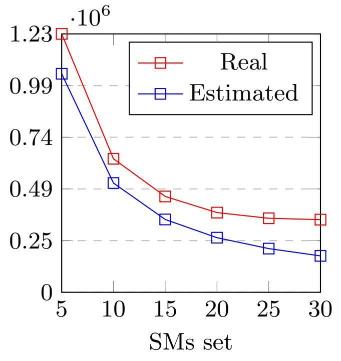
$$Sat(U_{bw}(k)) = \frac{1}{1 + e^{-\alpha(U_{bw}(k) - S_p)}} \tag{2}$$



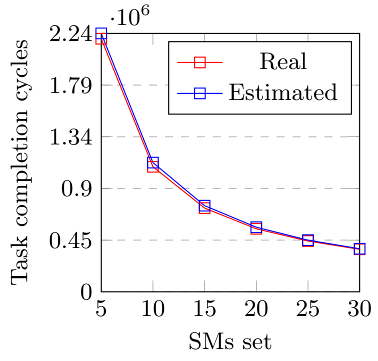
(a) VADD



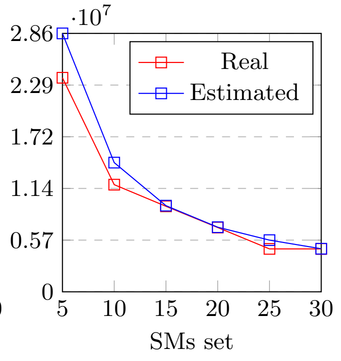
(b) SAXPY



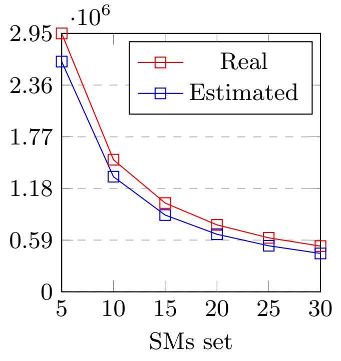
(c) COPY



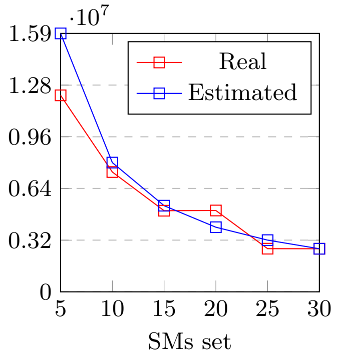
(d) RAYTRACE



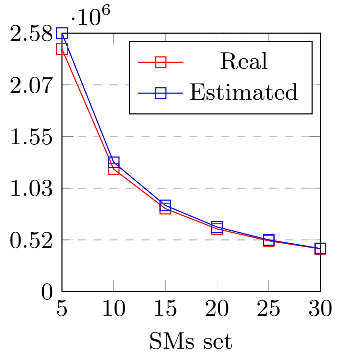
(e) DXTC



(f) CONV



(g) MVT



(h) PF

Fig. 7: Kernel latency prediction with varying SM size using eq. 4.

In eq. (2), the term α is used to model the saturation function steepness. We experimentally tested several values for our predictive model and found the best results when $\alpha = 100$. The S_p term represents the *saturation point*, which corresponds to a theoretical individual L2 memory bank per SM. In the GPU we are simulating, the L2 is composed of 24 banks to be accessed by 30 SMs, hence $S_p = 24/30 = 0.8$ gives us an estimation of the saturation point within the L2 cache.

We further define a *factor able to estimate the memory intensity* KAI, eq. (3), as the ratio between the total number of executed instructions and the number of L2 accesses, scaled down by a factor of 1000.

$$KAI = \frac{ExecutedInstructions}{L2Accesses} \times \frac{1}{1000} \tag{3}$$

At last, we define our kernel latency prediction function $T_c(k, n)$, eq. 4, that infers *the simulation cycles* needed by a kernel k to complete on n SMs, with $n \leq N$. $C_k(N)$ is the total cycles required by the kernel to complete in isolation with the full set of N SMs, see Fig. 6. It is easy to realize that equation (4) is a proportion calculated by scaling the completion time for the fraction of utilized SMs, and such proportion gets biased as kernel memory utilization increases.

$$T_c(k,n) = \frac{C_k(N)}{(Sat(U_{bw})+1)\cdot(N - KAI \cdot U_{bw})} \cdot \frac{N^2}{n} \tag{4}$$

Analyzing the behaviour of equation (4) compared to the actually simulated execution cycles in GPGPU-Sim, in Figure 7, the prediction has a larger error in the 3 memory intensive kernels, while it works fine in the other cases. However, it has been observed that the errors committed by our predictive formula, as analyzed on individual kernels and depicted in Figure 8, range from an underestimation of 30% to a maximum of 10% in situations where excessive memory usage is not required. In predictive models for real-time systems, it is essential to establish a margin of correctness. This is because it is improbable that the prediction will match the reference value precisely. In our results, we set an arbitrary margin of 20%, which aligns with the industry and academic practices in many application domains for estimating the worst-case execution time (WCET) [25,20,19]. Analyzing the absolute error produced by the predictive formula (Figure 8), we observe that the kernels with the highest bandwidth demand (VADD, SAXPY, and COPY) are the most challenging to predict due that the formula does not catch the exact behavior of the three kernels. Instead, the remaining ones, including the hybrid and compute-intensive kernels, show good results, remaining within the arbitrary margin of 20% of WCET estimation domain.

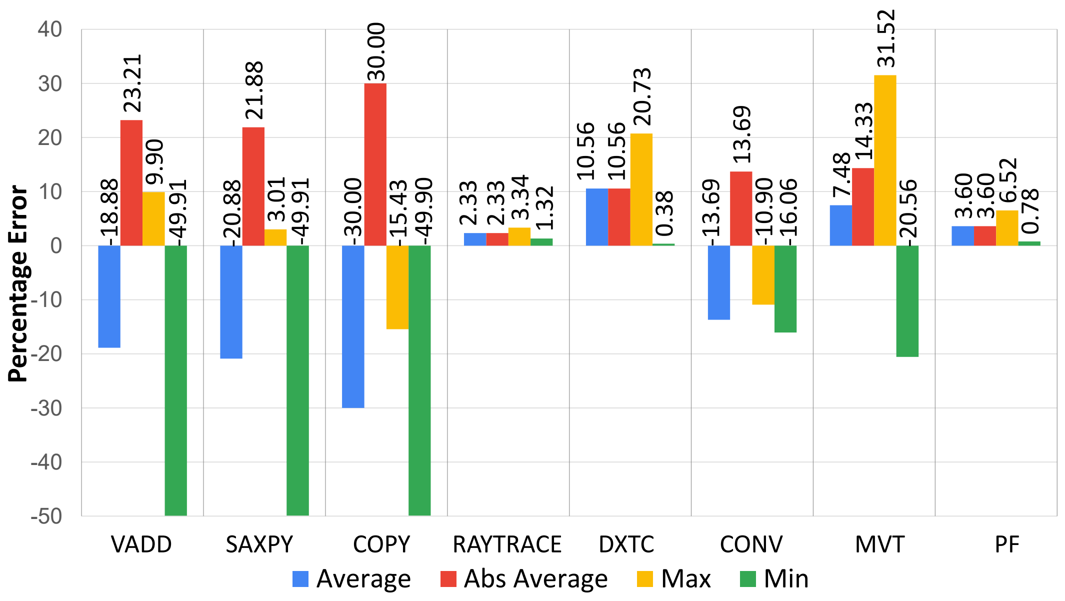


Fig. 8: Cycles prediction errors, using eq.(4) . Negative values are underestimations.

