



UNIVERSITÀ DI PARMA

UNIVERSITA' DEGLI STUDI DI PARMA

DOTTORATO DI RICERCA IN
"MATEMATICA"

CICLO XXXVI

Memory Interference and Performance Prediction
in GPU-Accelerated Heterogeneous Systems

Coordinatore:
Chiar.mo Prof. Leonardo Biliotti

Tutore:
Chiar.mo Prof. Nicola Capodieci

Dottorando: Alessio Masola

Anni Accademici 2020/2021 - 2022/2023

Abstract

Nowadays, a variety of applications, including automated factories, autonomous vehicles, and Cyber Physical Systems (CPS), are experiencing significant growth. Given the diverse range of challenges that must be addressed, such as real-time management and visualization of a factory's current state through a 3D digital twin, trajectory calculation within autonomous vehicles, and visualizing Human Machine Interfaces (HMI), as well as traffic management in smart cities equipped with cameras, IoT devices, and their associated features. In response, a broad array of *heterogeneous devices* with various *hardware accelerators* are being utilized across these different scenarios. In such applications, *power consumption* and *task execution latency* are key aspects to consider, hence investigating approaches that mitigate power consumption while still fully utilizing the computational power provided by the devices becomes necessary.

Modern devices use hardware processors that enable the acceleration of highly parallel and data hungry computational workloads; a widely known example of such parallel processor is the *Graphic Process Unit* (GPU), a hardware peripheral traditionally used for graphics rendering but nowadays it is also used as a general purpose compute accelerator.

This thesis addresses an analysis of the state of the art of techniques that can be employed to *optimize power consumption* and task execution latency, as well as two types of latencies/interference that tasks can potentially experience: latencies arising from tasks that are concurrently scheduled on the same acceleration unit, i.e., on a partitioned GPU, and the second type under consideration is the latencies experienced by tasks running on embedded boards, specifically on GPU-embedded systems, with a high computational load on the CPU side. Methods are proposed to understand and derive predictive models for latencies in both of the two types of interference. Furthermore, this thesis concludes with a comparative study of two GPU memory management methodologies: explicit copies versus unified virtual memory.

Contents

1	Introduction	7
1.1	Motivation	7
1.2	Scopes	8
1.3	Methodologies	9
2	Optimizing Energy Efficiency and Performance in GPUs for IoT and Embedded Systems	11
2.1	Introduction	11
2.2	GPU: an overview	12
2.2.1	Hardware GPU architecture	12
2.2.2	GPU caches in depth	13
2.2.3	Last Level Cache replacement policies	15
2.2.4	GPU workloads	17
2.2.5	Memory Access Optimization	18
2.2.6	Compute Optimization	21
2.2.7	Ray Tracing and memoization	22
2.2.8	Conclusion	25
3	Memory-Aware Latency Prediction Model for Concurrent Kernels in Partitionable GPUs	27
3.1	Introduction	27
3.1.1	GPU Programming Model and Scheduling	28
3.1.2	Cycle Accurate Simulation Through GPGPU-Sim	29
3.2	Methodology Overview	30
3.3	Simulation Settings and the Implemented GPGPU-Sim Extension	31
3.4	Memory Aware Performance Estimation	32
3.4.1	Kernels Memory Bandwidth Analysis	33
3.4.2	Kernels Completion Latency Analysis	34
3.5	Predicting Latencies Depending on the Assigned SMs	35
3.6	Modeling Memory Interference	38
3.6.1	Experiments and Analysis	39
3.6.2	Comparison with a Worst-interference Method	43
3.6.3	Latency and Interference Prediction Evaluations	44
3.7	Bandwidth Prediction	46
3.7.1	Bandwidth Prediction Results	47
3.8	Related Work on Estimation of Malleable Tasks	47
3.9	Conclusion	51

4	Machine Learning Techniques for Understanding and Predicting Memory Interference in CPU-GPU Embedded Systems	53
4.1	Introduction	53
4.2	State of the Art	54
4.3	Background	55
4.3.1	Nvidia Jetson AGX Xavier Architecture	55
4.3.2	The profiling tool - NCU	56
4.4	Setup and First Analysis	57
4.4.1	ncu and Kernels' Performance Metrics	57
4.4.2	Experimental Setup	59
4.5	Methodologies for Slowdown Prediction	60
4.5.1	Training and Test	60
4.5.2	PARKERNEL: A Parameterized Kernel	61
4.5.3	Methodologies and Results	63
4.6	Metrics Analysis	65
4.7	Conclusion	66
5	Dissecting Memory Explicit Copy vs Unified Virtual Memory	69
5.1	Introduction	69
5.2	Memory Management in CUDA	69
5.3	Analysis	72
5.4	PCA and Spearman Coefficient	74
5.4.1	Conclusion	76
6	Thesis Conclusion	83
6.1	Future Work	85

Acknowledgements

English version:

I would like to express my deepest gratitude to Professor Nicola Capodieci, an unwavering guide and an endless source of inspiration. His dedication during the three intense years of my doctoral studies paved the way for my research, teaching me the complexity of the scientific world with patience and determination. His constant support and wise guidance made this academic journey possible.

A heartfelt thank you goes to my esteemed colleagues at the University of Modena. Their understanding and ability to share diverse perspectives lightened the obstacles and enriched my educational path. Thank you for enduring my frustrations and providing an enlightening view on life and work challenges.

To my dear parents, I want to dedicate a heartfelt and affectionate thank you. Your sacrifices and unwavering encouragement propelled me to pursue the academic path without ever giving up in the face of challenges. Thank you from the bottom of my heart, Mom and Dad, for your continuous unconditional support, both in academia and life.

A special thanks goes to "my" Maria, life partner, and confidante. Her constant support during the dance of life has been a guiding light, offering me new perspectives and encouraging me to take on challenges at every turn. Thank you, my love, for your dedication and unconditional understanding.

I cannot forget my dear friends, who share joys and challenges in everyday life. Your conversations, board games, and constant presence have continually filled the jar of my essence. Thank you from the bottom of my heart for enriching my experience with your friendship and support.

Italian version:

Desidero esprimere la più profonda gratitudine al Professor Nicola Capodieci, guida instancabile e fonte inesauribile di ispirazione. La sua dedizione durante i tre intensi anni di dottorato ha spianato il percorso della mia ricerca, insegnandomi la complessità del mondo scientifico con pazienza e determinazione. Il suo supporto costante e le sagge indicazioni hanno reso possibile questo viaggio accademico.

Un caloroso ringraziamento va ai miei stimati colleghi presso l'Università di Modena. La loro comprensione e la capacità di condividere prospettive diverse hanno reso più leggeri gli ostacoli e arricchito il mio percorso formativo. Grazie per aver sopportato le mie frustrazioni e per aver fornito un'illuminante visione sui problemi di vita e di lavoro.

Ai miei cari Genitori, voglio dedicare un sentito ringraziamento con affetto. I vostri sacrifici e il vostro incoraggiamento instancabile mi hanno spinto a perseguire il cammino accademico senza mai arrendermi di fronte alle sfide. Grazie di cuore, Mamma e Papà, per il vostro incondizionato sostegno continuo anche per la vita.

Un grazie speciale va alla "mia" Maria, compagna di vita e confidente. Il suo appoggio costante durante il valzer della vita è stato un faro luminoso, offrendomi nuove prospettive e incoraggiandomi a mettermi in gioco in ogni fase. Grazie amore, per la tua dedizione e

comprensione incondizionata.

Non posso dimenticare i miei cari amici, che condividono gioie e sfide nella vita di tutti i giorni. Le vostre chiacchierate, i giochi da tavolo e la vostra presenza costante hanno riempito la giara della mia essenza. Grazie di cuore per arricchire la mia esperienza con la vostra amicizia e sostegno.

Chapter 1

Introduction

1.1 Motivation

Nowadays, various devices come equipped with chips capable of performing different types of processing while communicating with cloud and/or fog infrastructures, as discussed in Larrucea’s work [1]. These devices, which include wearables, handheld devices, embedded systems like smartphones, computers, laptops, servers, smartwatches, and autonomous vehicle (AV) control units (CU), often have constraints related to energy consumption.

Additionally, it is important to note that these devices can also exchange information and collaborate for computations. For instance, smartphones run multiple applications concurrently, such as 3D avatar rendering, AR (Augmented Reality) measurements via the camera, web browsing, messaging, and calls. Many of these tasks can potentially overlap in time, and if energy and latency requirements are not met, the device may freeze or quickly deplete its battery.

To address these challenges, these devices are equipped with System on Chip (SoC) solutions that integrate specialized processing engines. These engines are designed for specific tasks, like processing neural networks, handling network communications, and interacting with other IoT (Internet of Things) devices. Additionally, these SoCs may include programmable massively parallel accelerators like Graphic Processing Units (GPUs). GPUs represent an energy-efficient solution for certain tasks [2], but there is still the need to balance low latency and energy efficiency. In the past, GPUs were primarily used for graphics rendering and were not common in integrated SoCs. However, today, they are prevalent in embedded platforms and are employed for general-purpose (GP) computing tasks.

Leveraging the power of GPUs for massively parallel tasks can lead to inefficiencies or under-utilization of computation hardware, resulting in unwanted latency and energy wastage. By optimizing the architecture to exploit recurring memory requests and by eliminating redundant computations, the execution time and energy consumption of these devices can be improved. In recent years, researchers have explored various optimization strategies to reduce or optimize memory accesses, a significant source of energy consumption in such accelerators [3, 4, 5, 6]. For example, GPUs can be installed externally to the embedded device, serving as a processing element accessible through cloud or edge computing [3, 4]. Alternatively, GPUs can be integrated into the IoT SoC to execute on-board processing for graphic workloads [5, 6]. The various types of workloads, from cloud to graphics workloads, exhibit significant algorithmic differences, necessitating distinct architectural optimizations.

In summary, there are various optimization strategies already studied by researchers, which are valuable for understanding the issues in these types of architectures. Unfortunately, in the literature, there is a lack of in-depth studies on the underutilized resources in GPUs. For example, the execution of a single task often does not fully exploit the architecture’s

potential (e.g., maximum memory bandwidth or available Streaming Multiprocessor). When it becomes possible to execute multiple tasks concurrently to fully utilize the capabilities of a partitionable GPU architecture, the challenge arises of how these concurrently executing tasks on the same architecture experience inter-task latencies. Additionally, there is the need to predict these inter-task latencies in advance so to appropriately design scheduling policies.

Furthermore, within the domain of embedded SoCs with integrated GPUs, there is a scarcity of comprehensive studies on how tasks behave when the overall computational and memory load of the embedded architecture varies. Moreover, there is a lack of understanding of how to exploit the observable GPU hardware counters for mitigating memory interference coming from different sources.

1.2 Scopes

There are several research goals that are addressed in this thesis. One of the highly studied research areas involves investigating various *methods and strategies to reduce and optimize memory accesses*, which also results in energy reduction across different architectures. Within this thesis, the first objective is to explore these diverse optimization strategies for minimizing or improving memory accesses. The first objective, addressed in the Chapter 2, aims to *gain a better understanding* of how different architectures work and intends to assist system engineers in improving performance and reducing energy consumption for embedded, and discrete GPUs. In the Chapter 2, I investigate both generic and graphic workloads, as contemporary applications increasingly incorporate a mix of these task types.

As a second objective, explored in Chapter 3, within the domain of GPUs, a system has been developed to address the underutilization of hardware resources in GPUs by enabling concurrent scheduling of multiple tasks on the same GPU. This concurrent executions leads to inter-task interference that cannot be underestimated. For instance, when focusing on a system with real-time requirements and strict deadlines for each individual task, it is imperative to consider this interference that could lead to the scheduling unfeasibility in real-time system. Consequently, a comprehensive analysis (in Section 3.4 and subsequent sections) of what occurs during these concurrent executions has been carried out. This analysis provides the opportunity to create predictive models capable of determining in advance which types of tasks to schedule concurrently, considering that such decisions are crucial in real-time and latency sensitive scenarios.

The third addressed objective focuses on heterogeneous embedded platforms. There is a knowledge gap regarding the state of the art studies in these architectures used in various latency-sensitive applications such as the automotive domain, IoT real-time systems (RTS), smart cities, and many others. Regarding the state of the art, a comprehensive analysis of the behavior of these accelerators under high computational load conditions is lacking, which is essential to understand how the parallel computational engines in these modern platforms may interfere and consequently impact performance and predictability. For instance, GPU-accelerated tasks may run concurrently with other CPU tasks, and this is a common occurrence in almost every application scenario mentioned earlier. This third objective is discussed in this thesis in Chapter 4.

The final objective of this thesis, as outlined in Chapter 5, delves into a comparative analysis of two distinct memory management methodologies within the CUDA API framework: direct or explicit memory management and Unified Virtual Memory (UVM). This investigation, presented as preliminary work, is particularly focused on their application in executing tasks on GPUs within heterogeneous architectures. Despite the significance of these memory management approaches, existing research in this domain is missing a comprehensive

examination of task-specific metrics and their potential impact on future predictive modeling. Consequently, there exists a notable gap in understanding how these metrics could contribute to a more nuanced evaluation and comparison of the two methodologies. By addressing this gap, this chapter seeks not only to provide a detailed exploration of UVM versus direct memory management but also to shed light on the role of task metrics in informing predictive models, thereby enhancing our understanding of memory management strategies in CUDA-based heterogeneous computing environments.

1.3 Methodologies

To achieve the first scope, an exploration on how these optimization methodologies and algorithms works on different architectures have been conducted in Chapter 2, in order to gain a better understanding of how different architectures work and intends to assist system engineers in improving performance and reducing energy consumption for embedded, integrated, and discrete GPUs. This thesis specifically investigates generic and graphic workloads, as contemporary applications increasingly incorporate a mix of these task types.

With regards to the unexplored underutilized resources in GPUs, a well-known GPU architecture simulator called GPGPU-Sim is employed. GPGPU-Sim has been modified, in Section 3.3, in order to simulate the concurrent execution of multiple tasks and to have a partitionable management of parallel compute clusters present in the GPU. This modification enables to perform different tests and allows the system engineer to capture information regarding the execution behavior, of tasks scheduled concurrently, on different sets of compute clusters. Following the initial analysis, in Section 3.6, a predictive model is developed to predict latency, i.e., slowdowns and bandwidth demand, that tasks might experience based on the type of task being scheduled and the number of compute clusters they will execute on.

For the knowledge gap of the behavior of the embedded accelerators under high computational load conditions, in Chapter 4, there will be an in-depth examination of the behavior of GPU-accelerated tasks as the CPU memory pressure load varies. Subsequently, Machine Learning (ML) techniques will be applied (Section 4.5) to develop a predictor capable of forecasting the slowdowns experienced by applications as the CPU-side memory pressure load varies. In Section 4.6 a comprehensive analysis will be carried out using various methodologies, including statistical methods and results derived from the employed ML techniques, on the profileable metrics of GPU kernels. I therefore make an effort to understand how those metrics relate to the behavior of an individual task during execution. By performing such an analysis, the most critical metrics for predicting GPU application slowdowns, when overlapping execution on memory intensive CPU jobs will be highlighted.

In Chapter 5, dedicated to heterogeneous architectures, two memory management approaches for task execution on GPUs are examined. This initial analysis lays the groundwork for developing new predictive models capable of forecasting the performance of both methodologies beforehand. Within the chapter, a preliminary analysis is conducted on the differences between direct copy directives and the use of the Unified Virtual Memory (UVM) on Nvidia platforms, with a focus on CUDA memory management as a starting point. The objective of the chapter is to evaluate the advantages and disadvantages of each strategy through various analysis techniques. This analysis is performed by profiling metrics related to the execution behavior of various tasks on different platforms, both embedded and discrete. The primary goal is to provide initial insights on the profiled metrics, currently in the development and refinement phase.

Finally, the thesis ends with a summary of key findings in Chapter 6

Chapter 2

Optimizing Energy Efficiency and Performance in GPUs for IoT and Embedded Systems

2.1 Introduction

The significant growth of industrialization and the Internet of Things (IoT) has led to the daily use of various types of devices in both professional and non-professional live. In general, devices including wearables, handheld gadgets, and embedded systems such as smartphones, computers, laptops, servers, smartwatches, and onboard systems in autonomous vehicles, may face constraints related to energy consumption. It is important to note that these devices can also communicate with each other while conducting computations[1].

For instance, consider a mobile device like a smartphone that runs applications enabling tasks such as facial recognition with 3D avatars, augmented reality measurements via the camera, web browsing, receiving messages, and handling calls. Many of these tasks can potentially overlap in their execution timelines. If the energy or latency requirements are not met, the device may experience freezing or complete battery depletion during such operations. These devices are equipped with a System on Chip (SoC) that integrates various specialized processing engines. These engines are designed for specific tasks, including modules for neural network processing, data transmission over networks and IoT devices, as well as programmable massively parallel accelerators like Graphics Processing Units (GPUs). While GPUs are known for their energy efficiency [2], there is still a need to address latency issues while concurrently striving to conserve energy. In the past, GPUs were primarily utilized for rendering in computer graphics and were not integrated into SoCs. However, in today's landscape, they are a common feature in many embedded platforms and are employed for General Purpose (GP) tasks, also referred to as compute tasks. The utilization of this powerful hardware type, designed for massively parallel tasks, can either result in inefficiency or underutilization of the hardware architecture. This could result in undesirable latency and wasted energy resources. Through comprehensive optimization of the architecture, including the potential exploitation of recurrent memory requests and the elimination of redundant computations, it is possible to improve the execution times and potentially also reduce energy consumption for these devices.

In recent years, researchers have focused on this specific field, developing optimization strategies to limit or enhance, with different techniques, the memory accesses, the primary source of energy consumption in such accelerators[7]. Within this context, in this first chapter, has been conducted a thorough survey and review of numerous approaches that empower

system engineers to improve performance and potentially, as a side effect, also the energy consumption for embedded, integrated, and discrete GPUs. Specifically, the investigation has encompassed both generic and graphics workloads. This is due to the fact that applications involve a combination of graphic and generic computational tasks.

There are various methods for harnessing the power of GPUs, such as utilizing cloud-based computing services or integrating GPUs into IoT SoCs for on-board processing. These tasks typically fall into two categories: computational [3, 4] or graphical [5, 6]. However, there are differences in algorithms and requirements between these two types of workloads, which necessitate distinct architectural optimizations.