7 Modeling Memory Interference

Previous Section 6 defined $T_c(k,n)$ to effectively predict the number of cycles required for a kernel to complete in isolation as the number of SMs varies. When multiple kernels are concurrently executing on the same architecture, they all interfere with each other when accessing the memory, which has an impact on completion latencies.

In this section, we define a model for the amount of interference that can occur when two or more concurrent kernels are allocated to different and non-overlapping SM partitions on the GPU. This involves considering two very important aspects: 1. *modelling the decrease in the required memory BW of a kernel when mapped onto a n-sized partition, such as $n < N$*, and 2. *if m concurrent kernels run in different partitions, and each has its own memory BW requirement, then the sum of those BWs might exceed BW_{max}*. In this case, the latency of the running kernels will deteriorate due to interference.

It is also reasonable to assume that the more the available BW limit is exceeded, the higher the magnitude of interference and related performance deterioration will be observed. Following is a series of experiments testing our assumptions.

7.1 Experiments and Analysis

To analyze the interference behaviour, we conducted 7 experiments with the objective of covering both critical (total required BW < max BW) and non-critical (total required BW $\geq$ max BW). For each experiment, labelled from 1 to 7, the number of concurrent kernels varies between 2 and 6. The kernels involved in each experiment are then mapped to non-overlapping sets of SMs, such that the sum of the size of each partition of each kernel in a single experiment coincides with the maximum number of SMs ($N = 30$). These settings are summarized in Table 3, which lists the seven experiments, indicating for all concurrent kernels: their SM partition size, and their theoretical requested memory BW, calculated as the sum of the individual memory BW requirement as measured in isolation (Section 5.1). All the kernels in the experiments run to completion.

Table 3: Memory BW theoretical requirements: experimental settings.

ID	Kernel Name and SMs	Individual theoretical BW (GB/s)	Total theoretical BW (GB/s)
1	VADD 15 | RAYTRACE 15	296.03 | 19.32	315.35
2	VADD 5 | SAXPY 5 | MVT 20	120.9 | 113.19 | 109.5	343.61
3	RAYTRACE 10 | DXTC 10 | PF 10	13.39 | 0.09 | 59.55	73.14
4	COPY 15 | SAXPY 15	250.33 | 291.77	542.11
5	VADD 15 | CONV 15	296.03 | 140.37	436.40
6	VADD 20 | MVT 10	325.58 | 74.33	399.91
7	PF | MVT | RAYTRACE | DXTC | CONV | COPY each with 5	30.04 | 45.39 | 8.84 | 0.04 | 46.96 | 92.87	224.16

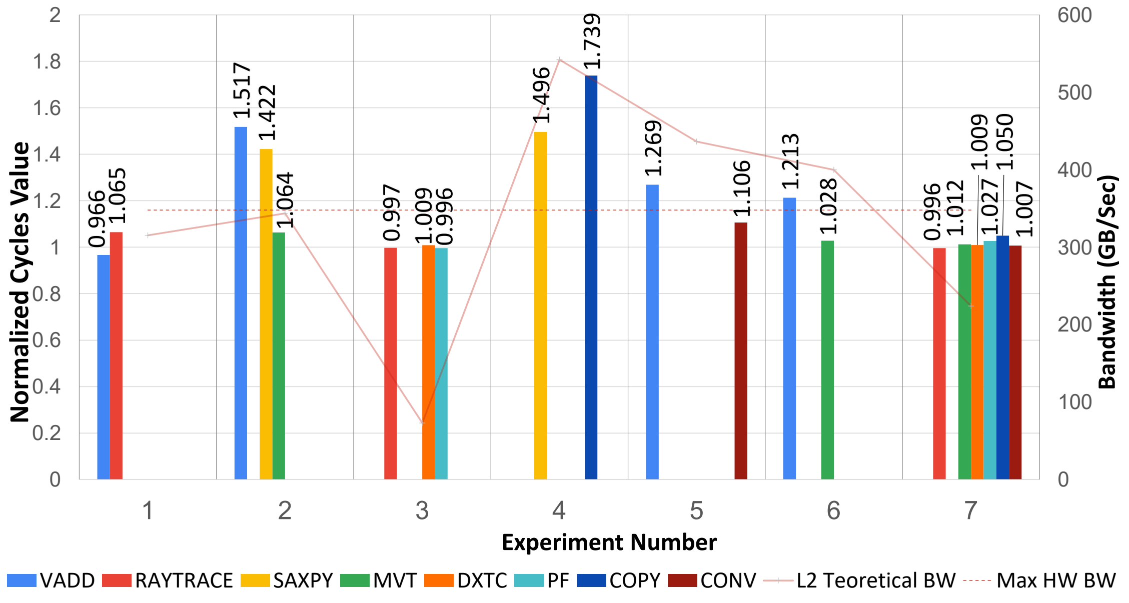


Fig. 9: Observed kernel latency deviation, requested BW (solid line), and nominal BW (dotted line).

Figure 9 exhibits the kernel latency deviation with the ratio $\frac{concurrent}{isolation}$ for the respective set of SM. It shows that interference exists and is likely to change depending on the memory access patterns of concurrently executing kernels and L2 cache accesses. From these experiments, we can also observe that the increase of kernel latency is dependent on the initial requested BW (solid line, right y-axis) compared to the nominal BW (dotted line). This consideration is the first pillar of our further prediction model.