In the following section, an overview of GPU operation is provided, delving into its hardware and software architecture. Subsequently, the discussion on the state-of-the-art GPU optimization strategies is presented in separate sections, one dedicated to computational workloads and the other to graphic applications. In each section, the significance of optimizations in the context of memory access improvement is underscored, and methods for circumventing redundant computations are explored. The work presented in this chapter is derived from my publication [8].

2.2 GPU: an overview

A description of the most common architectural blueprints from which GPU device vendors design these massively parallel accelerators is now provided. Beginning with this brief overview, the subsequent focus will be on the specific architectural aspects that recent research in GPU optimization strategies has emphasized.

2.2.1 Hardware GPU architecture

Although originally designed for graphics processing, modern GPUs have evolved into highly parallel compute accelerators suitable for various general-purpose applications [9]. In accordance with Flynn’s well-established computational architecture classification [10], a GPU can be categorized as a highly parallel computational accelerator positioned between the SIMT (Single Instruction Multiple Threads) and SIMD (Single Instruction Multiple Data) approaches. Within a GPU, numerous ALU (Arithmetic Logic Unit) pipelines cater to threads, executing instructions in look-step fashion.

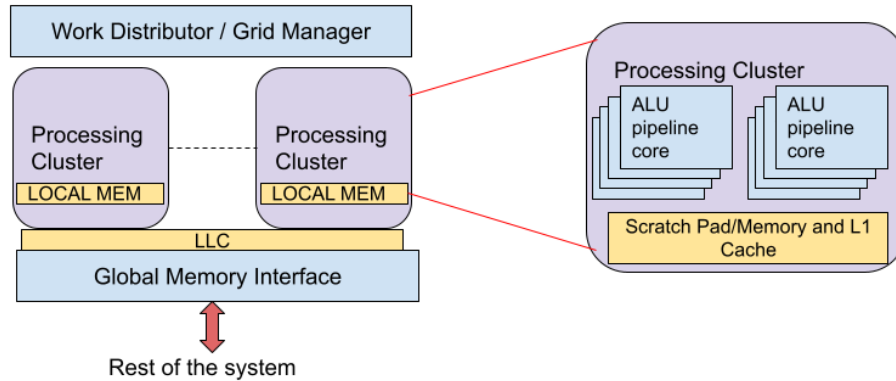


Figure 2.1: Overview of a modern GPU

In Figure 2.1, a general overview of a GPU is presented, with a focus on the programmable

aspect of the GPU. The ALU pipelines are organized into compute clusters, and each of these clusters relies on a memory hierarchy to access data. Specifically, each compute cluster contains internal and private memories, alongside an interface to shared cache levels. These shared cache levels, in contrast, are accessible to all compute clusters and facilitate explicit data sharing. The internal memories, such as L1 cache and scratch-pad memories, are primarily utilized for communication and synchronization among threads dispatched within the same processing cluster. On the other hand, the shared cache subsystem not only enables threads from different compute clusters to share data but also serves as the interface through which the GPU connects with the rest of the system.

Typically, in GPUs, each processing cluster facilitates high-bandwidth and low-latency memory access through a scratch-pad memory (also referred to as shared memory) and L1 cache [11]. Optimizing memory access in such a manner is contingent on the parallel threads within a GPU being able to coalesce their memory requests. Coalescing memory requests entails parallel threads retrieving contiguous data accesses, allowing these requests to be bundled together within a single cache line access.

However, achieving coalescence of memory accesses can be challenging or might be impossible depending on the characteristics of the workloads, particularly when memory access patterns are irregular or sparse [12].

A GPU can be connected to the rest of the system in two primary ways: as an integrated GPU (iGPU), residing within the same chip as the CPU host, or as a discrete peripheral (dGPU). In the latter case, the GPU has the capability to transfer data between GPU-visible-only memory address spaces and CPU-only memory address spaces via a PCI-express connection. When optimizing GPU workloads, memory accesses are a critical consideration since each level within the GPU’s memory hierarchy exhibits significant differences in bandwidth and latencies during data access.

Furthermore, accessing global memory, such as VRAM in dGPUs or directly accessing system RAM in integrated systems on chips, can substantially escalate energy costs for the workload in question. Activation costs for RAM banks constitute a substantial portion of the power budget for IoT-powered mobile and embedded devices. Consequently, it comes as no surprise that the majority of research efforts in GPU optimization are strongly concentrated on comprehending these challenges and proposing innovative approaches to manage memory accesses effectively.

To emphasize the specific terminology associated with all the blocks in Figure 2.1, it’s essential to note the distinct nomenclature employed by various vendors. In NVIDIA architectures (Figure 2.2), the processing clusters are denoted as *Streaming Multiprocessors (SMs)*. On the other hand, recent ARM and Intel iGPUs utilize the term *Execution Unit (EU)*. In contrast, AMD refers to the collection of parallel ALUs within their GPUs as (*Next-Generation Compute Unit (CU/NCU)*).

2.2.2 GPU caches in depth

A GPU typically features three levels of data caches. Integrated SoCs usually include a third-level LLC (Last Level Cache). For instance, in recently released NVIDIA embedded GPUs, the L3 cache is partitioned between the CPU and the iGPU, primarily serving as a victim cache [13]. In Intel GPUs, the L3 cache functions as a general-purpose, low-latency memory within the overall coherence domain with the CPU host [14]. As a result, it can be dynamically configured for use as a shared cache among the computing clusters and/or the CPU. Application developers also have the ability to reconfigure a level of GPU cache in NVIDIA devices, albeit at a higher level within the cache hierarchy, specifically the L1 data cache. To elaborate, developers targeting NVIDIA hardware can opt to configure a variable

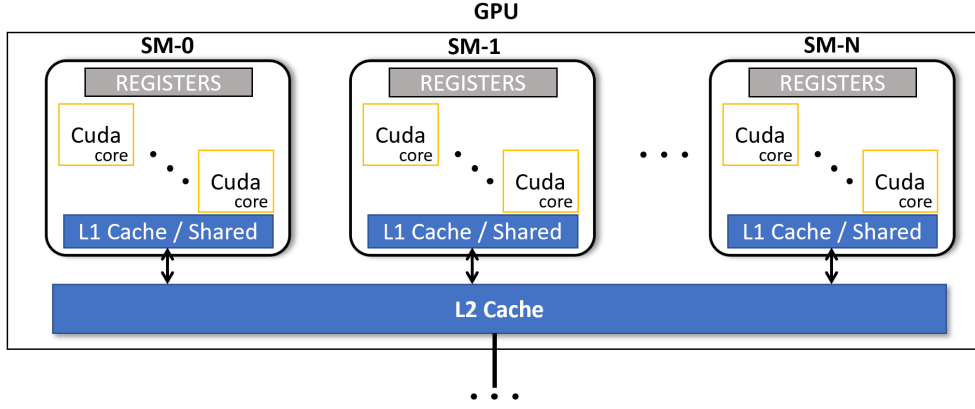


Figure 2.2: GPU architectures are composed of a set of SMs with L2 cache shared among each SM.

percentage of each compute cluster’s L1 cache as a scratch-pad memory. This means that a portion of the in-chip cache can be directly indexed, departing from the transparent manner in which hardware manages memory accesses through caching. From a programming perspective, relying solely on a larger L1 cache is undeniably more straightforward. In contrast, orchestrating memory accesses explicitly within the shared (scratch-pad) memory necessitates a more involved coding approach. In this context, Li et al. [15] conducted a study focusing on performance and power metrics within a limited set of simulated benchmarks. The objective was to comprehend the trade-off between hardware and software-managed memory accesses. Depending on the specific task at hand, judicious utilization of shared memory yields improved performance and significantly reduced power consumption. It is crucial to underscore that shared memory not only demands considerably more implementation effort but also necessitates careful dynamic sizing. This is because shared memory is allocated per compute cluster and is accessible to every block of threads executing in parallel, impacting what is termed as the *block occupancy factor* in the NVIDIA terminology. Consequently, inappropriately sized shared memory can restrict the number of parallel threads scheduled simultaneously within the same compute cluster [16].

In certain GPU designs, like older NVIDIA micro-architectures, the innermost cache level within a compute cluster underwent further segmentation into memory segments with specific functions. An illustrative example of this is the *texture cache*. The texture cache sets itself apart from the general-purpose L1 data cache due to its exclusive read-only access. It is highly optimized for specific access patterns and has been engineered for hardware-based manipulation of read operations, primarily catering to tasks like texture filtering and compression [17, 18].

Given these fundamental distinctions, researchers have delved into tailored optimizations for texture caches. Specifically, Hakura and Gupta [19] and Wong et al. [20] recognized how cache associativity in texture caches could exert a substantial impact on graphic application performance. While interested readers can explore the mentioned references for more detailed insights, it’s crucial to note that such optimizations may no longer hold relevance for nowadays hardware. In modern NVIDIA GPU architectures, the texture cache has been integrated into the L1 cache [21].

2.2.3 Last Level Cache replacement policies

Paulsson et al. provided a comprehensive examination of cache replacement policies concerning the GPU's Last Level Cache (LLC) in their work [22]. Their paper serves as an extensive survey of existing mechanisms, introduces innovative cache replacement policies, and conducts a thorough performance comparison among the surveyed mechanisms.

Here, an overview of the proposed policies are presented. First, the most straightforward and widely studied cache replacement policies are highlighted. These policies operate on mechanisms that are well-understood, as they are also frequently employed for CPU cache management [23]. Examples of such policies include:

- Least-Recently Used (LRU) [24]
- First-In First-Out (FIFO) [25]
- Most-Recently Used (MRU) [26]
- Last-In First-Out (LIFO) [27]
- Random [28]

More advance policies have also been investigated.

- LRU Insertion Policy (LIP) [29]
Similar to the LRU (Least Recently Used) [24] approach, LIP organizes cache lines in a queue with recency values. It adds a new cache line with the least-recently used value, and evictions occur on the least recently used cache line. During a cache hit, LIP moves the hit cache line to the most-recently used position.
- Bi-modal Insertion Policy (BIP) [29]
BIP is an extension of LIP designed to overcome LIP's limitations when two cyclic memory access patterns follow each other. BIP only changes during an insertion, where it utilizes a probability value to avoid inserting on the most-recently used position.
- Not Recently Used (NRU) [30]
In this policy, each cache line has a single bit that indicates whether it has been referenced or not. Non-referenced lines are eligible to become victims. During victim selection, the policy finds the first line with a reference value of 0. The insertion process sets the referenced value to 1. On a cache hit, the line's referenced value is updated to 1.
- Least Frequently Used (LFU) [31]
LFU uses a frequency value with a 4-bit counter that increments on each hit. This counter is used to determine which line should be evicted.
A new line is added with a frequency value of 0. Victim selection involves choosing the line with the lowest frequency value.
During a hit, the frequency counter is incremented by 1. If the counter is saturated, all lines in the set have their counters halved.
- Most Frequently Used (MFU) [32]
Each cache line starts with an initial frequency value of 0, which increments with each hit. When the counter saturates, all cache lines in the set have their counters halved. The line with the highest frequency value is selected for eviction. MFU increases the probability of newly loaded lines being referenced again, benefiting performance when switching between GPU workloads.

- Static and Bimodal Re-Reference Interval Prediction (SRRIP and BRRIP) [33]
- Segmented LRU (SLRU) [34]
SLRU combines LFU and LRU policies, associating two distinct lists with each cache set: a referenced list and a non-referenced list. These lists internally use the LRU policy but behave externally like the LFU policy. Newly loaded lines are inserted into the non-referenced list. Victim selection involves choosing the least-recently used line in the non-referenced list. Lines are promoted from the non-referenced list to the referenced list upon cache hits. Once the referenced list is full and a promotion is needed, the least recently used line in the referenced list is demoted.
- SLRU with Random Promotion [35]

As we add on in complexity in cache management, Dynamic Policies are also introduced:

- Adaptive Bypassing (AB) [35]:
As the name suggests, this policy is used to dynamically decide whether cache bypassing should be used for lines within a set. During a cache miss, the AB policy determines whether that line should bypass the cache. This decision is made using a bypassing probability assigned to each set. Then, an online mechanism tracks whether the bypassing decision for that set was effective or not. If it was effective, the bypassing probability is doubled, while in the case of a previously ineffective decision, the probability of cache bypass is halved.
- SLRU with Adaptive Bypassing [35]
- Set Dueling [29]
- Dynamic Insertion Policy (DIP) [29]
- Dynamic Re-Reference Interval Prediction (DRRIP) [33]
- Adaptive Bypassing (AB) [35] As the name implies, this policy is employed to make real-time determinations regarding whether cache bypassing should be implemented for a particular line within a set. When a cache miss occurs, the AB policy assesses whether that line should skip the cache. This determination is made using a bypassing probability assigned to each set. An online mechanism monitors the effectiveness of previous bypassing decisions for that set. If a previous decision was effective, the bypassing probability is doubled. Conversely, in the case of a prior unfavorable decision, the probability of cache bypass is halved.
- SLRU with Adaptive Bypassing [35]
- Set Dueling [29]
- Dynamic Insertion Policy (DIP) [29]
- Dynamic Re-Reference Interval Prediction (DRRIP) [33]

Paulsson et al. then proceeded to assess the performance of all these policies through simulations. However, drawing definitive conclusions is a challenging task. Among the *simple* replacement policies, LRU appears to outperform the others, especially FIFO, MRU, and LIFO, with LIFO and BIP demonstrating very similar performance. Slight improvements can be observed in the case of dynamic policies; however, the extent of improvement is heavily

reliant on architectural characteristics (e.g., cache size) and the specific characteristics of the workload. Nonetheless, the actual hardware implementation of these latter categories of replacement policies is considerably more complex than the more traditional policies. To illustrate, LRU seems to be the most commonly assumed mechanism for GPU’s L2 caches, as it serves as the default policy in the well-known GPGPU-sim cycle-accurate simulator [36]. However, recent research has shown that NVIDIA GPUs employ an L2 cache with a random-based replacement policy, where lines in different sets have varying probabilities of being replaced [21, 37].

2.2.4 GPU workloads

GPUs were initially developed as peripherals with a specific focus on rendering images for graphic applications [38]. However, this paradigm shifted with the introduction of programmable GPU pipelines, which were found to be well-suited for generic computing tasks that could harness a significant degree of parallelism. For example, problems involving matrices and vectors proved to be highly adaptable to GPUs, turning them into versatile general-purpose massively parallel accelerators.

In this evolving landscape, GPUs’ hardware capabilities underwent significant advancements, such as the introduction of fully programmable compute units and support for floating-point operations. Concurrently, the Application Programming Interfaces (APIs) used by developers to tap into the computational power of GPUs flourished and adapted accordingly [39].

Compute workloads

In the case of compute workloads, a GPU-accelerated program allocates memory on both the host CPU and the GPU device. Following this allocation, memory copies are orchestrated between the main system memory and the GPU-accessible memory. This ensures that when one or more kernels are launched, the data is readily available for computation and can be copied back to the host-visible memory once the required computations are completed [40]. A kernel serves as the parallel program executed by the GPU’s computing clusters, accommodating several parallel threads. Consequently, the programmer is responsible for associating each kernel with a launch configuration, grouping the requested GPU threads into a compute grid. This grid, in turn, is further subdivided into blocks of threads, which are distributed among the available GPU compute clusters [16]. These blocks are made up of *warps*, representing the smallest schedulable unit for a GPU. In the case of NVIDIA GPUs, a warp comprises 32 cooperative threads. Multiple kernels, each with distinct launch configurations, can be dispatched through multiple queues of commands. This programming pattern has enabled the successful porting of numerous parallel applications to GPUs [41, 42, 43].

Background on Graphic workloads

Graphics workloads exhibit both similarities and distinctions when compared to generic compute kernels. While the work remains parallel and is dispatched among the computing clusters within the GPU, graphic workloads often possess a fixed periodicity. Moreover, the typical output of a graphic kernel, known as a shader program, consists of a collection of images. These images collectively form the output frame, and in real-time rendering applications, a frame is frequently redrawn multiple times within a time interval. A comprehensive analysis by Ceballos et al. delves into the various types of data and associated artifacts that consti-

tute a modern 3D graphic application [44]. This analysis introduces specific terminology and concepts, including:

- **Buffer/Texture:** Memory resources to be read or written during the rendering operations.
- **Scene:** Execution of a graphics shader program consuming buffers and textures as input. A shader program can be thought as the *graphic* equivalent to a compute kernel.
- **Task:** The scene is computed in parallel by tiling the output image into independent tasks.
- **Frame:** The final blended result among the different scenes.

An intriguing question, as discussed in [44], centers around the extent to which data falling into the aforementioned categories can be effectively reused across successive frames within the same graphic application. This inquiry encompasses two key dimensions:

- **Inter-Frame Reuse:** memory resources (data) can be used by multiple frames.
- **Inter-Scene and Inter-Task Reuse:** it involves employing the same data and similar processing operations in frames belonging to different scenes or entirely independent tasks executed by the GPU.

The authors conducted a comprehensive evaluation of the impact of these various forms of graphic data sharing across an extensive set of benchmarks. They employed GLTraceSim [45] to simulate both an Integrated Intel HD Graphics 400 and a discrete NVIDIA GTX 1060. The findings from their study can be summarized as follows:

Sharing data across frames, known as *Frame Sharing*, proves remarkably efficient, with over 90% of potentially shareable data being reused in subsequent frames. Consequently, if the cache can accommodate the data for a single frame, it substantially alleviates the demand for required memory bandwidth. However, when we consider data sharing across scenes, the volume of reusable data is halved, while sharing data across tiles, referred to as *Task Sharing* (inter-task sharing), results in a more limited system sharing capacity, typically around 15%. It’s worth noting that sharing data across frames, tiles, or entire scenes not only reduces redundant memory transactions but also minimizes redundant computations, leading to enhanced performance and energy savings.

2.2.5 Memory Access Optimization

In this section of the thesis, recent research on GPU optimization at the level of memory accesses is examined in detail.

Replacement Policies

In the context of graphic workloads, the study presented in [46] introduces an approach designed for tile-based renderers. The authors propose an algorithm named MLCR (Memory Locality-aware Cache Replacement) that focuses on optimizing cache usage by efficiently distributing tile computations among the GPU’s compute clusters.

They introduce a crucial component called the *Tile Sequencer*, which encompasses a Tile Sequence Queue responsible for storing information about the sequence in which tiles should be rendered. In a dynamic fashion, each GPU compute cluster requests the next tile to be

rendered from the Tile Sequencer queue. Whenever a new tile is requested, the Tile Sequencer extracts a tile along with its ID from the queue for the GPU to render.