Figure 10 presents the actual execution pattern, starting and completion time (in cycles) of its corresponding set of kernels. Zooming in on this execution view, we identified a set of time windows (separated by vertical dotted lines), hereafter W, in which the set of concurrently running kernel changes. We assume a function $K(w)$ which returns the set of kernels present in the execution window $w \in W$. For example, in the last window $w^l \in W$ only $DXTC \in K(w^l)$ executes, while in the previous one $w^{l-1} \in W$, both $\{MVT, DXTC\} = K(w^{l-1})$ are concurrently executing. Moreover, we can compute the proportion of execution $P_{overlap}(k,n,w)$ of a kernel k mapped on n in a window w, assuming $k \in K(w)$, using eq. (5), with $C_k(n)$ from Fig. 6. This notion of execution windows is the second pillar of our predictive model.

$$P_{overlap}(k,n,w) = \frac{size(w)}{C_k(n)} \tag{5}$$

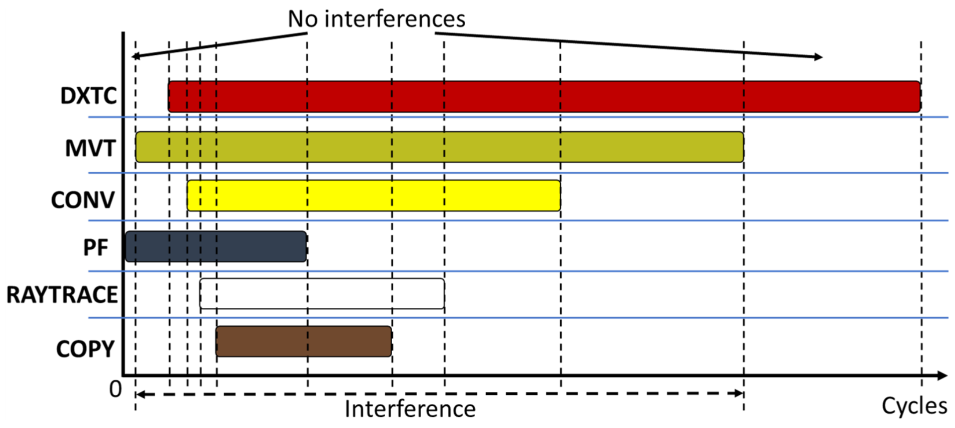


Fig. 10: Experiment 7 overlapping time windows of PF/MVT/RAYTRACE/DXTC/CONV/COPY, 5 SMs each.

However, to avoid the necessity to actually execute the set of kernels, we must make a set of assumptions to construct the set of execution windows W:

1. all kernels start at time 0, and
2. all kernels run for their worst-observed execution time in isolation on the designed set of SMs, i.e. Section 5.2.

For each overlapping time window $w \in W$, we can compute a portion of additional execution that a kernel will suffer following the two pillars applied in equations (6) and (7). The former applies a penalty only if the requested BW in the time window is higher than the effective BW EM_{BW}, and the latter computes the interference factor $I(w)$ of the over-requested BW. If the BW requested by the set of kernels $K(w)$ exceeds the effective BW, the penalty must be increased proportionally. In eq. (7), S_p, BW_{max}, and $BW_{isolated}(k,n)$ are identical as in Section 6.

$$\Pi(k,n,w) = \begin{cases} 1 & \sum\limits_{l \in K(w)} BW_{isolated}(l,n) < EM_{BW} \\ Penalties(k,n,w) & \sum\limits_{l \in K(w)} BW_{isolated}(l,n) \geq EM_{BW} \end{cases} \tag{6}$$

$$I(w) = MAX(1, \frac{\sum\limits_{l \in K(w)} BW_{isolated}(l,n)}{BW_{max} \times S_p}) \tag{7}$$

We therefore define a function $T_c(k, n, W)$, eq. (8), which predicts the completion latency accounting for the interference of a kernel k mapped on n SMs considering a set of execution window W. For each time window $w \in W(k)$ where k is present, it sums up the execution time portion of the kernel augmented with some penalty cycles.

$$T_c(k,n,W) = \sum_{w \in W(k)} T_c(k,n) \times P_{overlap}(k,n,w) \times I(w) \times \Pi(k,n,w) \tag{8}$$

We identified three penalties a kernel k can suffer from, eq. (9)[3]. The $Penalty_{SM}$, eq. (10), serves to guide the inequity of the partitions amongst the kernels mapped in the same window when kernel k is mapped on n SMs. The $Penalty_{access}$, eq. (11) keeps track of the level of interference experienced by kernel k on a portion of its execution as if the L2 accesses were evenly distributed, with eq. (13), amongst the kernels present in the same time window w. Finally, the $Penalty_{BW}$, eq. (12) is used to capture the imbalance of BW demand considering that the BW is evenly shared, with eq. (3), amongst the kernels present in the same window w.

$$\overline{Penalties} = Penalty_{SM} \times Penalty_{access} \times Penalty_{BW} \tag{9}$$

$$Penalty_{SM} = \frac{N}{n} \times \frac{1}{size(K(w))} \tag{10}$$

$$Penalty_{access} = \frac{L2Accesses(k,w)}{L2Access(w)} \times P_{overlap} \tag{11}$$

$$Penalty_{BW} = \frac{BW_{isolated}(k,n)}{Equity_{BW}(w)} \tag{12}$$

$$L2Access = \sum_{l \in K(w)} L2Access(l,w) \tag{13}$$

$$Equity_{BW} = \frac{BW_{max}}{size(K(w))} \tag{14}$$

However, we must bound the term $\overline{Penalties}$ to avoid unrealistic scenarios. The $\overline{Penalties}$ term represents the percentage number of cycles that needs to be added to the kernel latency to account for interference, it therefore cannot be lower than 1. On the other hand, having $\overline{Penalties}$ above 2 means that a single kernel concurrently running with another can, in the worst-case scenario, be totally stalled on L2 cache accesses, and a BW demand at 0. This case is impossible to reach with our working hardware architecture. We define $Penalties$ as in eq. (15) and to use it in previous equation (6). To effectively manage the kernel that addresses the majority of access requests and accommodates more requests to the memory controller, we elect a single kernel for each time window that is privileged and less penalized based on its high traffic BW demand. The elected kernel has the $Penalties$ set as the scenario where the $\overline{Penalties(k,n,w)}$ is less than 1.

$$Penalties = \begin{cases} \min(Penalties_{BW}(k,n,w), I(w)) & \overline{Penalties(k,n,w)} < 1 \\ Penalty_{MPD}(k,n,w) & \overline{Penalties(k,n,w)} \geq 2 \\ \overline{Penalties(k,n,w)} & Otherwise \end{cases} \tag{15}$$

In order to leverage the $2x$ term in $Penalties$, we define a Max Penalty Density ($Penalty_{MPD}$), eq. (16a). The concept of the request density of a kernel k, DR_k eq. (16b), and the maximum density value done in a certain time window w, is given by the term MDA_w, eq. (16c).

$$Penalty_{MPD}(k,n,w) = \frac{DR(k,n,w)}{MDA(w)} \tag{16a}$$

$$DR(k,n,w) = \frac{L2Access(k,w)}{T_c(k,n)} \tag{16b}$$

$$MDA(k,n,w) = \max_{k \in K(w)} \left(\frac{L2Access(k,w)}{T_c(k,n)} \right) \tag{16c}$$

7.2 Comparison with a Worst-interference Method

Most of the related works we could find aim to predict the efficiency, e.g. [28], of kernels while we aim to predict the execution time in presence of interference. We compare our method to a predictive

[3] Arguments (k, n, w) in eq. from (9) to are skipped for clarity.

model that considers the worst possible interference happening during concurrent executions of kernels, similarly to [28]. Then, a margin is added to the kernel latency in isolation to form a prediction of the latency accounting for interference.

Let us consider a set of kernels with different in-isolation latencies. The kernel X with the largest latency experiences interference from other kernels up to the completion of the second largest one. Then, the kernel X executes alone and does not suffer anymore from interference. Moreover, in our naive method, the kernels with a latency shorter or equal than the second largest one suffer a maximum interference as they would never have a time interval where they execute alone on the GPU. This behaviour is exhibited in Figure 12, where DXTC is the only kernel with a time interval without interference.

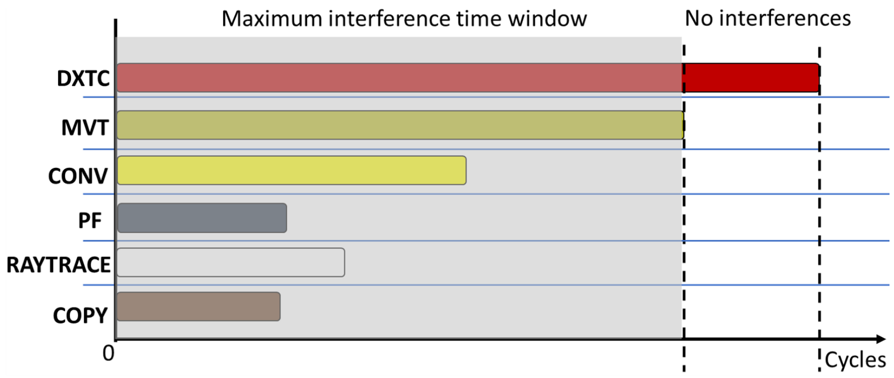


Fig. 11: A set of concurrent kernels with the maximum interference time interval in the light grey overlay.

In Figure 9, we observe that the kernel COPY from experiment 4 suffered a maximal slowdown of 73%. As an arbitrary value, the naive approach uses that value on all kernels fully present and on the portion of execution present in the maximum interference time window. We note that our 73% is a lower margin than the one presented in [28] (over 100%), however, this plays against us, as we expect that a larger margin for a naive approach generates a larger error.

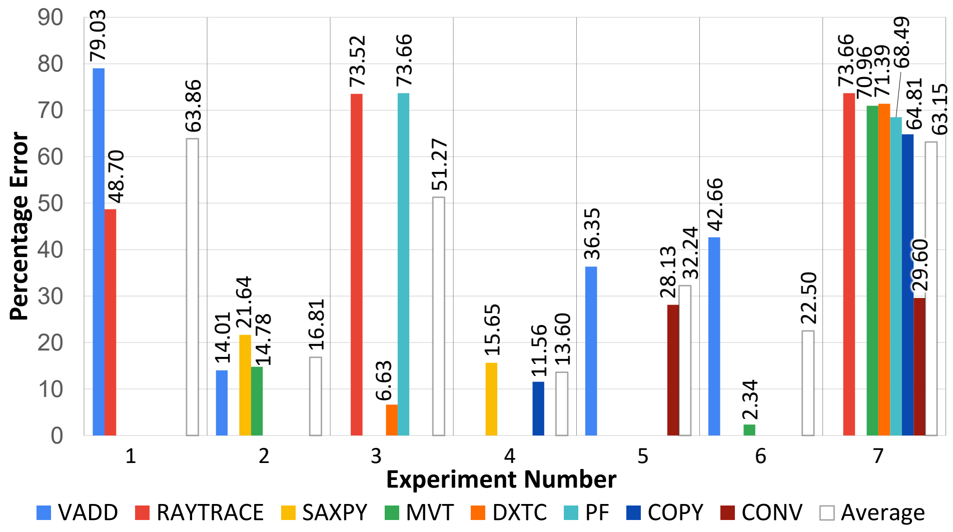


Fig. 12: Naive latency estimation, kernel cycles increased by a factor of 73%.

In Figures 12 and 13, the x-axis identifies the considered experiments from Table 3, and the y-axis represents the error percentage. A positive error percentage means an overestimation, while a negative

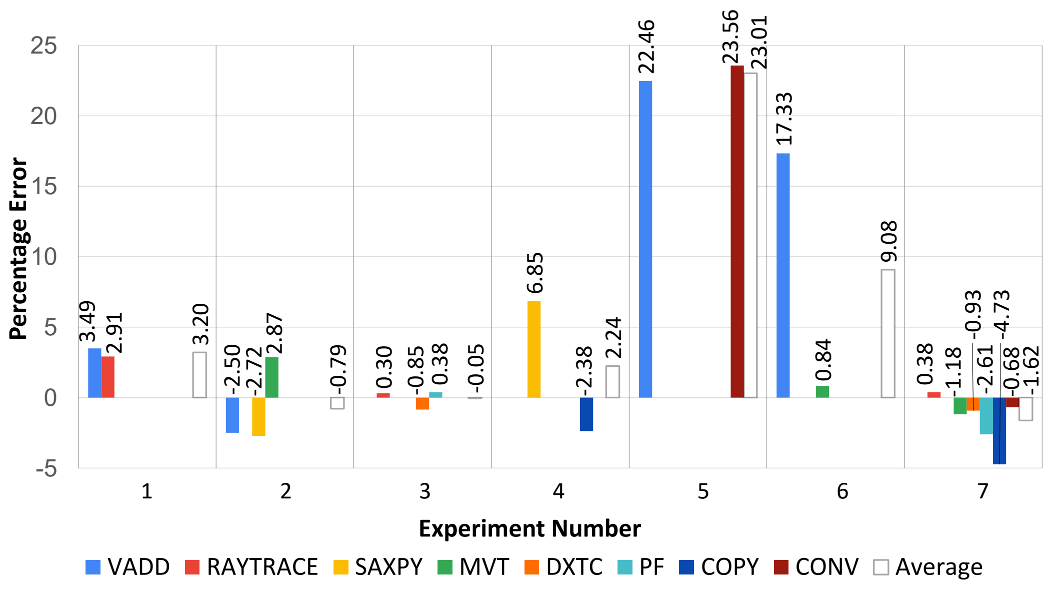


Fig. 13: Kernel latency estimation prediction error, using eq. (8), on experiments from Table 3.

one means an underestimation. The naive method results only in overestimation, while our method is more mitigated. However, the average error of the naive method is 42.38%, in contrast, our predictive model outperforms it with an absolute average error of 5%.

7.3 Latency and Interference Prediction Evaluations

To validate our predictive model and estimate its error, we randomly generated 100 groups of kernels, in addition to the 7 previous, with a size varying in $[2, 6]$, resulting in 107 experiments. In each group, for each kernel, we randomly decide the number of used SMs in the set 5, 10, 15, 20, 25, with the constraints that the total number of used SM for the group must not exceed the total number of available SMs (30).