The construction of tiles, which are essentially rectangular areas that partition the screen, follows a hierarchical structure. Tiles are grouped to form recursive bounding structures, especially when rendering geometry spans across multiple areas. This hierarchical approach enhances memory access locality. The authors propose a cache replacement policy where each cache line is associated with a counter that keeps track of the total fetches for that line within a particular set and a level value indicating the level of the tile hierarchy to which the cache line belongs. With each global load, the fetch value increases, and it decreases every time a compute cluster finishes processing the cache line data. A critical component called RLM (Rendering Level Monitor) determines the major level, which is the tile level within the hierarchy currently accessed by the highest number of computing clusters. If all lines in a set belong to the same level, the cache lines with the lowest fetch count value are replaced. Otherwise, lines that do not belong to the major level are replaced in a round-robin fashion.

This tile-based approach effectively reduces the LLC (Last Level Cache) cache miss rate, decreasing it from 46% to 38% on average. By employing the MLCR replacement policy on L2, the cache miss rate is further reduced to nearly 36%.

Further advancements in GPU optimization beyond LRU have been explored by researchers. In particular, the study presented in [47] delves into enhancing GPU performance for compute applications by employing cache partitioning and an improved variant of PseudoLRU, which is based on insertion/promotion vectors (IPV) [48]. An IPV is essentially an array of integer values that operates as a vector. In the context of a k-way associative cache, these values range from 0 to k-1. They play a critical role in determining the new location of a cache block when it is referenced by a request, as well as the location for inserting a newly incoming block into the cache. The specific values within the IPV are determined through genetic algorithms. Different IPV's exhibit varying performance characteristics based on the memory or compute-bound nature of the kernels being executed. This approach effectively reduces the L2 cache miss rate by up to 10.11%, resulting in a notable speedup of 1.7%.

Selective Caching

Data reuse is a critical consideration, particularly in graphic workloads where it plays a central role in optimization strategies. Nevertheless, it's equally essential to explore scenarios when data reuse is less prevalent, as observed in some graphic workloads. In a study by Agarwal et al. [49], a selective caching approach is introduced. This approach prohibits the GPU from caching memory lines that would require coherence updates to synchronize between the CPU and GPU. The authors demonstrated that by employing a carefully designed selective caching policy based on request coalescing, performance levels close to those achieved by costly and sometimes undisclosed hardware coherence mechanisms can be attained (up to 93%).

Conversely, some researchers have investigated the possibility of limiting or bypassing caching, particularly at the L1 level, to achieve performance benefits. In this context, Huangfu et al. [50] examined memory access characteristics in GPGPU applications using the Rodinia benchmark suite [51]. Their analysis revealed that a significant portion of data from memory accesses in Rodinia benchmark applications is either not reused at all or reused only a limited number of times. To address this, they propose a profiling-based approach to bypass caching for data accesses characterized by low utilization rates, with the goal of improving performance and energy efficiency. The profiling phase involves associating specific addresses with load utilization counters and reuse factors. These values are then combined, and if the resulting expected traffic remains below a certain threshold, their approach prevents caching the line

in L1. Using the GPGPU-Sim simulator, the authors evaluated this algorithm on an NVIDIA Fermi GTX 480 with the Rodinia benchmark suite. On average, the proposed method reduces global memory traffic by 22.2%, improves system performance by 13.8%, and reduces energy consumption by approximately 6%.

A similar approach was explored in [52] for comparable compute benchmarks. Their algorithm consists of three steps: (1) analyzing access patterns, (2) estimating memory traffic, and (3) determining which load instructions should be cached in L1. In essence, their algorithm discovers memory access patterns for individual SIMD load instructions within a compute kernel through extensive profiling. By storing information about these patterns and representing data addresses as functions of thread IDs, they propose a mechanism to bypass L1 caching, reducing unnecessary data movement. In their study, kernel completion latency was reduced by up to 16% on real hardware.

Non-contiguous memory accesses present a challenge in efficiently handling GPU memory requests, leading to increased memory traffic. Researchers have explored the possibility of using variable cache line sizes to address this issue and accommodate diverse access patterns exhibited by GPU kernels. One solution introduced by researchers is Elastic-Cache, as outlined in [53]. Elastic-Cache supports both fine and coarse-grained cache-line management and builds upon prior work, including Amoeba-Cache [54], originally designed for CPUs. This approach revolves around the concept of variable cache-line sizes. Elastic-Cache is designed to adapt to different scenarios. When an application demonstrates effective coalescing behavior, it allows for wider cache-line access. Conversely, when applications struggle to exploit broader cache lines, the cache line size is reduced accordingly. The reported results indicate that when applied at the L1 level, elastic caches yield an average improvement of 58% in Instructions Per Cycle (IPC) across a wide range of compute and graphics benchmarks, particularly benefiting applications characterized by irregular memory access patterns.

CPU and GPU interactions

Understanding the implications of sharing data through caches in a heterogeneous System-on-Chip (SoC) comprising a multicore CPU host and an integrated GPU is crucial [55]. In such a setup, explicit data transfers between the device and host can be replaced by a unified shared memory space accessible to both the CPU and GPU [55]. However, this arrangement raises two significant performance-related concerns:

- CPU and GPU Memory Coherence for Shared Data [56]: Ensuring that both the CPU and GPU can access shared data without conflicts or inconsistencies is a complex challenge. Coherence mechanisms must be in place to manage data consistency across these different processing units.
- Memory Interference Between CPU and GPU [13]: The CPU and GPU share memory resources, which can lead to interference when both are actively accessing memory. This interference can negatively impact overall system performance.

In the context of memory interference within CPU systems, traditional solutions often employ memory bank [57] and cache [58] partitioning strategies. Cache partitioning, in particular, is a widely used runtime Last-Level Cache (LLC) management approach in multi-core processors. It assigns exclusive access to cache ways to CPU cores, reducing interference among memory clients with simultaneous memory requests. Researchers in [59] propose a solution to address these issues in integrated systems where CPUs and GPUs share not only the main SoC memory but also compete for shared cache access. They introduce a Latency

Sensitivity-based Cache Partitioning (LSP) framework that incorporates a lightweight run-time mechanism to quantify tasks’ latency-sensitivity metrics. This framework facilitates the partitioning of shared LLC resources between the CPU and GPU. The LSP algorithm initiates with an initial partition where the entire cache is allocated to the GPU. Through a profiling phase, cache resources are dynamically moved from the GPU to the CPU. In this context, a cache resource corresponds to a cache way. A component called the System State Monitor periodically samples performance metrics related to Instructions Per Cycle (IPC) and LLC access latency. A cost function is derived based on these metrics, as well as LLC hit rate and off-chip memory access latencies. This utility function allocates larger LLC partitions to memory clients more sensitive to LLC latency. The results demonstrate that LSP partitioning enhances overall system throughput by an average of 8% (with a maximum improvement of 27%) when compared to the state-of-the-art partitioning mechanism, TAP (tlp-aware cache management policy) [59].

2.2.6 Compute Optimization

As outlined in Section 2.2.4 and 2.2.4, both graphic and compute workloads can exhibit periodic behavior, where a substantial portion of previously computed values is repeated in subsequent job instances with similar or identical input data. This recurrence in computation, particularly evident in graphic workloads, presents an opportunity to reduce the overhead of redundant calculations. To address this, the concept of *memoization* [60] is introduced.

Memoization is an optimization technique designed to circumvent redundant computations by harnessing the outcomes of previous executions that share similar input data characteristics. In simpler terms, when the system identifies that input data remains relatively consistent across various instances of the same task, it can reasonably infer that the results of these instances will also exhibit similarity. This avoidance of repetitive computations proves highly effective in enhancing execution speed and conserving energy. Here’s how memoization works: when a computation is initially performed, its result is dynamically cached within a Look-Up Table (LUT), associating it with a representation of the input data used. Subsequent executions of the same task first attempt to match their input data with the values stored in the LUT. If a match is found, the cached result is retrieved, sparing the need for redundant computation. Memoization relies on the assumption that the same input values will consistently yield the same result or only exhibit minor variations.

Researchers in [61] delve into the realm of memoization within the context of mobile graphic applications, particularly emphasizing the prevalence of scene replication across multiple frames in these applications. Throughout the paper, they focus on the ARM Mali 400-MP mobile GPU, providing insights into its internal architecture. The ARM Mali 400-MP GPU incorporates programmable vertex and fragment shaders, organized as computing clusters, which harness Tile-Based Deferred Rendering (TBDR) [62]. This GPU comprises three primary components: the *Geometry Unit*, the *Tiling Engine*, and the *Raster Unit*.

To curtail a significant portion of redundant computations, the researchers augment the existing Parallel Frame Rendering (PFR) [63] approach with a task-level memoization scheme. PFR allows the simultaneous computation of two frames by logically dividing the GPU hardware into two separate computing clusters. In their investigation of potential fragment redundancy across frames, the authors highlight that employing exact-input to exact-output memoization is not always a practical approach. Consequently, they introduce the concept of imperfect hash functions to be applied to frame input data. This approach seeks to strike a balance between computation avoidance and image quality preservation.

Their memoization approach comprises three distinct phases: the *identification of potential*

candidate fragments, the *lookup of previously memoized fragment data*, and the *substitution of fragment computation* with memoized information.

Initially, input image fragments undergo an assessment to determine if they meet specific criteria that qualify them as candidates for memoization. These criteria are based on factors such as the number of registers and texture samples required by the fragment. Fragments failing to meet these requirements proceed through the compute pipeline without undergoing memoization operations. Conversely, qualifying fragments progress to the next stage, where their inputs are subjected to hashing for the lookup step. Subsequently, a lookup operation is performed on the Look Up Table (LUT). A successful hit in the LUT signifies that memoized data for the fragment’s prior color exists, obviating the need for further calculations for that particular fragment. Conversely, if no match is found in the LUT, the GPU proceeds with the standard computation process for the fragment. Regarding the management of the Look Up Table (LUT), the researchers tested a Last Recently Used (LRU) policy. To evaluate the efficacy of their approach, benchmarking was carried out using the TEAPOT toolset [64]. The results demonstrated an average speedup of 14.8% across both 2D and 3D applications.

A recent advancement in this field is the introduction of a technique called Rendering Elimination (RE) [65]. RE is designed to reduce energy consumption by 43% and achieve an average speedup of 1.74x in Android applications. This approach is a variation of memoization and focuses on avoiding redundant tile computations. In RE, the signatures of tiles are derived by hashing their inputs. These inputs encompass various vertex attributes, including the vertex positions that make up the fundamental primitives for 3D objects, as well as their texture mapping coordinates, positions, and normals. Given that primitives, such as triangles and lines, can span across multiple frame tiles, these tiles are organized into blocks, which are then associated with the vertex attribute inputs. Compared to fragment memoization, RE imposes minimal hardware overhead, accounting for less than 1% of the total GPU area. Moreover, the operative latency of RE is often concealed by other processes within the graphics pipeline.

An adaptation of memoization, with a focus on avoiding redundant computations by leveraging past calculations, is presented in [66]. This approach, known as RAVEN, is tailored for rendering applications in mobile games, with the aim of reducing real-time energy consumption. The fundamental idea behind RAVEN is that if two consecutive frames are highly similar, subsequent frames can be delayed to conserve energy. To implement RAVEN, the system first calculates a similarity index between consecutive frames using the Structural Similarity Index (SSIM) [67]. Then, this index is compared to a predefined threshold. When the threshold is met, the RAVEN mechanism introduces delays into the rendering loop to postpone or skip the computation of similar frames. Experiments detailed in the mentioned paper demonstrate that RAVEN [66] can save up to 35% of total device energy consumption without causing significant degradation in the user experience during mobile gaming. Notably, the authors engaged 12 human participants to calibrate the SSIM threshold, using their feedback on the generated frames. It’s worth mentioning that many researchers have shown the applicability of memoization to generic compute workloads as well, as demonstrated in works like [68, 69, 70].

2.2.7 Ray Tracing and memoization

Recent GPUs come equipped with advanced hardware features tailored for ray tracing, a technique that offers developers a more precise model of how light interacts with three-dimensional scenes. In ray tracing, scene objects, i.e., their spatial positions, are organized using acceleration structures (AS), such as Bounding Volume Hierarchies (BVH). These BVH structures are tree-like data arrangements traversed by rays originating from the observer’s eyes or

light sources within the scene [71]. The traversal of BVH structures is a pivotal aspect of ray tracing-based graphical applications [72]. Unlike traditional raster-based graphics, where scenes are rendered based on a grid of pixels, ray tracing employs rays that emanate from the observer’s eyes and interact with objects and light sources, accumulating color information as they progress. In this rendering paradigm, a common approach is to cast a ray from the observer’s eyes to each pixel in the image to be generated. This ray may either *hit* an object within the scene, resulting in a hit, or miss any objects, signifying a *miss*. In the case of a hit, the ray may be altered, for instance, by reflection or refraction, based on the material it intersects, initiating iterative scene traversal operations. The traversal operations are crucial components in ray tracing-based applications, and they significantly differ from those in the traditional rendering pipeline. Therefore, it is essential to explore specific computational optimizations to prevent redundant calculations. In this context, the authors of [73] introduce a memoization process that employs a hashing function to create a prediction table for storing similar *hits* during BVH-tree traversal. A successful prediction can direct a ray directly to a leaf node, bypassing the need for an entire data structure traversal. Conversely, a misprediction leads to a full traversal operation, which is the baseline behavior in such scenarios. The findings presented in the paper indicate that, under ideal circumstances where memory and hardware complexity are not limiting factors, it is possible to predict as many as 38% of the rays using an optimal hashing function and a specialized hardware Look-Up Table (LUT) designed for Acceleration Structure (AS) traversals, referred to as a predictor table. In a formal sense, a ray with origin o , direction d , and length (t and d respectively) can be expressed by the equation $o + t \cdot d$, where t is a parameter that determines a specific point along the ray. Each computing cluster within the GPU is equipped with its predictor table, where each row resembles a set-associative cache. Each way within this cache functions as a predictor, comprising a valid bit, a ray hash tag, and one or more slots to store predicted nodes (essentially an offset into the BVH tree buffer). To optimize the indexing of the predictor table, the authors devised a key by combining the three-dimensional ray origin with the two-dimensional ray direction through a hash function.

To prevent inconsistencies in Single Instruction Multiple Data (SIMD) execution caused by mispredictions, the authors introduced a warp repacking extension in the predictor. This extension includes a queue that ensures coherent workload distribution among threads within a warp. Furthermore, it creates opportunities for memory access coalescing.

The authors conducted simulations and benchmarking on various 3D workloads, proposing and implementing a simulated ray tracing (RT) Core (similar to those found in the latest NVIDIA GPUs) within the GPGPU-sim 4.0.1 framework. Based on their simulations, the authors reported that their prediction system achieved an average speedup of 26% compared to the baseline RT unit, all while reducing memory accesses by 13%.

Regarding shading, which involves the actual computation to determine pixel colors after the initial ray casting phase, recent patents [74, 75] explore the potential to reduce the number of pixel computations through variable shading techniques. However, the impact of these methodologies on performance remains uncertain.

Paper / Patent	Target Study	Compute Optimization	Cache Access Opt.	Improved Metrics
[49]	Compute	-	Selective Caching	Avoid Redundant Cache Accesses
[46]	Graphics	-	Cache Replacement Policy	Cache Hit Ratio
[50]	Compute	-	Avoiding Caching data Unreused	Avoiding Cache Pollution
[52]	Both	-	Utility-Based Cache Partitioning(UPC)	Cache Hit Ratio
[59]	Compute	-	Latency Sensitivity-based Cache Partition (LSCP)	Cache Hit Ratio
[53]	Compute	-	Elastic-Cache (L1 cache)	Cache Hit Ratio
[61]	Graphics	Memoization	Exploiting Redundant Computations	Computation Timing
[63]	Graphics	Reusing Redundant Computations across frames	Cache texture locality	Computation Timing
[76]	Graphics	Visibility Buffer	-	Performance and less G-Buffer Footprint
[47]	Compute	-	Tree-based PLRU with IPV	Cache Hit Ratio
[77]	Graphics	-	L2 Access	L2 Bandwidth Optimization
[66]	Graphics	Frame Skipping	-	Power Consumption
[78]	Graphics	Prediction Reuse	Reuse Probability	Cache Hit Ratio
[79]	Graphics	Visibility Rendering Ordering	-	Less Overshading
[65]	Graphics	Skipping Tiling	-	Power Consumption and Performance
[73]	Graphics(Ray-Tracing)	Rays Look-up-Table / Memoization	-	Acceleration Structure (Ray-Traversal)
[75], [74]	Graphics	Recasting Calculation	-	Improve Image Quality and Avoiding Shading Op.

Table 2.1: Summarizing architectural optimization in graphic and compute GPU based workloads

2.2.8 Conclusion

In this chapter, an in-depth exploration of various methodologies has been undertaken. This includes a thorough examination of different cache policies for the Last Level Cache (LLC) in GPUs, aiming to enhance our understanding of optimal strategies for managing cache at the highest level. Furthermore, the discussion extends to the nuanced realm of memoization techniques employed in different domains. Building on this groundwork, the subsequent chapter will focus towards a more detailed investigation of optimization techniques, specifically addressing memory interference. This key aspect of the research aims to unravel intricacies related to how memory interactions can be finely tuned to optimize overall system performance.