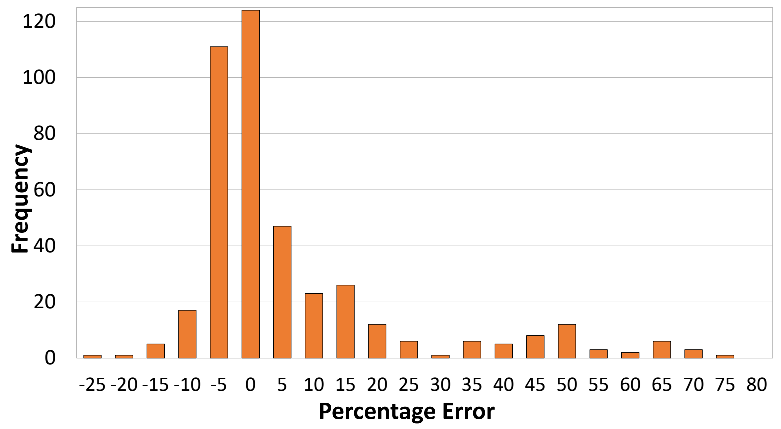


Fig. 14: Kernel latency prediction error, using eq. (8), of the 107 kernel sets.

Figure 14 shows the error distributions of our predictive mode in equation (8). In order to derive general and useful information about the distribution of errors, we considered the error calculated from the difference between the emulated cycle and the one estimated by our formula. On Overall, the formula demonstrated an average error rate of 10.7% with an overestimation bias and a variance

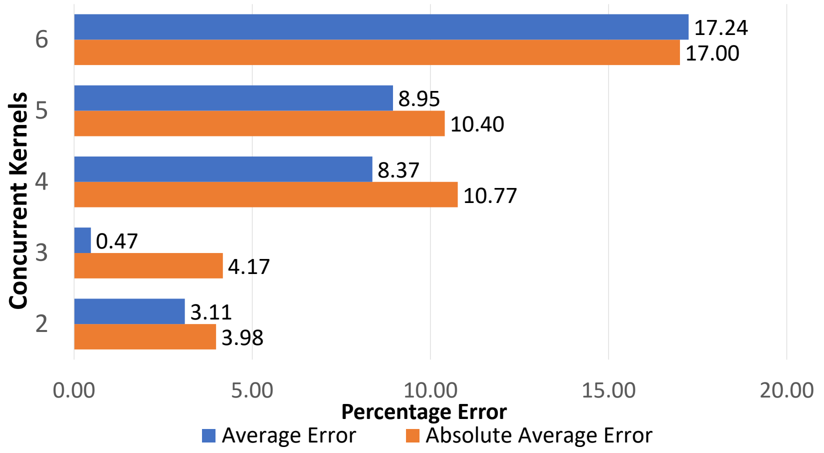


Fig. 15: Absolute and average kernel latency prediction error per number of concurrent kernels using eq. (8). All averages are positive, due to the weight of the overestimation.

of 2.71%. As shown in Figure 14, sporadic underestimations (negative values) and overestimations (positive values) occur during the prediction process. It is important to note that in the real-time field, particularly in the scheduling field where timing is critical, it is more favorable to overestimate rather than underestimate the execution time required by a kernel to perform its tasks. The results of our experiments, in which 2, 3, 4, 5, or 6 kernels were executed concurrently, resulted in a total of 420 single kernel predictions with 135 instances of underestimation and 285 instances of overestimation. The maximum sporadic overestimation was 76%, which occurred in 1 case out of 420, and the minimum sporadic underestimation was 20%, which also occurred in 1 case out of 420, or about 0.2%.

By also analyzing the individual different sub-classes error obtained in Figure 15, we can recover more information. Based on the normal average (not the absolute one), the obtained formula overestimated the whole experiment's classes (Figure 15). On average, in the sub-classes with 2 and 3 concurrent kernels, the formula overestimated the execution of each kernel by 3.11% and 0.47% respectively, with also an absolute average error of 3.98% and 4.17% respectively, indicating a high level of accuracy in predicting the cycles compared to the emulation engine. For the two sub-classes of 4 and 5 kernels, the formula overestimates on average, 8.37% and 8.95% respectively, meanwhile for the last sub-class with 6 kernels, we obtain the larger error committed of 17.24%. These last 3 sub-classes cases turn out to be the most unstable ones because more kernels require access to the same L2 banks and thus generate more interference than expected by our formula.

The results of the predictive model on different sub-classes remain well below the absolute error of 20% used in various domains for predicting the WCET, as already mentioned in Section 6, indicating excellent performance.

8 Bandwidth Prediction

In the previous sections, we successfully developed a predictive model that is able to predict the number of execution cycles required for the completion of a single kernel in both isolated and concurrent execution scenarios. To fully complete our model, we now aim to accurately predict the memory BW required by a kernel mapped on a set of SMs in isolation from an initial profile on all SMs. Hence, preventing the need to experimentally acquire the memory BW by repeatedly executing a kernel on different SM cluster sizes.

To achieve this goal, we model the BW *measured* during the isolated execution with the full set of SMs as $BW_{isolated}(k, N)$, and the $BW_{threshold}$ as $BW_{MAX} * S_p$ (defined in Section 6). By utilizing the $BW_{threshold}$, we are able to switch between linear and nonlinear forms in order to adaptively change the predictor behaviour. We define the $BW(k, n)$ as the BW *requirement* for a kernel k when it is executed on a different set of SMs (n), as outlined in eq. (17).

$$BW(k,n) = \begin{cases} \frac{BW_{isolated}(k,N)}{N} \cdot n & BW_{isolated}(k,N) < BW_{Threshold} \\ EM_{BW} \cdot (1 - e^{(-\frac{n}{MAX(1,Miss_{banks})})}) & BW_{isolated}(k,N) \geq BW_{Threshold} \end{cases} \tag{17}$$

The EM_{BW}, with the value of 330 GB/s, is the maximum bandwidth that we measured during our initial phase analysis in Section 5.1, meanwhile, the $Miss_{banks}$ are defined as the difference between the N SMs available and the number of total L2 banks defined by the hardware architecture. In this part of the study, we used the same configuration present in Table 1, where the L2 cache is with 24 banks, hence, the $Miss_{banks}$ is set to 6. With this exponent, we tried to model the hardware bottleneck in which each SM does not have a separate L2 bank to work without interference generated by the other SMs.

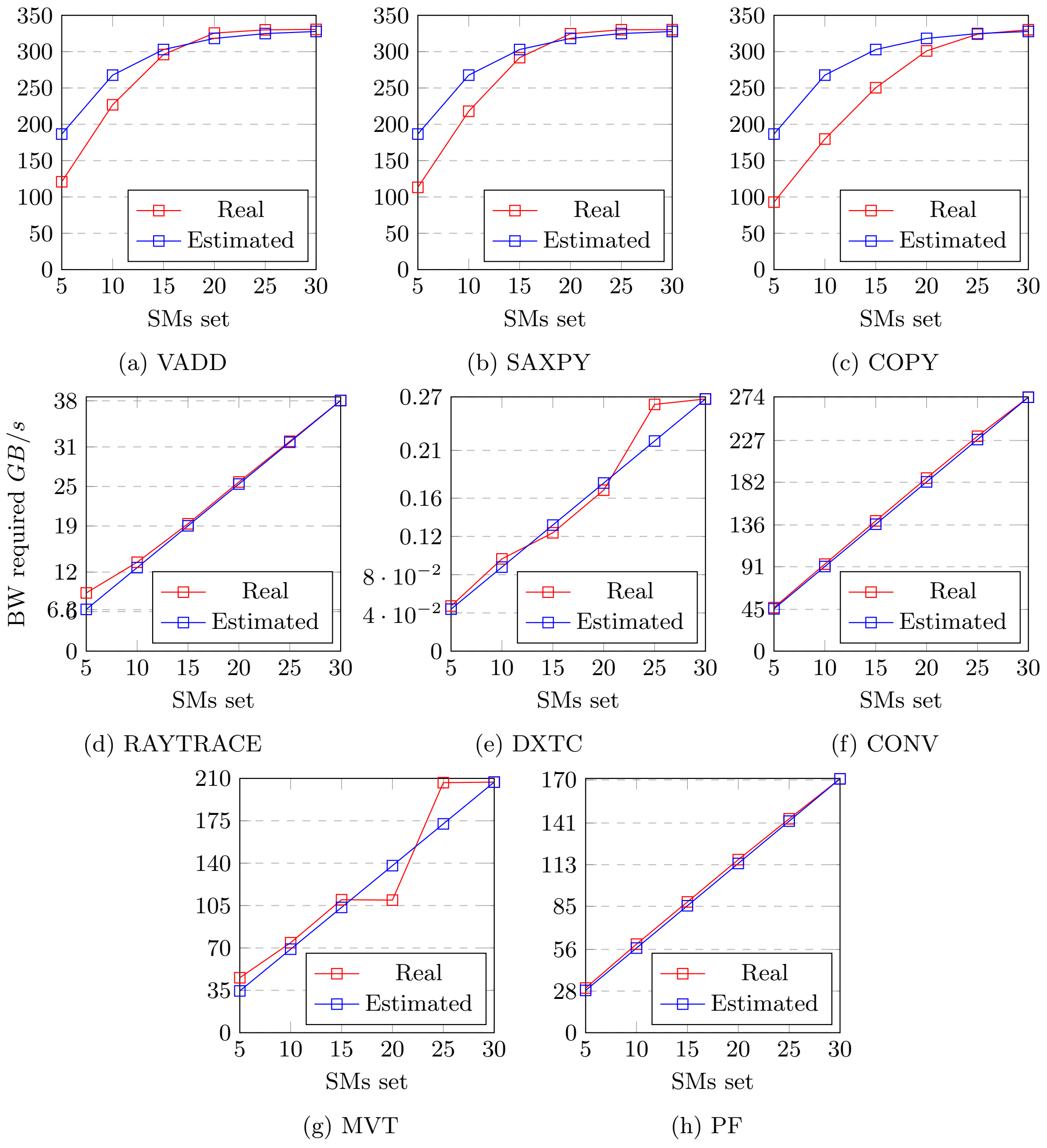


Fig. 16: Predicted BW for each kernel.

8.1 Bandwidth Prediction Results

The overall error obtained through Equation 17 is 4.60% on average and 11.23% using the absolute average and the results of the L2 cache BW prediction are presented in Figure 16, along with the

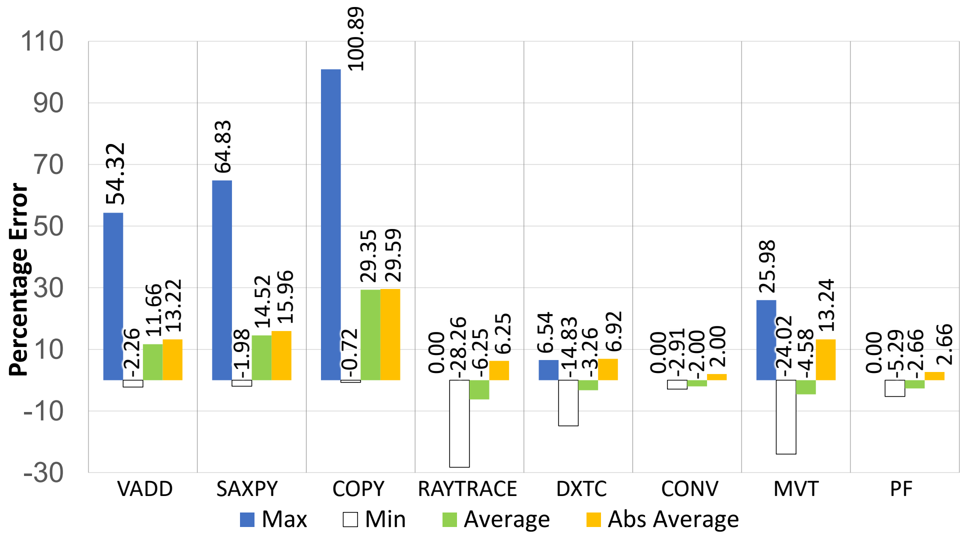


Fig. 17: Min, max, avg, and absolute avg BW prediction error, using eq. (17).

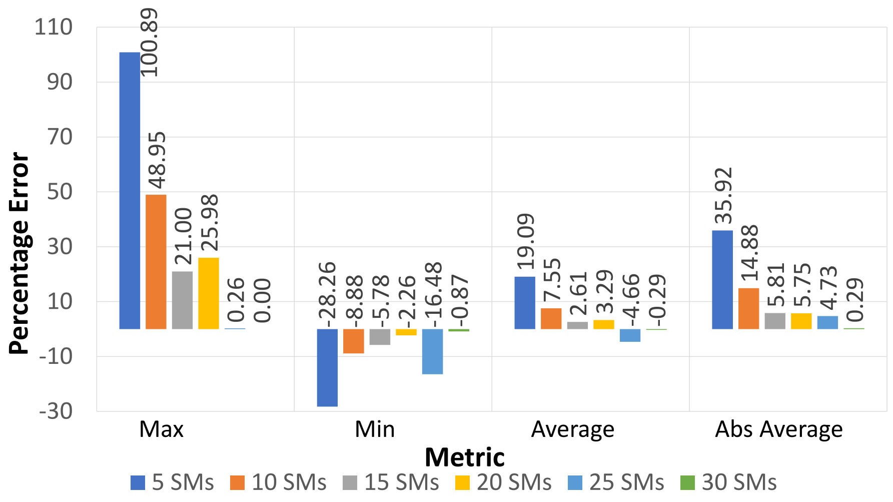


Fig. 18: BW avg and variance prediction error of eq. (17) varying the SMs size.

associated errors in Figure 17. Our approach demonstrates an overestimation of the L2 BW requirement for memory-intensive kernels such as VADD, SAXPY, and COPY when a smaller set of SMs is used. In contrast, the BW demand predictions for hybrid and computational kernels are shown to be highly accurate with our formula; nevertheless, underestimations still occur.