Chapter 3

Memory-Aware Latency Prediction Model for Concurrent Kernels in Partitionable GPUs

3.1 Introduction

In the domain of Cyber-Physical Systems (CPS) and their associated applications, there is a growing demand for processing platforms capable of handling data and compute-intensive tasks. These platforms typically incorporate multi-core CPUs that can collaborate with massively parallel compute accelerators. Among the prevalent architectural solutions for these accelerators, wide known choice is represented by General-Purpose Graphics Processing Units (GP-GPUs or simply GPUs). GPUs are often chosen due to their remarkable performance-per-watt ratio, widespread availability, and the existence of well-established ecosystems comprising APIs, libraries, and development kits that streamline their utilization. As the complexity of applications grows and their computational requirements expand, the architectural designs of GPUs must adapt to this evolving landscape. GPU vendors' road-maps indicate that *advancements in transistor scaling continue to boost GPU performance by elevating GPU frequencies and enlarging die sizes* [80]. This growth in die size leads to an increment in the count of parallel processing element clusters housed within a single GPU die, hence, each new generation of GPUs tends to be *larger* in terms of computational capacity. However, when individual compute kernels operate in isolation, they may *not fully harness* the increased parallelism provided by these modern GPUs [81, 82]. Consequently, system engineers are exploring mechanisms and methodologies for dispatching multiple kernels to GPUs within overlapping time windows, dynamically allocating varying numbers of GPU cores and memory resources to different kernels based on their timing and latency requirements. Assigning GPU resources to kernels can be achieved through various mechanisms, one of which is referred to as *Multi-Kernel Execution*. In this approach, groups of threads belonging to different kernels can be distributed across all of the GPU's compute clusters. While this method has proven highly effective in enhancing overall throughput [83], it presents significant challenges in controlling individual kernel latencies [16]. These challenges arise because GPU threads must not only compete for compute resources (e.g., internal cluster schedulers, ALU pipelines, etc.) but also contend for access to the GPU cluster's cache hierarchy and local scratch-pad memories. An alternative approach involves spatial kernel partitioning. In this paradigm, the GPU's compute clusters are partitioned into non-overlapping segments, with one or more clusters assigned to individual kernels. This configuration designates the GPU's Last-Level Cache (LLC) as the primary contention point when multiple kernels execute within overlap-

ping time windows. Consequently, system engineers can focus on mitigating a single memory contention point once the scaling of kernel execution times concerning assigned GPU partitions is understood. However, accurately deriving the scaling of kernel execution times based on variable compute cluster partitions is a non-trivial task. In an ideal scenario, this scaling function should be linear, meaning that the kernel’s execution time decreases linearly as more GPU compute clusters are linearly assigned. However, the GPU’s parallel capabilities *may surpass the degree of parallelism inherent to the kernel*, potentially reaching the theoretical peak performance without fully utilizing all GPU partitions. Furthermore, memory bandwidth plays a crucial role, with *highly memory-bound kernels* causing the L2 to VRAM memory bandwidth to act as a *bottleneck*, further impeding expected performance scaling. This aspect has been extensively explored and is known as the GPU roofline model [84, 85].

In this chapter of the thesis, the objective is to explore the behavior of GPU spatial partitioning when multiple kernels execute concurrently within separate GPU partitions. Specifically, an intuitive yet practical performance prediction model that is sensitive to memory interference and considers the unique characteristics and memory utilization patterns of individual kernels will be introduced. Notably, the proposed approach stands out from prior research by offering reasonably accurate predictions of kernel execution times and demonstrating scalability when dealing with an increasing number of concurrent GPU kernels. This part serves as an initial investigation into addressing memory interference within partitions, with the overarching aim of ultimately contributing to the development of efficient schedulers for managing the simultaneous execution of multiple kernels on a GPU.

The work presented in this chapter is derived from my publication [86], which was featured in the Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP).

3.1.1 GPU Programming Model and Scheduling

In this section, the focus will be on the GPU programming model and the selected simulation environment. Regarding GPU terminology, without excessive loss of generality, this part will use the NVIDIA terminology, presented in Section 2.2 of Chapter 2, in the context of the widely adopted CUDA API ¹.

From a software perspective, developing a GPU application involves addressing a heterogeneous programming challenge. This is because the host CPU is responsible for issuing copy and compute commands to the GPU, and this is accomplished using dedicated APIs designed to facilitate access to the GPU’s computational acceleration features [39]. Examples of these APIs include NVIDIA CUDA (Compute Unified Device Architecture), which is a proprietary API (Application Program Interface), and OpenCL ² which, in contrast, is an open-standard.

In GPU APIs like CUDA and OpenCL, commands sent from the CPU to the GPU are primarily associated with the execution of parallel programs on the GPU’s processing clusters. These executable units in the GPU are commonly referred to as *kernels*. Therefore, a GPU application is responsible for managing data allocation, transferring data between CPU-visible and GPU-visible memory areas (and vice versa), and invoking kernel execution.

Programmers have limited control over the scheduling mechanisms of a GPU application. Kernels are dispatched through parallel queues of commands known as *streams*, in CUDA terminology. In NVIDIA GPUs, commands within the same stream are executed in the order they are submitted. If the occupancy of a kernel within a stream does not saturate the compute capability of every Streaming Multiprocessor (SM), multiple kernels from different streams can execute in parallel. Threads within each kernel are distributed across all sets

¹CUDA API: <https://docs.nvidia.com/cuda/>

²<https://www.khronos.org/opencl/>

of SMs in a round-robin fashion. Consequently, if one kernel does not fully utilize all the memory (e.g., shared memory and registers) or compute resources (e.g., CUDA cores) of the SMs, another kernel from a different stream can run in parallel [16].

It’s essential to emphasize that the current CUDA APIs available for consumer-level GPUs do not provide programmers with control over how kernels are assigned to specific *partitions* on NVIDIA GPUs. In the CUDA programming model, there is no publicly available method for specifying which partitions should be utilized for a particular kernel. Consequently, the default CUDA stream scheduler allocates resources and enforces resource sharing among blocks of threads from different kernels within a single Streaming Multiprocessor (SM). This resource sharing can have an impact on the predictability of execution latencies.

The term *partition* refers to a specific subset of SMs. In our case, these partitions are contiguous, meaning that they are ordered based on their IDs. For instance, in a GPU with 30 SMs numbered from 0 to 29, a contiguous partition comprising 10 SMs might include SMs with IDs ranging from 0 to 9. In contrast, a non-contiguous partition would include SMs with IDs that are not ordered.

Although workarounds have been proposed for research and experimental purposes to enable inter-SM resource partitioning rather than the default behavior (e.g., CUDA persistent threads [87]), it’s worth noting that these workarounds have inherent limitations when it comes to GPU partitioning, as discussed in Section 3.8. Therefore, in this thesis, has been implemented and examined a memory-aware latency prediction model on partitionable GPUs using a cycle-accurate simulator (GPGPU-Sim).

3.1.2 Cycle Accurate Simulation Through GPGPU-Sim

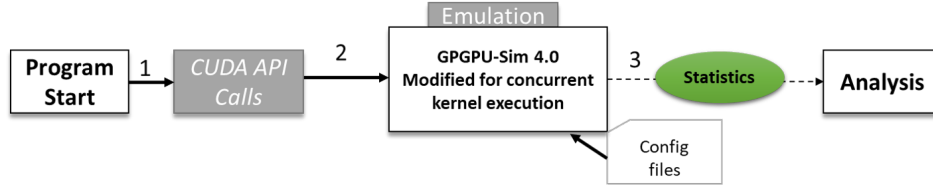


Figure 3.1: Workflow from host to GPGPU-Sim.

To attain cycle-level accuracy in simulating emerging hardware architectures, the implemented and executed approach is done by adopting GPGPU-Sim [36]. GPGPU-Sim is a cycle-level simulator with the remarkable capacity to replicate a wide range of GPU architectures and execute computing workloads written using well-known APIs such as CUDA or OpenCL. As demonstrated, GPGPU-Sim, developed and maintained by a dedicated research group, grants us the capability to scrutinize the majority of market-available architectures. This serves as a valuable resource, offsetting the inherent scarcity of documentation concerning low-level architectural specifics often provided by manufacturers.

To assess the correlation between the simulations and the real-world behavior of GPU hardware, the creators of GPGPU-Sim compared the Instruction Per Clock (IPC) as measured in simulations and the corresponding real-world figures obtained from NVIDIA GPUs across a set of benchmark kernels. Impressively, this scrutiny yielded an impressive similarity index of 98.37%. Such a high degree of similarity instills confidence in the analyses conducted using this simulator. Furthermore, it grants the flexibility to tweak certain hardware characteristics, thereby facilitating an exploration of the limitations inherent in existing GPU devices and the testing of architectural enhancements aimed at further optimizing performance. GPGPU-Sim was utilized to perform timing analysis in terms of emulation cycles and to quantify the

interference that arises at the L2 cache memory level when multiple kernels are executed concurrently. The L2 cache memory is the focus of this section, as it represents the initial contention point for GPU thread blocks from different kernels that are assigned to non-overlapping sets of SMs. An effort has been made to develop a custom extension for GPGPU-Sim to facilitate the specification of the mapping of concurrent kernels into user-defined SM partitions.

GPGPU-Sim produces a compiled library, enabling the simulator to interpret the compiled instruction program set of CUDA device low-level source code, referred to as PTX (Parallel Thread Execution). It then simulates this code within the desired hardware architecture, as illustrated in Figure 3.1. The emulated architecture comprises multiple Single Instruction Multiple Thread (SIMT) cores that mimic multithreaded pipelined Single Instruction Multiple Data (SIMD) execution units. These execution units are organized into clusters (SMs). Each SM contains various ALU units, application-specific co-processors, as well as the complete memory hierarchy and interface described in the previous section. The emulated SMs are interconnected with the memory partition through a network. Within the interconnection network, packets containing operations to be executed are injected into a SIMT FIFO (First Input First Output) queue, which manages and redirects them to the corresponding SIMT SM instruction cache. These packets may include a memory response to service an instruction fetch miss or to pass through a memory pipeline via a Load Store unit (LDST). Once instructions are loaded into the instruction cache, they undergo fetching, decoding, and eventual execution by the entire core set.

3.2 Methodology Overview

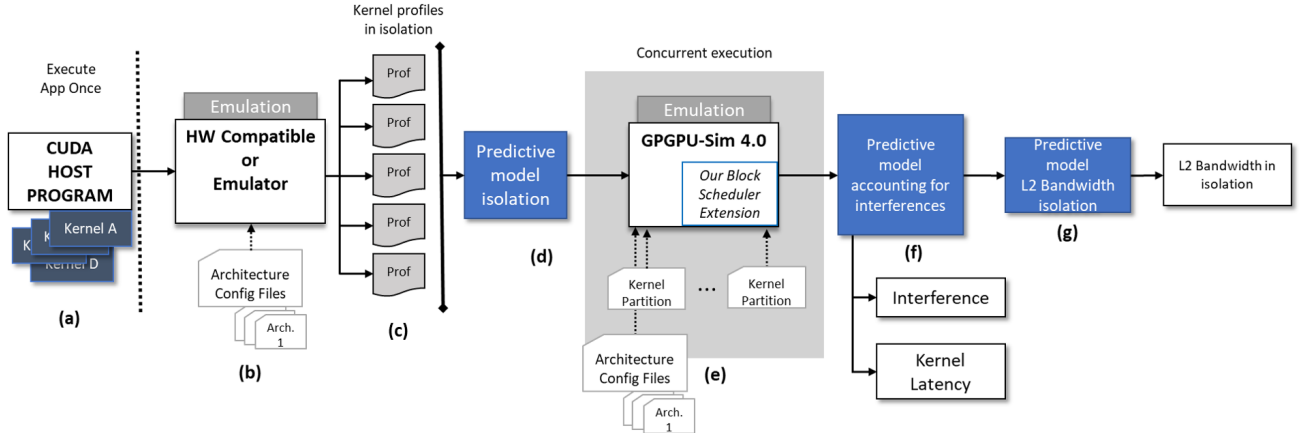


Figure 3.2: Methodology overview.

The developed methodology (GPGPU-Sim extension) is depicted in Figure 3.2, providing an overview of the steps involved in conducting the comprehensive analysis that underlies the various predictive models presented in this thesis. To investigate the behavior of a partitioned GPU, the concept of dynamic partitioning within a GPU, enabling the assignment of SMs to kernels has been introduced. In the subsequent section, an elaboration on the engineering modifications made to GPGPU-Sim to adapt its thread block scheduling algorithm to support concurrent execution of multiple kernels within partitioned SM sets is presented.

With the essential infrastructure established to facilitate the study (Figure 3.2, blocks a and b), two profiling phases have been carried out:

1. *isolation profiling* and,

2. parallel kernel profiling.

The isolation profiling, detailed in Section 3.4, involves profiling individual kernels in a baseline scenario where isolated kernels may be dispatched across the entire pool of available SMs, representing the default GPU behavior. This process results in the collection of profiles detailing each kernel’s execution times and memory behavior. The evolution of these profiles is then observed when the same kernel is mapped to a subset of SMs (Figure 3.2, block c). Consequently, trends for completion latency and requested memory bandwidth (BW) concerning SM partition sizes for each kernel are established. These profiles serve as the basis for deriving predictive models that enable system engineers to determine the default behavior of kernels on the GPU (Figure 3.2, block d) without the necessity of testing all conceivable partition sizes (Section 3.5).

Additionally, a method in Section 3.7 for predicting the L2 cache bandwidth required by a single kernel running in isolation across varying SM sets (Figure 3.2, block g) is presented.

3.3 Simulation Settings and the Implemented GPGPU-Sim Extension

The chosen GPU architecture for the experiments was the NVIDIA RTX 2060, primarily because GPGPU-Sim Version 4.0.0 supports its emulation and it exhibits a high correlation index with real hardware (99% [36]). The used configuration environment is detailed in Table 3.1.

To analyze the target scenarios, GPGPU-Sim has been modified in order to enabling the mapping of a GPU kernel to a subset of available SMs. This modification requires the use of a special configuration file as input, during the simulator initialization phase, the `createSIMTcluster()` function in the `gpu-sim.cc`³ source file loads this configuration file.

Table 3.1: RTX 2060 configuration. Same configuration file presents in the GPGPU-Sim configurations files’ folder.

Parameters	Value
Name	RTX 2060
Clusters	30
SM per Cluster	1
Total SM	30
L1 Cache	32B sector, 4 banks, 96-way
L2 Cache	24 banks, 128 KB block, total 3MB
Machine ISA	sm_75
Nominal HW Bandwidth	348 GB/s

In each line, the *configuration file* includes a kernel name and a range of integer values corresponding to SM IDs. These kernel names are obtained after the initial execution and are used within the configuration file. The associated ranges for the kernels define the partitions to which each kernel will be mapped. These partitions are contiguous and unfragmented, meaning each kernel will be mapped to a partition composed of contiguous SMs, by ensuring that different partitions do not overlap. This mapping is stored as a data structure visible to all emulated streaming multiprocessors. This approach involves establishing a static task-set to define static partitions. To support dynamic creation of kernel-to-SM-partition mapping during the entire simulation, GPGPU-Sim has been extended, meanwhile, the rest of the

³GPGPU-Sim repository - https://github.com/gpgpu-sim/gpgpu-sim_distribution.

simulator initialization, in which the actual GPU object is created, remains unchanged with respect to the standard GPGPU-Sim version.

The extension involves reading the data structure containing partition information, which necessitates making certain modifications to the *gpu-sim* class (.cc and .h) and the *shader.cc*. The modified version of GPGPU-Sim with this feature is available at https://git.hipert.unimore.it/amasola/gpgpu_sim.

From the CPU side code (host), multiple kernels are dispatched using multiple streams. In the emulated CUDA runtime, the workload of different kernels is divided into thread blocks, referred to as CTAs (Cooperative Thread Arrays), which each individual emulated SM can manage. In the standard version of GPGPU-Sim, this process involves initially selecting a kernel and then considering the dispatching of blocks associated with that specific kernel. This procedure follows the conventional stream scheduling logic (refer to Section 3.1.1). To provide further detail, a method is called for every SM instance, known as SIMT cluster in GPGPU-Sim, to retrieve a kernel for execution. Once a kernel is chosen, its CTAs are distributed among the SMs in a round-robin manner.

The kernel selection method, referred to as `select_kernel()` in the code, has been altered to limit the selection of kernels to those associated with the partition containing the SIMT cluster where CTAs will be dispatched. This modification ensures that thread blocks of particular kernels are effectively mapped to predefined sets of SMs (refer to Figure 3.3).

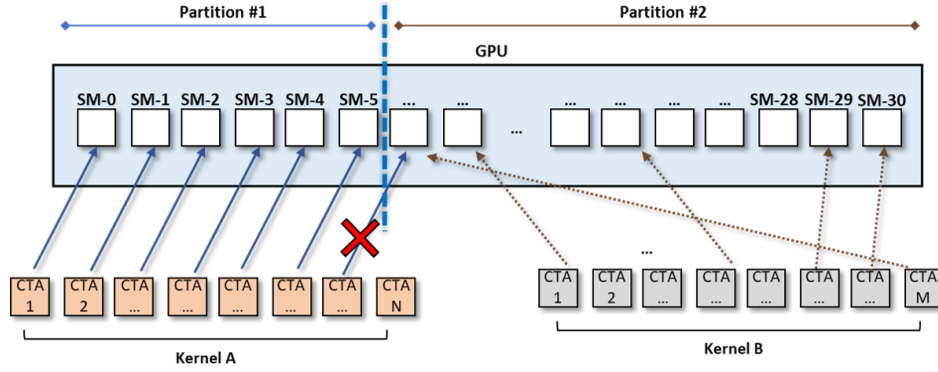


Figure 3.3: Assigning kernels to specific GPU partitions in the proposed GPGPU-Sim extension. This example have two kernels for two partitions.

An host CUDA application has been developed to launch various CUDA kernels in different CUDA streams for conducting simulations.

This benchmark application is employed in conjunction with various settings for the additional configuration files to observe the behavior of concurrently executing kernels across different partition sizes. The objective is to comprehend the performance scaling of individual kernels when executed in different partitions while considering memory interference on shared memory, specifically the GPU L2 cache.

3.4 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 3.2.

Table 3.2: Kernels used for profiling on an NVIDIA GeForce RTX 2060 emulated architecture.

Kernel	Input (MiB)	Output (MiB)	Shared Mem.	Launch Config.	KAI	Duration ms	Origin
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 [88]
PF	0.5 + 255.5	0.5	2 Kb	(1024,1,1),(256,1,1)	0.475	0.337	Rodinia [51]

During this phase, the cycles needed for each kernel to finish, the L2 memory bandwidth, and the GPU L2 access count were profiled individually for each kernel. These metrics were collected while altering the SM partition size. The number of SMs allocated for each kernel across the different experiments is $\in [5, 10, 15, 20, 25, 30]$. To profile the information of each individual kernel, every kernel has been compiled and launched using the configuration provided in Table 3.2 on GPGPU-Sim. This process allows the collection of various execution details and statistics about the hardware counters. It will not be necessary to repeat the tests multiple times since GPGPU-Sim is designed to be a deterministic simulation, unlike real hardware architectures where the interaction between GPU driver and Operating System is unpredictable, potentially introducing variability in the observed metrics.