A summary of the overall BW prediction error average and variance is presented in Figure 18, along with the error obtained for individual SMs set (ranging from 5 to 30). On average, the error ranges from 0.3% to 19% when smaller SMs sets are chosen for kernel execution. The highest error prediction occurs when the SMs set is 5, with a prediction error of 19%.

9 Related Work

Not so many works found in the literature explicitly deal with performance estimation based on memory interference. The most recent one is HSM [29], in which performance decrease in co-running kernels scenarios are calculated as the ratio between instruction per cycle in isolated and shared modes in the different kernel-SM partition configurations. Another important work on this topic is the DASE (Dynamical Application Slowdown Estimation) model proposed in [10]. The authors of this work estimate the slowdowns caused by shared resources on concurrent GPGPU kernels and provide a performance estimation model for a single GPGPU application. Similarly to our work, their contribution involves a profiling phase, in which they consider cases where two kernels run concurrently. Our research takes into account different sets of concurrent kernels, potentially unlimited (tested up to six co-runners). Another related work is presented by Hong et al. in [9], in which they propose a model to estimate the execution cycles for the GPU architecture for a single kernel. They model the memory BW differently, by assuming that it is equally distributed among active SMs at runtime. The presented and other works highlighted that this is not always the case as BW distribution unfairness might occur depending on different key metrics that characterize the concurrent kernels [12]. Moreover, the concept of unfairness caused by the memory intensity of concurrent kernels is also present in the work of Adwait Jog et al. in [11], where they analyze the percentage BW utilization of different applications. Our work highlights the importance of the BW utilization term in predicting memory interference and defines different terms to model it. In the work of Hongwen Dai et al. [7], they demonstrate the benefits of balancing memory access between two kernels that are respectively memory-intensive and compute-intensive by limiting the number of memory instructions from two concurrent kernels to achieve fair memory access requests and avoid memory pipeline stalls. Another key aspect highlighted in this work, which is also present in our research in Section 7, is that a memory-intensive kernel may dominate the usage of the memory pipeline and execution in an SM, leading to significant performance loss for a compute-intensive kernel whose memory requests suffer delays and cannot be timely served. Other researchers have also proposed hardware schemes and methods to better exploit concurrent kernel execution on GPUs. For example, Xu et al. [27] proposed an intra-SM slicing approach to exploit underutilized memory throughput with an improved intra-SM partitioning for the CTA of two different kernels. Additionally, Thomas et al. used the roofline model to classify applications as compute-bound or memory-bound based on their operational intensity and applied different CTA scheduling modes based on the task category and core partitioning to improve performance [22]. In this work, they also highlighted the potential performance degradation when memory-bound tasks were assigned to small SM partitions. We proposed a similar analysis by predicting and measuring the memory BW required by a kernel mapped to different GPU partitions. Adriaens et al. [1] proposed the use of spatial multitasking to group SMs into different sets that can run different kernels (up to four) in order to maximize application speedup. Ukidave et al. [23] studied the real-time support for adaptive spatial partitioning on GPUs and highlighted the importance of L2 in this process. Aguilera et al. [2] demonstrated the unfairness of spatial multitasking and proposed a fair resource allocation strategy for both performance and fairness. Emir C. Marangoz et al. [15] showed that co-running applications can have interference in L2 cache, as we also highlighted in our work, and proposed a new architecture for managing BW reservation for a single kernel during concurrent execution. Unlike most of the research efforts we summarized in this section, our goal was to provide a predictive model for memory interference in cases of GPU SM spatial partitioning. This model should then be used by a memory-aware scheduler [4] that can dynamically assign the correct SM partition to each kernel, ensuring that each kernel can meet its temporal constraints. The sequence of operations that we present for modelling the effect of interference in shared memory hierarchies can be adapted to any GPU architecture and, unlike many of the previously cited papers, does not impose constraints on the number of kernels running simultaneously on the GPU.

It is also worth mentioning the recently released NVIDIA MiG (Multi Instance GPU) technology, that is able to provide stronger isolation among GPU partitions, by allocating cache partitions. While this approach is very effective to avoid memory interference, it is only available in only a subset of highend hardware. Moreover, limitations on the dinamicity in which we can define partitions are present.

10 Conclusion

In this paper, we conducted a thorough examination of the performance of inter-SM partitioning of concurrent kernels using GPGPU-Sim. More specifically, we modified GPGPU-Sim in order to perform an analysis of the interference caused by concurrent kernel execution in a multi-kernel partitioning scenario. Based on the number of concurrent kernels, the cycles required for each kernel during overlap with other kernels, and the single BW required by each kernel, we developed a methodology able to predict kernel performance deterioration in the presence of memory interference caused by L2 cache sharing. In order to test the validity of our proposed approach, an extensive set of experiments has been set up. The results of these experiments indicate that, on average, our predictive model overestimated the number of cycles required during concurrent kernel executions by 10.7% (absolute value) with a variance of 2.71%. Furthermore, when 2 or 3 kernels were competing on the GPU, our formula overestimated the cycles required by 3.11% and 0.47% respectively. Additionally, we developed a formula to predict the L2 cache BW demand by a single kernel in order to have prior knowledge of the BW requirements for individual kernels in relation to the SMs that will be assigned for their execution. Our prediction formula of the BW demand for a kernel when a smaller set of SMs is used yielded an average error range of 0.3% to 35% on different SMs sets and a total average error of 4% and 11% of absolute average error. System engineers may use our modified version of GPGPU-Sim to conduct an initial analysis of the kernels under examination, and gain insights into how the cycles required for completion and BW demand vary as the number of allocated Streaming Multiprocessors changes in an isolated scenario. By doing so, then the predictive models we presented in this paper can be easily adapted for different hardware architectures, i.e. GPUs with a different SM count and memory bank configurations. When dealing with additional kernels – that were not involved in the model construction phase – once an interference predictive model that is specific of the utilized HW is found, only a minimal, but necessary, set of profiling actions are required to be taken. These profiling phases involve measuring individual kernel performance in isolation and with no SM partitioning, as only measures about execution time, BW and compute-to-instruction ratio are needed. It is important to notice how such measures can also be easily extracted on real hardware. In the future, the developed predictive formulas will be further tailored to different types of HW architecture, including the new Nvidia RTX 3000 series. We also plan to use our predictive model to extend state-of-the-art task models in real-time literature with the purpose of accounting for memory interference among GPU kernels.