3.4.1 Kernels Memory Bandwidth Analysis

In the initial phase of profiling experiments, the behavior of how the memory bandwidth requirements change for each kernel as the number of SMs assigned for execution increases has been profiled and it is depicted in Figure 3.4. It is evident that when different SMs access the memory interface in parallel, as each SM is directly connected to distinct L2 banks, the memory bandwidth tends to decrease as the number of SMs allocated to a kernel is reduced. This reduction in parallelism for memory interface access leads to lower memory bandwidth. Based on the results from this initial profiling phase and another research [89], the kernels were categorized into three types: *Memory Intensive*, *Hybrid*, and *Computational*.

- *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 3.4a.

- *Hybrid*: $\{CONV, MVT, PF\}$ the BW demand ($[10\%, 70\%]$) scales linearly based on the number of SMs with average utilization of compute units, Figure 3.4b.
- *Computational*: $\{RAYTRACE, DXTC\}$, the BW utilization is low ($< 10\%$) while the computation requirement linearly increases with the number of SMs, Figure 3.4b.

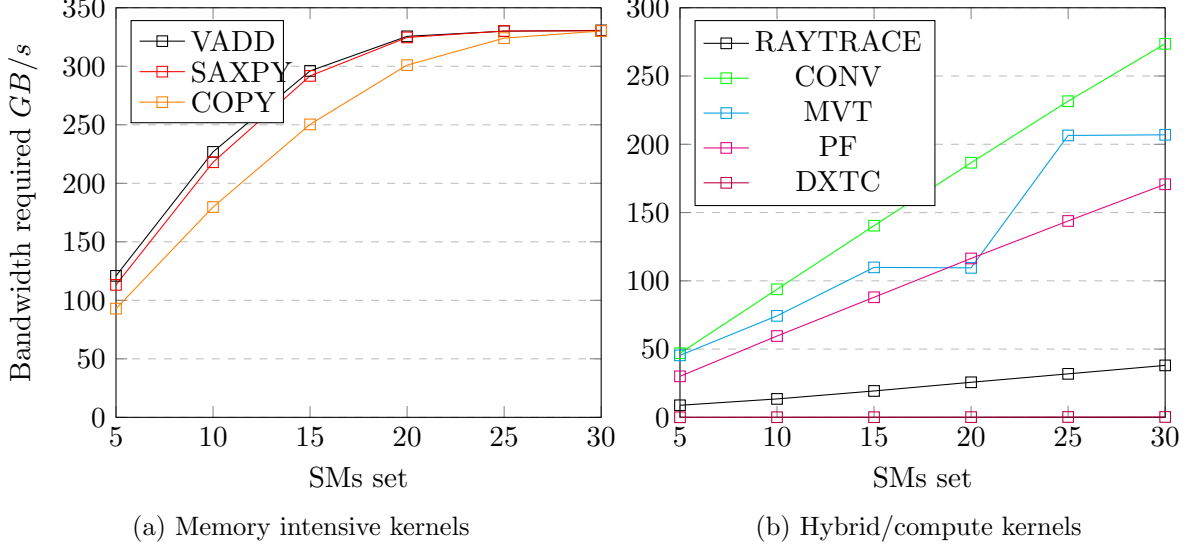


Figure 3.4: Memory BW required by each kernel.

Multiple tests were conducted to investigate potential profiling errors in MVT. However, it remains unclear why MVT exhibits unexpected behavior (as depicted in Figure 3.4b) when the SM set size is either 20 or 30, despite these sizes having values near those of the previous sets. This peculiar behavior could be potentially attributed to the simulated hardware architecture.

It is also worth noting that the BW reaches a saturation point at 330 GB/s while the architecture's nominal BW is 348 GB/s. This saturation point will be subsequently referred to as the *Effective Maximum Bandwidth* (EM_{BW}).

3.4.2 Kernels Completion Latency Analysis

Secondly, the focus shifts towards examining how the kernel execution time scales as the number of assigned SMs change. Figure 3.5 summarizes the collected results, representing the execution time in simulation cycles. Concerning memory-intensive kernels, a distinct pattern emerges, characterized by a decreasing exponential trend in kernel latencies. However, these latencies tend to stabilize once they approach the memory saturation point, as illustrated in Figure 3.4. In essence, the anticipated linear decrease in latency gets disrupted due to memory latencies taking precedence over instruction execution times, in contrast, this behavior is not observed in the case of other kernel types. While their latency trends are not perfectly linear, most of the tested kernels never reach a plateau. Instead, an uninterrupted exponential decrease in completion times is witnessed.

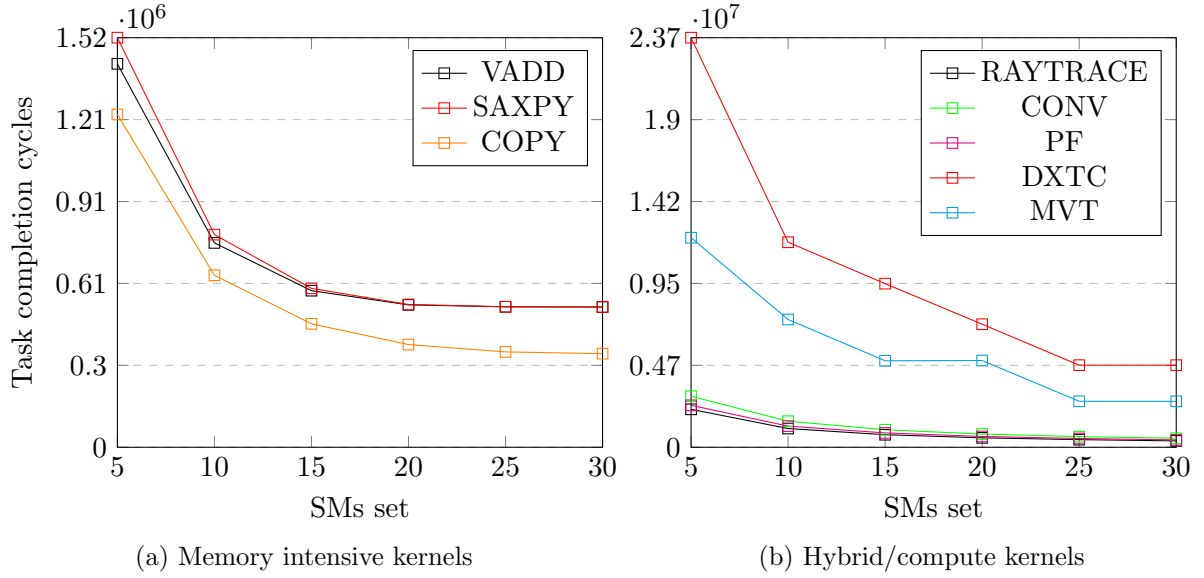


Figure 3.5: Completion latencies analysis for each kernel (cycle analysis).

The comprehensive analyses of both bandwidth (BW) and kernel latency provide valuable insights for the construction of further predictive models.

3.5 Predicting Latencies Depending on the Assigned SMs

Based on the initial findings of Section 3.4, it becomes evident that the number of SMs has a significant impact on both memory bandwidth (BW) and kernel completion latency (Cycles). To elaborate, in the case of non-memory intensive kernels, there is a linear correlation between memory demand and completion cycles, resulting in proportional scaling. However, for memory-intensive kernels, as their memory requirements grow, their capacity to fully utilize the increased parallelism offered by a larger number of SMs diminishes. As a result, the predictive model is pioneering in its consideration of this categorization of memory and latency behavior among kernels.

Initially, the calculation involves the determination of kernel’s BW utilization, denoted as $U_{bw}(k)$. This utilization, expressed in Eq. 3.1, is defined as the ratio between the maximum nominal L2 BW (BW_{max}), which is 348 GB/s in our hardware architecture, and the effective requested BW by the kernel when executed on the maximum available SMs N , denoted as $BW_{isolated}(k, N)$.

$$U_{bw}(k) = \frac{BW_{isolated}(k, N)}{BW_{max}} \quad (3.1)$$

To model the different behavior of memory-intensive kernels in the subsequent formulas, it is necessary to model the behavior of the GPU memory controller that handles memory access requests made by the SMs to operate. The GPU memory controller will reach a saturation point where it will no longer be possible to accommodate all requests simultaneously. Due to this hardware bottleneck, the cycles will no longer scale linearly, as a memory saturation is reached, as observed in Figure 3.4a and 3.5 for memory-intensive kernels. Therefore, it is necessary to model the trend of memory saturation based on the utilized bandwidth factor $U_{bw}(k)$ to capture the behavior of the hardware architecture and how memory requests work. To capture this characteristic, a sigmoidal function is used with appropriate parameters as in Figure 3.6 as other researchers used in [90].

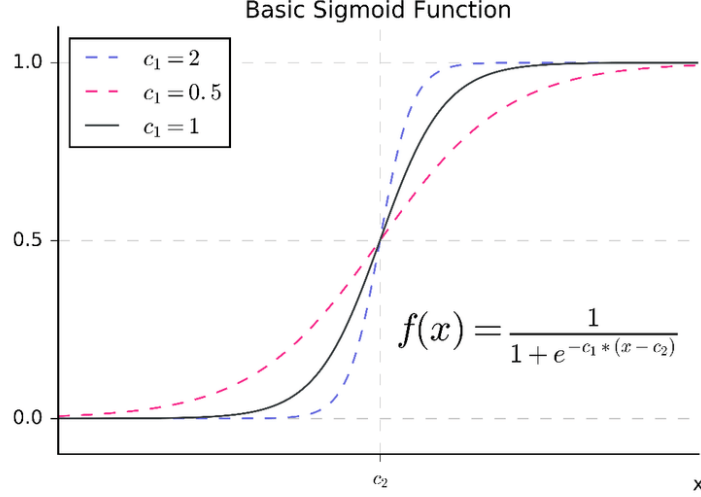


Figure 3.6: Parametric sigmoidal function [90].

Next, the introduction of a *memory saturation*, denoted as $Sat(U_{bw}(k))$ in Eq. 3.2, is based on the BW utilization $U_{bw}(k)$ as defined in Eq. 3.1. Upon analyzing Eq. (3.1), it becomes evident that as $U_{bw}(k)$ approaches 1, the memory BW saturation point is reached more rapidly. The saturation function $Sat(U_{bw}(k))$ effectively encapsulates the BW behavior observed across all three kernel categories.

$$Sat(U_{bw}(k)) = \frac{1}{1 + e^{-\alpha(U_{bw}(k) - S_p)}} \quad (3.2)$$

In Eq. (3.2), the parameter α is utilized to control the steepness of the saturation function. Through experimental testing of various values of α , it is determined that the predictive model yielded better results when α was set to 100.

The term S_p represents the *saturation point*, which corresponds to the theoretical allocation of an individual L2 memory bank per SM. In the GPU under simulation, the L2 cache consists of 24 banks serving 30 SMs, resulting in a value of $S_p = 24/30 = 0.8$ that approximates the saturation point within the L2 cache.

In order to model how much timing the kernel spend in accessing the memories, the *kernel activity intensity* factor, denoted as KAI in Eq. (3.3), is defined as the ratio between the total number of executed instructions and the number of L2 accesses, scaled down by a factor of 1000, thereby providing an estimation of the memory intensity. The KAI concept draws partial inspiration from findings by Hong et al. in [91], where they assert that "*misses-per-kilo-instruction (MPKI) is a metric for gauging the memory intensity, and as a proxy for application performance, ... MPKI estimate the performance of GPU applications more accurately*". Additionally, they note that "*performance is directly proportional to the achieved DRAM bandwidth, and is inversely proportional to the size of datum that needs to be fetched to/from DRAM in order to commit an instruction*" indicating a dependency on both MPKI and bandwidth. Further, they emphasize that "*performance is not only dependent on MPKI, but also bandwidth ... the correlation between MPKI and IPC is -44.4%. While MPKI is the number of misses that need to be served to commit 1000 instructions*" that are the others key aspect that is going to be used for the latency prediction function.

$$KAI = \frac{ExecutedInstructions}{L2Accesses} \times \frac{1}{1000} \quad (3.3)$$

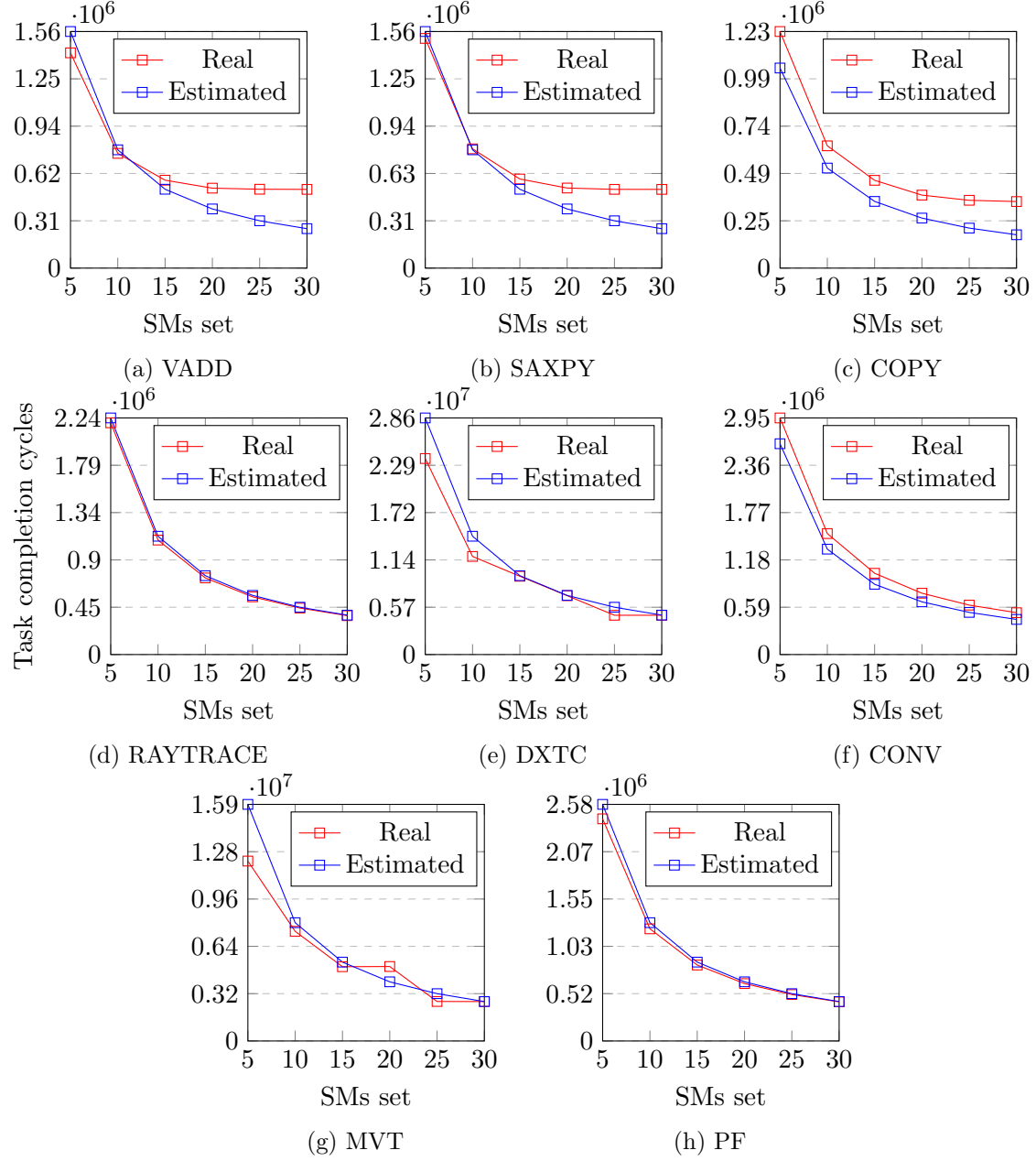


Figure 3.7: Kernel latency prediction with varying SM size using Eq. 3.4

To model the predictive function of kernel execution time (in cycles) on a reduced number of SMs, it is crucial to examine how total execution cycles vary based on the allocated number of SMs and the type of kernel, as shown in Figure 3.5. We can distinguish two distinct behaviors of SM-cycles: a linear one for non-memory-intensive and hybrid kernels, and a non-linear one for memory-intensive ones. In the linear case, the number of cycles that a single SM can execute is considered and scaled proportionally to the desired number of SMs ($\frac{N}{n}$). However, in the non-linear case, it is essential to consider memory usage (U_{bw}), KAI, and memory saturation relative to usage ($Sat(U_{bw}(k))$) to simulate the saturation point based on the number of SMs. This saturation point indicates that linear scaling is no longer possible due to memory bandwidth saturation. Therefore, a multiplicative factor $\frac{N}{(Sat(U_{bw})+1) \cdot (N - KAI \cdot U_{bw})}$ is introduced, which remains at 1 for non-memory-intensive kernels but improves the predictive function of memory-intensive kernel.

Finally, the abbreviated version of kernel latency prediction function $T_c(k, n)$ is proposed in Eq. (3.4). This function estimates the simulation cycles required for a given kernel k to finish by executing on n SMs, where n is constrained to be less than or equal to N . Here, $C_k(N)$ represents the total cycles necessary for the kernel to complete in isolation with the full set of N SMs, as illustrated in Figure 3.5. Eq. (3.4) essentially calculates a proportion by scaling the completion time based on the fraction of SMs utilized, and this proportion becomes increasingly biased as kernel memory usage rises/diminished.

$$T_c(k, n) = \frac{C_k(N)}{(Sat(U_{bw}) + 1) \cdot (N - KAI \cdot U_{bw})} \cdot \frac{N^2}{n} \quad (3.4)$$

Examining the behavior of Eq. (3.4) in comparison to the simulated execution cycles in GPGPU-Sim, as depicted in Figure 3.7, it is highlighted that the prediction tends to exhibit larger errors in the case of the three memory-intensive kernels, while it performs well in other scenarios. However, it's important to note that the errors resulting from our predictive formula, as analyzed on individual kernels and illustrated in Figure 3.8, range from underestimations of 30% to a maximum of 10% in situations where excessive memory usage is not required. In predictive models for real-time systems, establishing a margin of correctness is essential because perfect alignment between prediction and reference values is unlikely. In this results, have been set an arbitrary margin of 20%, which aligns with industry and academic practices across various application domains for estimating the worst-case execution time (WCET) [92, 93, 94]. Upon examining the absolute error produced by the predictive formula (Figure 3.8), it becomes evident that predicting the behavior of kernels with the highest bandwidth demand (VADD, SAXPY, and COPY) is more challenging. This is primarily because the formula fails to precisely capture the behavior of these three memory intensive kernels. Conversely, the remaining kernels, including the hybrid and compute-intensive ones, yield favorable results and consistently remain within the predefined 20% margin within the WCET estimation domain.