Acknowledgment

This work was partly supported by ECSEL Joint Undertaking in H2020 project IMOCO4.E, grant agreement No.101007311.

References

1. Adriaens, J.T., Compton, K., Kim, N.S., Schulte, M.J.: The case for gpgpu spatial multitasking. In: IEEE International Symposium on High-Performance Comp Architecture. pp. 1–12. IEEE (2012)
2. Aguilera, P., Morrow, K., Kim, N.S.: Fair share: Allocation of gpu resources for both performance and fairness. In: 2014 IEEE 32nd International Conference on Computer Design (ICCD). pp. 440–447. IEEE (2014)
3. et al., A.K.: Tango: A deep neural network benchmark suite for various accelerators. CoRR **abs/1901.04987** (2019)
4. Bak, S., Yao, G., Pellizzoni, R., Caccamo, M.: Memory-aware scheduling of multicore task sets for real-time systems. In: 2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications. pp. 300–309. IEEE (2012)
5. Capodieci, N., Cavicchioli, R., Marongiu, A.: A taxonomy of modern gpgpu programming methods: On the benefits of a unified specification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (2021)
6. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S.H., Skadron, K.: Rodinia: A benchmark suite for heterogeneous computing. In: 2009 IEEE international symposium on workload characterization (IISWC). pp. 44–54. Ieee (2009)
7. Dai, H., Lin, Z., Li, C., Zhao, C., Wang, F., Zheng, N., Zhou, H.: Accelerate gpu concurrent kernel execution by mitigating memory pipeline stalls. In: 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA). pp. 208–220. IEEE (2018)
8. Gupta, K., Stuart, J.A., Owens, J.D.: A study of persistent threads style gpu programming for gpgpu workloads. In: 2012 Innovative Parallel Computing (InPar). pp. 1–14 (2012). https://doi.org/10.1109/InPar.2012.6339596
9. Hong, S., Kim, H.: An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In: Proceedings of the 36th annual international symposium on Computer architecture. pp. 152–163 (2009)
10. Hu, Q., Shu, J., Fan, J., Lu, Y.: Run-time performance estimation and fairness-oriented scheduling policy for concurrent gpgpu applications. In: 2016 45th International Conference on Parallel Processing (ICPP). pp. 57–66. IEEE (2016)

11. Jog, A., Bolotin, E., Guz, Z., Parker, M., Keckler, S.W., Kandemir, M.T., Das, C.R.: Application-aware memory system for fair and efficient execution of concurrent gpgpu applications. In: Proceedings of workshop on general purpose processing using GPUs. pp. 1–8 (2014)
12. Jog, A., Kayiran, O., Kesten, T., Pattnaik, A., Bolotin, E., Chatterjee, N., Keckler, S.W., Kandemir, M.T., Das, C.R.: Anatomy of gpu memory system for multi-application execution. In: Proceedings of the 2015 International Symposium on Memory Systems. pp. 223–234 (2015)
13. Khairy, M., Shen, Z., Aamodt, T.M., Rogers, T.G.: Accel-sim: An extensible simulation framework for validated gpu modeling. In: 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA). pp. 473–486. IEEE (2020)
14. Konstantinidis, E., Cotronis, Y.: A quantitative roofline model for gpu kernel performance estimation using micro-benchmarks and hardware metric profiling. Journal of Parallel and Distributed Computing **107**, 37–56 (2017)
15. Marangoz, E.C., Kang, K.D., Shin, S.: Designing gpu architecture for memory bandwidth reservation. In: 2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). pp. 87–89. IEEE (2021)
16. Navarro, C.A., Hitschfeld-Kahler, N., Mateu, L.: A survey on parallel computing and its applications in data-parallel problems using gpu architectures. Communications in Computational Physics **15**(2), 285–329 (2014)
17. Olmedo, I.S., Capodieci, N., Martinez, J.L., Marongiu, A., Bertogna, M.: Dissecting the cuda scheduling hierarchy: a performance and predictability perspective. In: 2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS). pp. 213–225. IEEE (2020)
18. Pai, S.e.a.: Improving GPGPU concurrency with elastic kernels. In: Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems. p. 407–418. ASPLOS '13, Association for Computing Machinery, New York, NY, USA (2013)
19. Rouxel, B., Skalistis, S., Derrien, S., Puaut, I.: Hiding communication delays in contention-free execution for spm-based multi-core architectures. In: 31st Euromicro Conference on Real-Time Systems (ECRTS 2019). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2019)
20. Silva, K.P., Arcaro, L.F., De Oliveira, R.S.: On using gev or gumbel models when applying evt for probabilistic wcet estimation. In: 2017 IEEE Real-Time Systems Symposium (RTSS). pp. 220–230. IEEE (2017)
21. Sun, Y., Agostini, N.B., Dong, S., Kaeli, D.: Summarizing cpu and gpu design trends with product data. arXiv preprint arXiv:1911.11313 (2019)
22. Thomas, W., Toraskar, S., Singh, V.: Dynamic optimizations in gpu using roofline model. In: 2021 IEEE International Symposium on Circuits and Systems (ISCAS). pp. 1–5. IEEE (2021)
23. Ukidave, Y., Kalra, C., Kaeli, D., Mistry, P., Schaa, D.: Runtime support for adaptive spatial partitioning and inter-kernel communication on gpus. In: 2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing. pp. 168–175. IEEE (2014)
24. Wang, Z., Yang, J., Melhem, R., Childers, B., Zhang, Y., Guo, M.: Simultaneous multikernel gpu: Multi-tasking throughput processors via fine-grained sharing. In: 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA). pp. 358–369. IEEE (2016)
25. Wartel, F., Kosmidis, L., Gogonel, A., Baldovino, A., Stephenson, Z., Triquet, B., Quinones, E., Lo, C., Mezzetta, E., Broster, I., et al.: Timing analysis of an avionics case study on complex hardware/software platforms. In: 2015 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 397–402. IEEE (2015)
26. Williams, S., Waterman, A., Patterson, D.: Roofline: an insightful visual performance model for multicore architectures. Communications of the ACM **52**(4), 65–76 (2009)
27. Xu, Q., Jeon, H., Kim, K., Ro, W.W., Annavaram, M.: Warped-slicer: efficient intra-sm slicing through dynamic resource partitioning for gpu multiprogramming. In: 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA). pp. 230–242. IEEE (2016)
28. Yandrofski, T., Chen, J., Otterness, N., Anderson, J.H., Smith, F.D.: Making powerful enemies on nvidia gpus. In: 2022 IEEE Real-Time Systems Symposium (RTSS). pp. 383–395. IEEE (2022)
29. Zhao, X., Jahre, M., Eeckhout, L.: Hsm: A hybrid slowdown model for multitasking gpus. In: Proceedings of the twenty-fifth international conference on architectural support for programming languages and operating systems. pp. 1371–1385 (2020)