3.6 Modeling Memory Interference

In Section 3.5, has been introduced the function $T_c(k, n)$, which predicts the number of cycles required for a kernel to finish independently based on the number of SMs. Nonetheless, simultaneous execution of multiple kernels leads to alterations in latency due to their memory access conflicts.

In this section, a model is established to quantify the level of interference that may arise when two or more kernels run concurrently on separate, non-overlapping SM partitions within the GPU. This analysis encompasses two crucial factors:

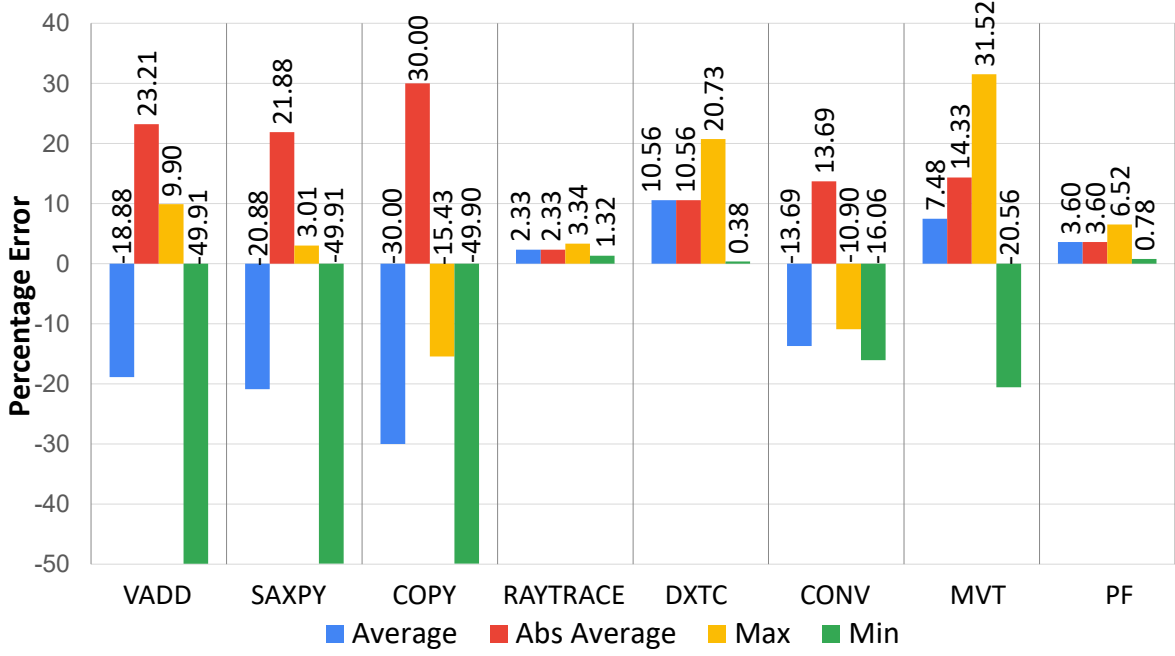


Figure 3.8: Cycles prediction errors of the predictive Eq. (3.4). Negative values are underestimations.

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 active kernels is expected to degrade as a result of interference. Additionally, it's logical to posit that as the bandwidth limit is increasingly surpassed, interference and the ensuing performance degradation will intensify. Subsequently, a set of experiments is conducted to validate these hypotheses.

3.6.1 Experiments and Analysis

To examine interference behavior, a series of seven experiments was conducted with the aim of encompassing both critical (where the total required bandwidth is less than the maximum available bandwidth) and non-critical (where the total required bandwidth is greater than or equal to the maximum available bandwidth) scenarios. Each experiment, denoted from 1 to 7, involved varying numbers of concurrent kernels, ranging from 2 to 6. The kernels assigned to each experiment were distributed among non-overlapping sets of SMs in such a way that the combined size of SM partitions for each kernel within a single experiment matched the maximum number of available SMs ($N = 30$), implying that each kernel will have its own unique subset of SMs where it can be scheduled and executing as Figure 3.3 represent.

These experimental configurations are summarized in Table 3.3, which provides an overview of the seven experiments, detailing the SM partition sizes for each concurrent kernel and their respective theoretical memory bandwidth requirements. These bandwidth requirements were computed by summing the individual memory bandwidth measurements obtained in isolation (Section 3.4.1). All kernels in the experiments run to completion.

Figure 3.9 illustrates the kernel latency deviation expressed as the ratio $\frac{\text{concurrent}}{\text{isolation}}$ for the specific set of SMs. It indicates the presence of interference, which is susceptible to variations

Table 3.3: 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

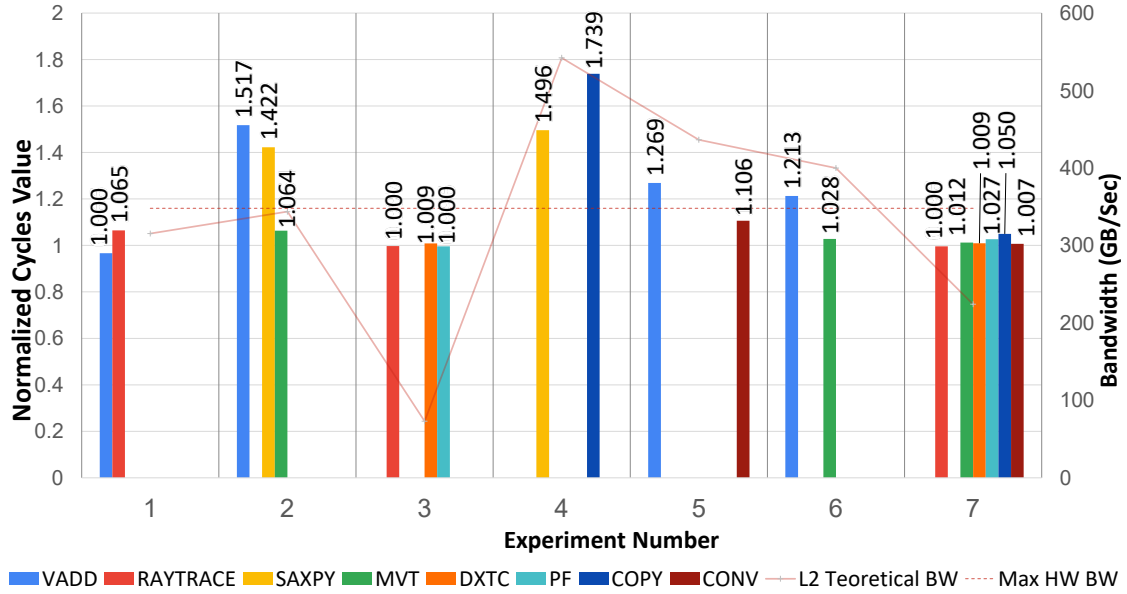


Figure 3.9: The experiments conducted. Observed kernel latency deviation, requested BW (solid line), and nominal BW (dotted line).

based on the memory access patterns of concurrently running kernels and L2 cache accesses. These experiments also highlight that the *extent of kernel latency increase is linked to the initially requested bandwidth* (solid line, right y-axis) in comparison to the nominal bandwidth (dotted line). This observation serves as the *initial foundation for the extended prediction model*.

Figure 3.10 displays the emulated execution pattern, indicating the start (first scheduling) and completion times (in cycles) for its corresponding set of kernels. Upon closer examination of this execution view, a series of time windows (distinct sections separated by vertical dotted lines), referred to as W , has been identified. We introduce a function $K(w)$, which assigns the set of kernels present in each execution window $w \in W$. For example, in the last window $w^l \in W$, only $DXTC \in K(w^l)$ is executing, while in the previous window $w^{l-1} \in W$, both $MVT, DXTC \in K(w^{l-1})$ are concurrently in operation. Additionally, we can compute the proportion of execution $P_{overlap}(k, n, w)$ for a kernel k mapped onto n within a window w , assuming $k \in K(w)$. This calculation is performed using Eq. (3.5), with $C_k(n)$ obtained from Figure 3.5. This concept of execution windows forms the second fundamental component of the predictive model.

$$P_{overlap}(k, n, w) = \frac{size(w)}{C_k(n)} \quad (3.5)$$

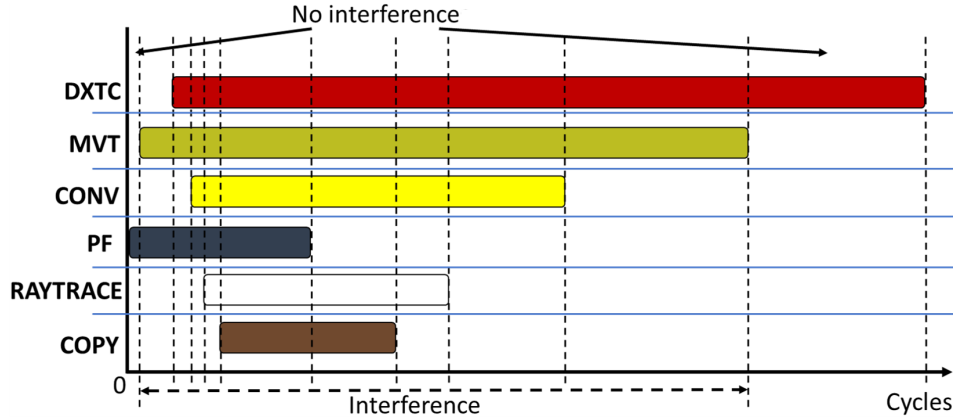


Figure 3.10: Experiment 7 overlapping time windows of PF / MVT / RAYTRACE / DXTC / CONV / COPY, 5 SMs each.

To circumvent the need for the actual execution of the kernel set, we need to rely on a set of assumptions to establish the set of execution windows, denoted as 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 3.4.2.

The formulas presented in this section have been developed drawing inspiration from various research conducted in this field [95, 96, 97]. However, until now, no one had fully captured these relationships. Nevertheless, these studies have provided essential insights for formulating the equations presented.

For each overlapping time window $w \in W$, we can calculate the additional execution burden that a kernel will experience, using the principles established in equations (3.6) and (3.7). The former imposes a penalty only when the requested bandwidth in the time window exceeds the effective bandwidth EM_{BW} , while the latter computes the interference factor $I(w)$

related to the over-requested bandwidth. If the combined bandwidth requested by the set of kernels $K(w)$ exceeds the effective bandwidth, the penalty should be adjusted accordingly. In Eq. (3.7), the terms S_p , BW_{max} , and $BW_{isolated}(k, n)$ retain the same definitions as outlined in Section 3.5.

$$\Pi(k, n, w) = \begin{cases} 1 & \sum_{l \in K(w)} BW_{isolated}(l, n) < EM_{BW} \\ Penalties(k, n, w) & \sum_{l \in K(w)} BW_{isolated}(l, n) \geq EM_{BW} \end{cases} \quad (3.6)$$

$$I(w) = MAX(1, \frac{\sum_{l \in K(w)} BW_{isolated}(l, n)}{BW_{max} \times S_p}) \quad (3.7)$$

Therefore, we introduce a function $T_c(k, n, W)$, as expressed in Eq. (3.8), which forecasts the completion latency while factoring in the interference for a kernel k running on n SMs, taking into account a set of execution windows denoted as W . For each time window $w \in W(k)$ where kernel k is active, this function aggregates the kernel's execution time along with additional penalty cycles.

$$T_c(k, n, W) = \sum_{w \in W(k)} T_c(k, n) \times P_{overlap}(k, n, w) \times I(w) \times \Pi(k, n, w) \quad (3.8)$$

Three types of penalties that a kernel k may incur are identified, as indicated in Eq. (3.9) ⁴:

- $Penalty_{SM}$, as defined in Eq. (3.10), addresses the imbalance of SM partitions among kernels executing within the same time window when kernel k is assigned to n SMs.
- $Penalty_{access}$, expressed in Eq. (3.11), quantifies the level of interference experienced by kernel k during a portion of its execution, assuming L2 accesses are evenly distributed among all the kernels present in the same time window w , as shown in Eq. (3.13).
- $Penalty_{BW}$, defined in Eq. (3.12), captures the disparity in BW demand, considering that BW is equally shared among the kernels present in the same time window w , as per Eq. (3.14).

$$\overline{Penalties} = Penalty_{SM} \times Penalty_{access} \times Penalty_{BW} \quad (3.9)$$

$$Penalty_{SM} = \frac{N}{n} \times \frac{1}{size(K(w))} \quad (3.10) \quad L2Access = \sum_{l \in K(w)} L2Access(l, w)$$

$$Penalty_{access} = \frac{L2Accesses(k, w)}{L2Access(w)} \times P_{overlap} \quad (3.11) \quad (3.13)$$

$$Penalty_{BW} = \frac{BW_{isolated}(k, n)}{Equity_{BW}(w)} \quad (3.12) \quad Equity_{BW} = \frac{BW_{max}}{size(K(w))} \quad (3.14)$$

However, it is crucial to impose constraints on the term $\overline{Penalties}$ in order to avoid unrealistic scenarios. The $\overline{Penalties}$ term represents the percentage of cycles that must be

⁴For clarity, the arguments (k, n, w) are omitted from Eq. (3.9) to (3.14).

added to the kernel's latency to account for interference. Therefore, it cannot be less than 1. Conversely, if $\overline{Penalties}$ exceeds 2, it implies that in the worst-case scenario, a single kernel running concurrently with another could be completely stalled on L2 cache accesses, with a BW demand of 0. Such a scenario is impossible to achieve with the hardware architecture used. We define $Penalties$ as shown in Eq. (3.15) and use it in the previous Eq. (3.6). To effectively manage the kernel that generates the majority of access requests and accommodates more requests to the memory controller, a single kernel has to be selected for each time window. The elected kernel, chosen for each time window based on the kernel bandwidth demand, is considered privileged and experiences fewer penalties due to its high traffic BW demand. The elected kernel has its penalties set according to the scenario where the average penalties ($\overline{Penalties}(k, n, w)$) are less than 1.

$$Penalties = \begin{cases} \min(Penalties_{BW}(k, n, w), I(w)) & \overline{Penalties}(k, n, w) < 1 \\ Penalty_{MPD}(k, n, w) & \overline{Penalties}(k, n, w) \geq 2 \\ \overline{Penalties}(k, n, w) & Otherwise \end{cases} \quad (3.15)$$

In order to leverage the $2x$ term in $Penalties$, a Max Penalty Density ($Penalty_{MPD}$) has been defined in Eq. (3.16a). The concept of the request density of a kernel k , DR_k Eq. (3.16b), and the maximum density value done in a certain time window w , is given by the term MDA_w , Eq. (3.16c).

$$Penalty_{MPD}(k, n, w) = \frac{DR(k, n, w)}{MDA(w)} \quad (3.16a)$$

$$DR(k, n, w) = \frac{L2Access(k, w)}{T_c(k, n)} \quad (3.16b)$$

$$MDA(k, n, w) = \max_{k \in K(w)} \left(\frac{L2Access(k, w)}{T_c(k, n)} \right) \quad (3.16c)$$

3.6.2 Comparison with a Worst-interference Method

Most of the related studies focus on predicting the efficiency [98], in contrast, our objective is to forecast execution times under interference conditions. We can compare the approach to a predictive model that accounts for the worst-case interference during concurrent kernel executions, similar to the methodology in [98]. This model incorporates a margin added to the kernel latency in isolation, resulting in a prediction that considers interference.

Consider a set of kernels, each with distinct in-isolation latencies. The kernel denoted as X has the longest latency and encounters interference from other kernels until the second longest one completes its execution. Afterward, kernel X runs in isolation and no longer experiences interference. In our naive approach, any kernels with latencies shorter than or equal to the second longest one encounter maximum interference since they never have a time interval during which they execute independently on the GPU. This behavior is illustrated in Figure 3.12, where DXTC is the only kernel with a time interval that executing alone and avoid interference.

In Figure 3.9, it is evident that the COPY kernel from experiment 4 experienced a significant slowdown, reaching a maximum of 73%. In the naive approach, an arbitrary value of 73% is applied to all kernels when they are fully present and during the portion of execution within the maximum interference time window.

It's important to highlight that the chosen 73% represents a more conservative margin compared to the one employed in [98] (over 100%). However, this conservatism poses a

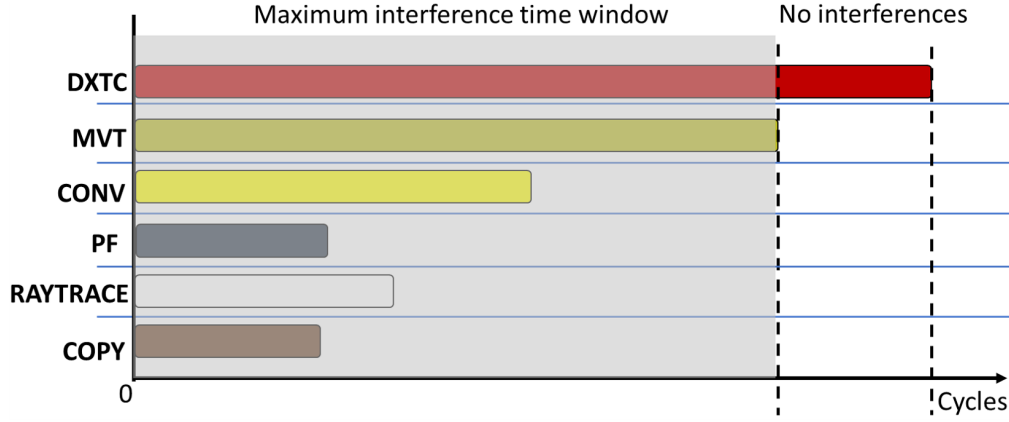


Figure 3.11: A set of concurrent kernels with the maximum interference time interval in the light grey overlay.

challenge, as a larger margin in a simplistic approach usually leads to a greater prediction error.

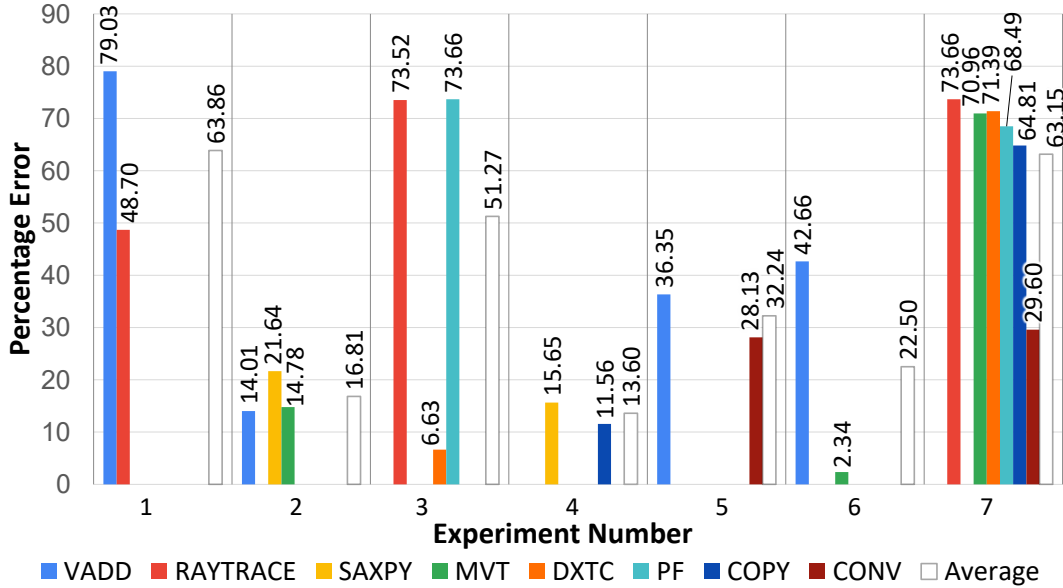


Figure 3.12: Naive latency estimation, kernel cycles increased by a factor of 73%.

In Figures 3.12 and 3.13, the x-axis corresponds to the experiments listed in Table 3.3, while the y-axis indicates the error percentage. A positive error percentage indicates overestimation, whereas a negative value signifies underestimation. The naive method consistently results in overestimation, while the created approach demonstrates a more balanced performance. However, it's worth noting that the average error for the naive method is quite high at 42.38%. In contrast, the created predictive model outperforms the naive method, achieving an absolute average error of only 5%.

3.6.3 Latency and Interference Prediction Evaluations

To validate the predictive model and estimate its error, 100 additional groups of kernels were randomly generated, in addition to the initial 7, with group sizes ranging from 2 to 6, resulting in a total of 107 experiments. Within each group, the number of SMs used by each kernel was

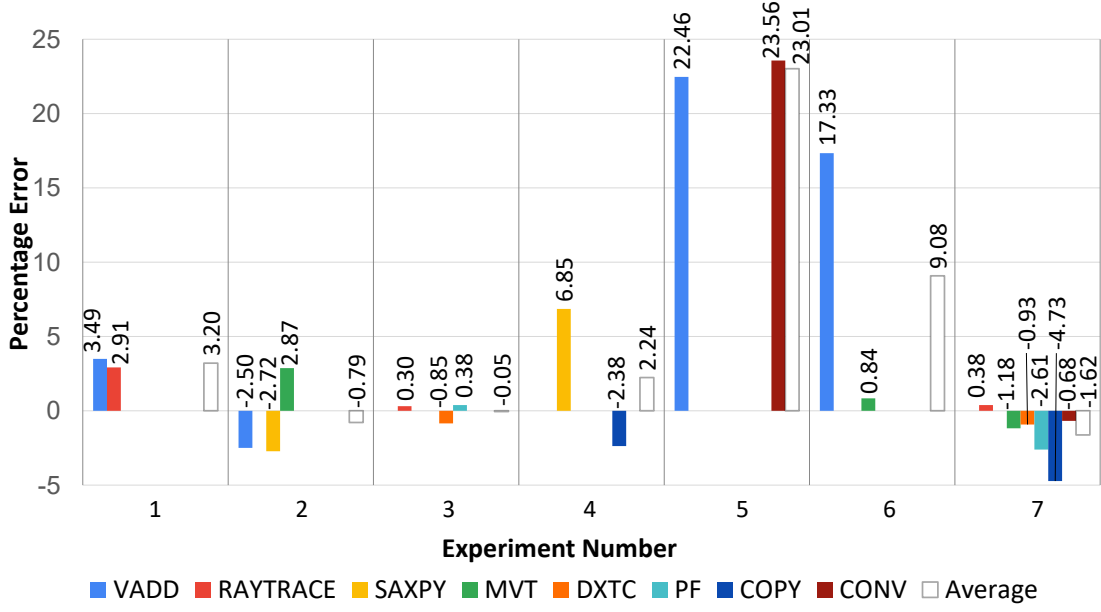


Figure 3.13: Kernel latency estimation prediction error, using Eq. (3.8), on experiments from Table 3.3.

randomly determined from the set 5, 10, 15, 20, 25, with the condition that the total number of SMs used in the group did not exceed the total available SMs (30).

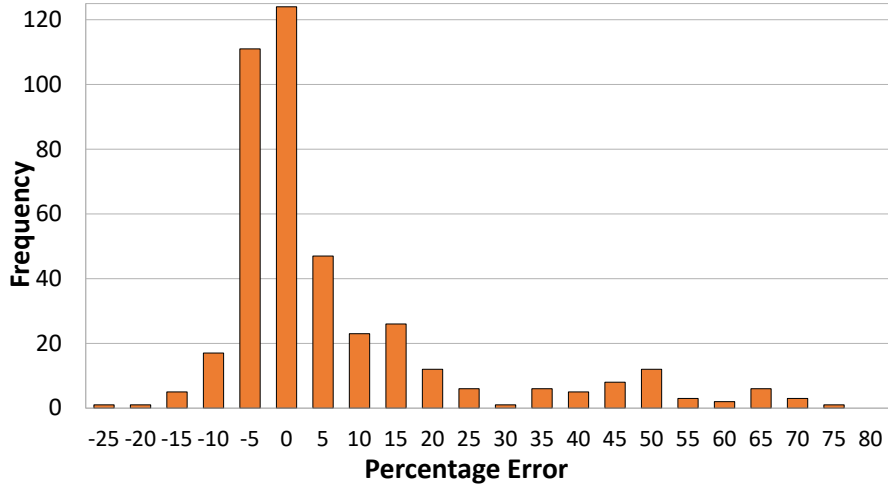


Figure 3.14: Kernel latency prediction error, using Eq. (3.8), of the 107 kernel sets.

Figure 3.14 illustrates the error distribution of the predictive model in Eq. (3.8). To gain comprehensive insights into the error distribution, we can calculate the error as the difference between the emulated cycle and the estimation provided by the formula. On average, the formula exhibited a 10.7% error rate with a tendency to overestimate, along with a variance of 2.71%. As depicted in Figure 3.14, occasional underestimations (negative values) and overestimations (positive values) were observed in the prediction process. It's worth noting that in real-time applications, especially in scheduling scenarios where timing is critical, it is generally more advantageous to overestimate rather than underestimate the execution time required by a kernel to complete its tasks. The final results of the 107 experiments, in which

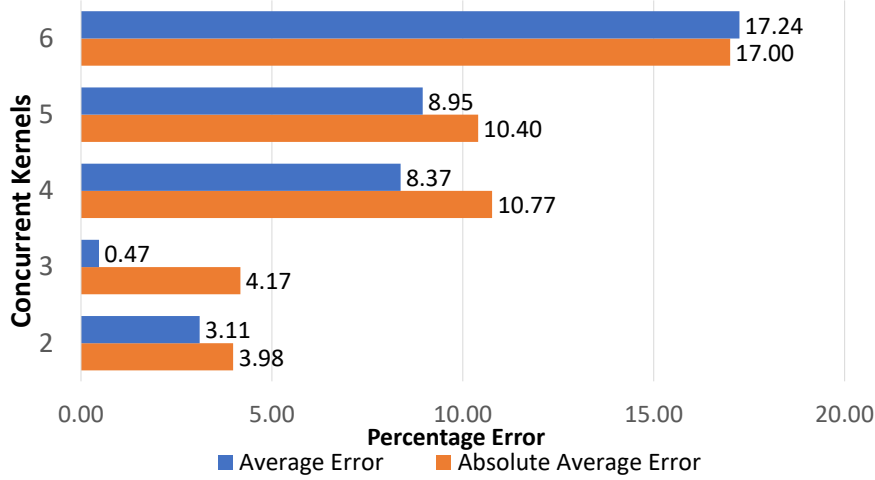


Figure 3.15: Absolute and average kernel latency prediction error per number of concurrent kernels using Eq. (3.8). All averages are positive, due to the weight of the overestimation.

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%.

Going more in-depth, by analyzing the individual different sub-classes error obtained in Figure 3.15, we can recover more information. Based on the normal average (not the absolute one), the derived formula consistently overestimated the execution times across all experiment classes (Figure 3.15). Specifically:

- In the sub-classes with 2 and 3 concurrent kernels, the formula overestimated the execution of each kernel by an average of 3.11% and 0.47%, respectively. The absolute average error for these sub-classes was 3.98% and 4.17%, indicating a high level of accuracy in predicting the cycle counts compared to the emulation engine.
- For the two sub-classes with 4 and 5 kernels, the formula exhibited higher overestimation, with average errors of 8.37% and 8.95%, respectively.
- In the last sub-class with 6 kernels, the formula yielded the largest error of 17.24%. These sub-classes with higher kernel counts turned out to be more unstable because more kernels needed access to the same L2 banks, generating more interference than initially estimated by our formula.

The results of the predictive model on different sub-classes consistently fall below the absolute error threshold of 20%, which is commonly used in different domains for predicting the WCET, as previously discussed in Section 3.5. These results demonstrate the effectiveness of the proposed predictive model.

3.7 Bandwidth Prediction

In the preceding sections, a predictive model capable of *estimating the number of execution cycles needed for a single kernel's completion*, whether it operates in isolation or concurrently with other kernels, has been successfully developed. To fully refine the model, the next goal is

to accurately forecast the memory bandwidth requirements of a kernel when it is mapped onto a set of SMs in isolation. This eliminates the necessity of empirically determining memory bandwidth by repeatedly running a kernel on various SM cluster sizes. The formulas in this section are empirically derived, utilizing the architecture’s parameters and drawing inspiration from the same concept utilized in 3.5.

To accomplish this objective, a model for the *BW measured* during isolated execution with the complete set of SMs is established, denoted as $BW_{isolated}(k, N)$. Additionally, a threshold value, $BW_{threshold}$, is introduced, calculated as the product of BW_{MAX} and S_p (described in Section 3.5). This threshold, $BW_{threshold}$, enables the dynamic switching between linear and nonlinear prediction modes, allowing for adaptive behavior, similar to the Section 3.5. The definition of $BW(k, n)$ as the *BW requirement* for a specific kernel k when it is executed on a different set of SMs (n) is detailed in Eq. (3.17).

$$BW(k, n) = \begin{cases} \frac{BW_{isolated}(k, N)}{N} \cdot n & BW_{isolated}(k, N) < BW_{Threshold} \\ EM_{BW} \cdot (1 - e^{(-\frac{n}{MAX(1, Miss_{banks})})}) & BW_{isolated}(k, N) \geq BW_{Threshold} \end{cases} \quad (3.17)$$

The EM_{BW} , quantified at 330 GB/s, represents the maximum bandwidth observed during the initial phase analysis discussed in Section 3.4.1. On the other hand, the term $Miss_{banks}$ is defined as the discrepancy between the total number of available SMs, denoted as N , and the total count of L2 banks specified by the underlying hardware architecture. In this particular phase, the same configuration detailed in Table 3.1 was utilized. In this configuration, the L2 cache comprises 24 banks, hence, the $Miss_{banks}$ is set to 6. This choice of parameters aims to model the hardware constraint where each SM lacks its own dedicated L2 bank, thus subject to interference from other SMs.

3.7.1 Bandwidth Prediction Results

The prediction error is determined by calculating the ratio between the predicted bandwidth and the real value, subtracting 1, and then multiplying the result by 100 to obtain a percentage value. The comprehensive error, as determined by Eq. 3.17, averages at 4.60%, with an absolute average of 11.23%. The results for L2 cache bandwidth predictions are graphically depicted in Figure 3.16, while the associated errors are illustrated in Figure 3.17.

The created method tends to overestimate the L2 bandwidth requirements, particularly for memory-intensive kernels like VADD, SAXPY, and COPY when a smaller set of SMs is employed. Conversely, predictions for hybrid and computational kernels align closely with the actual requirements, albeit occasional underestimations persist. Figure 3.18 summarizes the overall bandwidth prediction error, its average, and variance, encompassing individual SMs sets ranging from 5 to 30. On average, the prediction error ranges from 0.3% to 19%, with the highest prediction error occurring when the SMs set is 5, resulting in a 19% prediction error.

3.8 Related Work on Estimation of Malleable Tasks

There are limited works in the literature that explicitly address performance estimation based on memory interference. One notable recent work is HSM [99], which calculates performance degradation in scenarios where multiple kernels are co-running. This degradation is quantified as the ratio between instructions per cycle in isolated and shared modes across various kernel-SM partition configurations. Another significant contribution in this domain is the DASE (Dynamical Application Slowdown Estimation) model proposed in [95]. In this work,

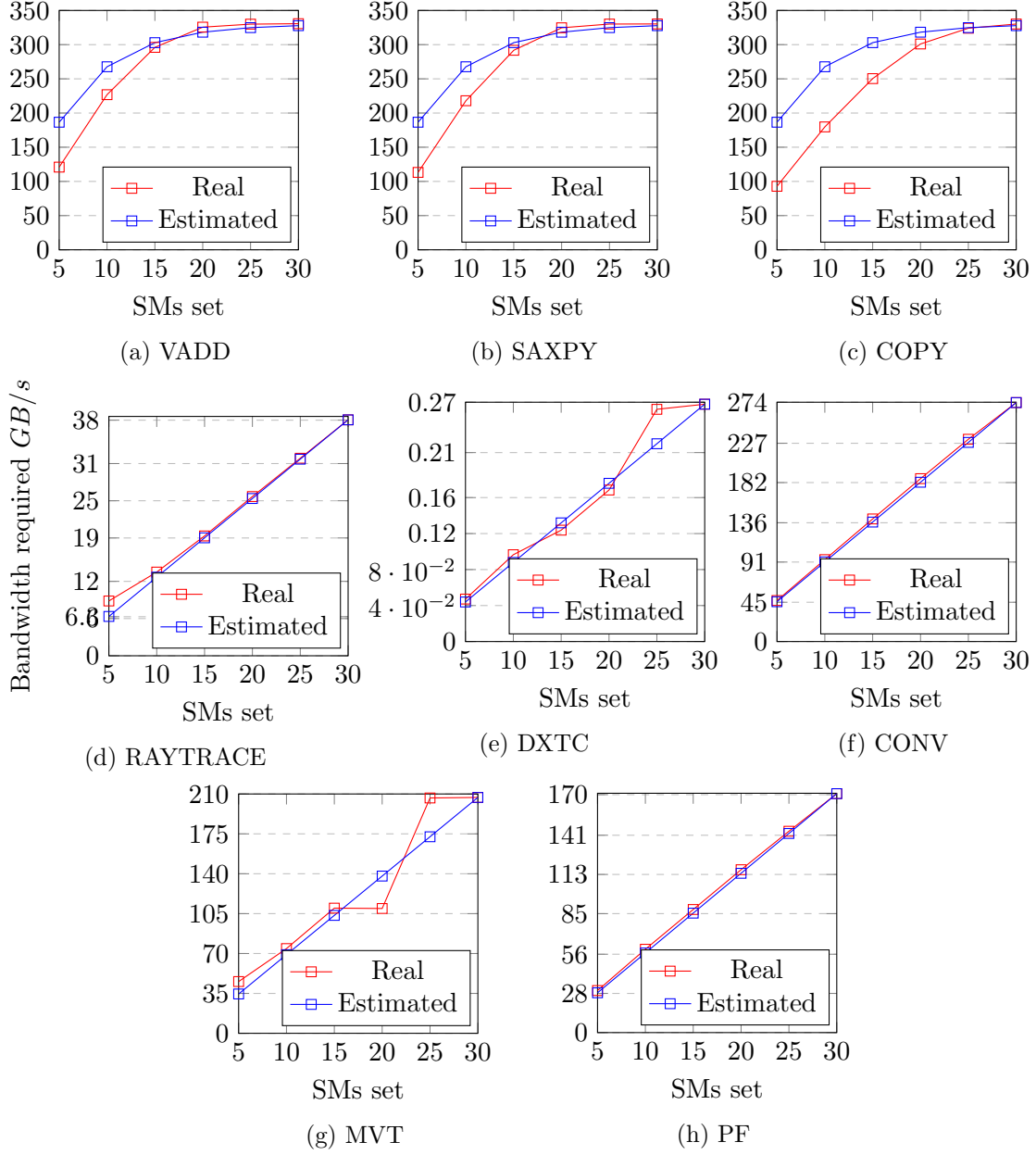


Figure 3.16: Predicted BW for each kernel.

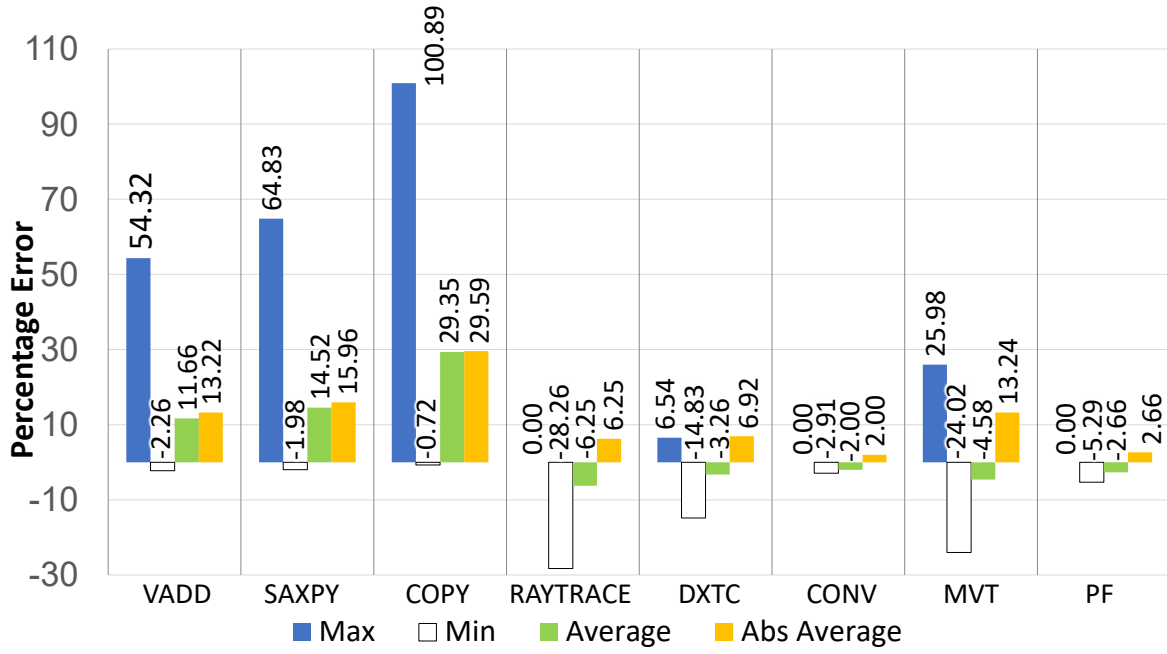


Figure 3.17: Min, max, avg, and absolute avg BW prediction error, using Eq. (3.17).

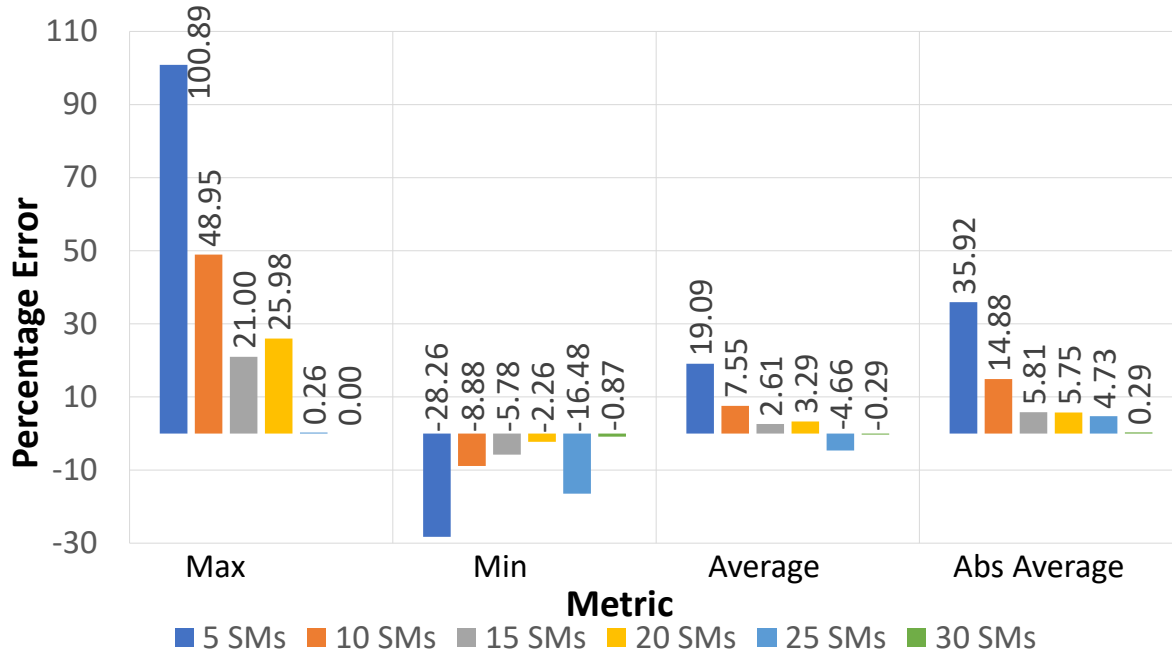


Figure 3.18: BW avg and variance prediction error of Eq. (3.17) varying the SMs size.

the authors estimate the slowdowns induced by shared resources in concurrent GPGPU kernel execution and provide a performance estimation model tailored for single GPGPU applications. Similar to the approach created in this thesis, their research involves a profiling phase that considers scenarios with two concurrently executing kernels. However, the solution of this chapter extends beyond this by examining a wider range of concurrent kernel combinations, including scenarios with up to six co-running kernels. Another related work is presented by Hong et al. in [11], where they propose a model for estimating execution cycles in GPU architectures for single kernels. They assume that memory bandwidth is evenly distributed among active SMs at runtime. However, in this chapter, along with other cited studies, highlights that such equal bandwidth distribution is not always the case, and bandwidth unfairness can occur depending on various metrics characterizing concurrent kernels [91]. Furthermore, the concept of unfairness resulting from the memory intensity of concurrent kernels is also explored in the work of Adwait Jog et al. in [96]. They analyze the percentage bandwidth utilization of different applications, meanwhile, in this chapter is highlighted the significance of the bandwidth utilization term in predicting memory interference and introduces distinct terms to model it.

In the study conducted by Hongwen Dai et al. [97], they emphasize the advantages of balancing memory access between two kernels, one memory-intensive and the other compute-intensive. They achieve this by constraining the number of memory instructions issued by both concurrent kernels to ensure equitable memory access requests and prevent memory pipeline stalls. This work highlights a critical aspect that aligns with our research in Section 3.6: the potential dominance of a memory-intensive kernel in memory pipeline usage within an SM, leading to notable performance degradation for compute-intensive kernels whose memory requests experience delays and are not serviced promptly. Various researchers have proposed hardware schemes and techniques to enhance the exploitation of concurrent kernel execution on GPUs. For instance, Xu et al. [100] introduced an intra-SM slicing approach to leverage underutilized memory throughput through improved intra-SM partitioning for two different kernel CTAs. Thomas et al. applied the roofline model to categorize applications as compute-bound or memory-bound based on their operational intensity and implemented distinct CTA scheduling modes and core partitioning to enhance performance [101]. In their work, they also highlighted the potential performance degradation when memory-bound tasks were assigned by a small SM count. In this thesis a similar analysis is detailed; more specifically by measuring the memory bandwidth requirements of kernels mapped to different GPU partitions. Adriaens et al. [102] proposed spatial multitasking to group SMs into sets that can run different kernels (up to four) to maximize application speedup. Ukidave et al. [103] explored real-time support for adaptive spatial partitioning on GPUs and emphasized the importance of L2 cache in this context. Aguilera et al. [104] demonstrated the unfairness of spatial multitasking and suggested a fair resource allocation strategy for optimizing both performance and fairness. Emir C. Marangoz et al. [105] illustrated interference in L2 cache during co-running applications, as i did in this thesis, and proposed a novel architecture for managing bandwidth reservation for single-kernel concurrent execution. In contrast to many previous research efforts mentioned in this section, the primary objective was to provide a predictive model for memory interference in scenarios involving GPU SM spatial partitioning. This model can be utilized by a memory-aware scheduler [106] to dynamically allocate the appropriate SM partitions to each kernel, ensuring that each kernel meets its temporal constraints. The outlined sequence of operations for modeling interference effects in shared memory hierarchies can be adapted to various GPU architectures. Notably, the approach presented in the previous sections does not impose constraints on the number of concurrently executing kernels on the GPU, differentiating it from previous research.

Additionally, it is worth mentioning the recent release of NVIDIA MiG (Multi Instance

GPU) technology⁵. MiG offers enhanced isolation among GPU partitions by allocating cache partitions. While this method is highly effective in mitigating memory interference, it is currently available on a limited subset of high-end hardware. Furthermore, there are restrictions on the flexibility of defining and reconfiguring partitions dynamically.

3.9 Conclusion

This chapter presents the developed scheduler extension for GPGPU-Sim, which enables the investigation of a partitionable GPU's behavior and the concept of dynamic partitioning within a GPU by allowing the assignment of SMs to different kernels. Two profiling phases were conducted: one in isolation and another with concurrent kernels. A set of kernels was used to conduct these phases and gather more information about each kernel's behavior. Each kernel was categorized as memory intensive, compute, or hybrid based on the required bandwidth demand.

It was highlighted that memory intensive kernels reach a saturation point in DRAM bandwidth demand, where scaling the number of SMs does not decrease their execution time, while the other two categories of kernels scale linearly based on the number of assigned SMs. In Section 3.5, a model was introduced capable of estimating execution latency on a subset of SMs considering only the timing and bandwidth demand of a single execution with the entire set of SMs available on the emulated architecture.

The predictive formula ranges from underestimations of 30% to a maximum of 10% in scenarios where excessive memory usage is not required. Upon examining the absolute error produced by the predictive formula with an arbitrary margin of 20%, which aligns with industry and academic practices for estimating worst-case execution time (WCET) across various application domains, the model remains within the 20% WCET estimation domain.

To investigate interference behavior when multiple kernels execute concurrently on separate (non-overlapping) partitions, a series of seven experiments was conducted. These experiments aimed to cover critical scenarios, where the total required bandwidth is less than the maximum available, as well as non-critical scenarios, where the total required bandwidth equals or exceeds the maximum available.

In Section 3.6, a model was developed to quantify the level of interference that may arise when two or more kernels run concurrently on distinct partitions within the GPU. This model is capable of predicting kernel execution latencies and considers various factors to quantify interference levels. These factors include penalties based on execution overlapping windows within kernels, SM partition imbalances among kernels, assumed even distribution of L2 cache accesses and differences in bandwidth demand. The kernel with the highest bandwidth demand will saturate the memory controller by accommodating more requests than other kernels executing concurrently. Additionally, the model assumes behavior specific to this single kernel and selects, for each time window, the kernel generating the majority of access requests and accommodating more requests to the memory controller, thereby minimizing penalties for it.

The model's absolute average error in the initial seven experiments is 5%, outperforming a naive method that applies a 73% increase to each individual kernel, resulting in a 42.38% absolute average error. To validate the predictive model and estimate its error, 100 additional kernel groups were randomly generated, supplementing the initial 7 groups, with group sizes ranging from 2 to 6. This resulted in a total of 107 experiments. On average, the model exhibited a 10.7% error rate with a tendency to overestimate, accompanied by a variance of 2.71%.

⁵<https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html>

In Section 3.7, a model predicting the memory bandwidth requirements of a kernel in isolation execution when mapped onto different SM sets has been developed. This model eliminates the need for empirical determination of memory bandwidth through repeated kernel executions on various SM cluster sizes. The model's error averages at 4.60%, with an absolute average of 11.2%.

Chapter 4

Machine Learning Techniques for Understanding and Predicting Memory Interference in CPU-GPU Embedded Systems

4.1 Introduction

In contemporary contexts, heterogeneous embedded boards have become prevalent in various application domains, including the automotive industry, Internet-of-Things (IoT) systems, and industrial robotics. These platforms are characterized by their combination of multicore CPUs collaborating with highly parallel accelerators like GPUs and/or other Application Specific Integrated Circuits (ASICs). However, within the realm of research literature, there exists a noticeable gap in understanding the timing behavior that arises when multiple accelerators on the same board access shared memory hierarchies, such as the system DRAM. Previous research has underscored the significance of memory interference as a substantial threat to both performance and predictability [107, 108, 109]. Furthermore, device manufacturers often do not provide documentation that contains the necessary architectural details required by system engineers to comprehend and mitigate the impact of memory interference on real-time workloads.

This chapter is dedicated to analyzing memory interference behavior within an embedded platform where multiple CPUs contend with an integrated GPU for access to the system DRAM. The specific focus here is to comprehend and investigate the behavior of GPU-side applications, often referred to as kernels, while memory-intensive applications run on the CPU side, sharing the main memory accessible to both processing units. For this purpose, it has been utilized the Nvidia Jetson AGX Xavier embedded board, a well-known platform [110].

The Nvidia Jetson AGX Xavier board offers a robust software development kit (SDK) for GPU-based application development, complete with various profiling tools designed to assist application developers. These profiling tools enable the examination of hardware counters and kernel execution metrics, providing insights into the characteristics and behavior of GPU kernels. In this context, the objective is to uncover correlations and valuable insights regarding the relationship between these metrics and the slowdown experienced by GPU kernels as the number of concurrently executing CPU tasks accessing DRAM, each with varying memory requirements, varies. This exploration serves as the foundation for developing methodologies capable of quantifying CPU memory interference magnitude, accounting for GPU kernel characteristics, and offering predictions regarding GPU kernel execution time degradation due to

DRAM interference.

The contribution of this chapter lies in the development of methodologies for creating predictive models capable of estimating the interference that a GPU kernel may encounter. This is achieved through the utilization of a combination of well-established machine learning (ML) techniques, which include Support Vector Machines for Regression (SVR) [111], Random Forest Regressor (RFR) [112], and Deep Neural Networks (DeepNN) [113]. Utilizing these techniques in a data-driven approach allows us to identify the key factors involved in memory interference and subsequently predict kernel latency degradation. This predictive capability is crucial for accurate response time analysis [114, 115].

This chapter is structured as follows:

Section 4.2 covers the related work. Section 4.3 offers a brief overview of the architectural characteristics of a Jetson AGX Xavier, outlines the Nvidia CUDA API, and describes the experimental environment setup. Section 4.5 introduces the machine learning methodologies applied in this research to predict the GPU kernel’s slowdown based on its baseline metrics. Section 4.6 examines the most critical metrics resulting from the training phases for each ML technique. Finally, Section 4.7 provides concluding remarks and suggests potential avenues for future research. The work presented in this chapter is derived from my publication [116], which was featured in the 2023 IEEE 29th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA).

4.2 State of the Art

Interference on shared memory resources has been previously observed in various contexts, including multicore CPUs [117, 118, 119], GPU computing clusters [120, 121], and integrated SoCs involving both CPU and GPU [109, 107, 13].

Regarding the latter issue, several approaches have been proposed to mitigate the impact of memory interference on both performance and predictability. Ali et al. [122] introduced BWLOCK++, a mechanism implemented on Nvidia TX2 that allows bandwidth reservation for GPU kernels, reducing CPU memory-intensive activity by employing software throttling. Conversely, [123] presented a similar mechanism where software throttling is used to protect CPU tasks from GPU memory activity, aiming to mitigate memory interference. Houdek et al. [124] investigated these concepts through the implementation of hardware throttling. They conducted an analysis of the memory controller behavior in an Nvidia TX1 using hardware counters and subsequently introduced a hardware throttling mechanism for bandwidth regulation based on PREM [125] memory scheduling policies. While these research efforts in characterization and mitigation have enabled us to quantify memory interference and design suitable schedulers, there is, to the best of the knowledge, no prior work specifically focused on understanding and developing a predictive model for CPU-GPU interference based on kernel characteristics. Additionally, earlier research predominantly assumed that requested memory bandwidth is the most effective predictor for interference effects. With a more comprehensive characterization of GPU kernels, it can be used to substantially enhance the understanding of interference. To achieve this objective, in this thesis a leverage fine-grained kernel profiling and machine learning (ML) methodologies have been used. Previous work has also explored the use of ML approaches to derive predictive models for latency deterioration caused by memory conflicts [126, 127, 128, 129, 130]. For example, Saeed et al. [126] introduced a mechanism capable of predicting the execution time of two concurrently running applications on a multi-core processor. Their predictive model relies on hardware performance events selected based on the Spearman correlation coefficient. In the context of predicting and mitigating multicore CPU interference, Mishra et al. [127] employed linear feature selection to identify relevant

metrics. Kim et al. [128] employed a Random Forest regression model to predict interference using data collected during an offline profiling phase.

Regarding GPU kernels, Ayub et al. [129] delve into interference detection during concurrent kernel execution on discrete GPU devices, employing various ML techniques. Their primary objective is to assess the suitability of two kernels for concurrent execution. However, their dataset feature selection process is relatively coarse-grained, relying solely on four data items related to the kernels' launch configurations and register pressure. In contrast, the profiling tools available on Nvidia embedded boards empower users to collect metrics, counters, and other execution details with much finer granularity. These tools are crucial instruments to leverage for the purposes of this chapter. A similar approach has been extensively applied by Bramley et al. [130] in a different context. They conducted an in-depth analysis in the field of road vehicles and functional safety to investigate whether bit faults in SDRAM lead to computational errors in GPU kernels. They introduced the concept of *Liveness*, defined as the duration during which data was last written and used throughout the entire kernel execution period, to assess the vulnerability of GPU memory and potentially calculate its failure rate. To achieve this, they trained a neural network to predict the Liveness of a kernel, utilizing 147 performance metrics extracted from a kernel set using the `nvprof` tool.

The work of Bramley et al. inspired the research presented in this chapter, although in a distinct context focused on predicting CPU-GPU memory interference.

4.3 Background

This section use the Nvidia terminologies used in Section 2.2.1 of Chapter 2 and introduces a summary of the most important architectural characteristics of a Nvidia Jetson AGX Xavier, the used tool and the selected environment for the experiments.

4.3.1 Nvidia Jetson AGX Xavier Architecture

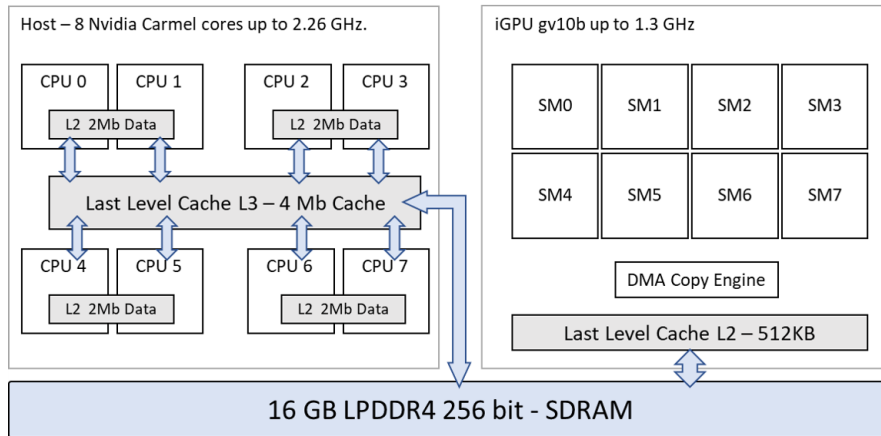


Figure 4.1: The architecture of Nvidia Jetson AGX Xavier, with a particular focus on the upper part of this figure, illustrates the connection between the CPU complex and the integrated GPU.

The Nvidia Jetson AGX Xavier represents a recent addition to embedded platforms, featuring a custom System-on-Chip (SoC) architecture that amalgamates various processing units and accelerators, delivering substantial performance while maintaining a relatively low power consumption profile. Within its internal components (as illustrated in Figure 4.1), the

platform encompasses a 64-bit ARM processor boasting an 8-core CPU cluster, denoted as Carmel, capable of clock speeds of up to 2.26 GHz. Furthermore, it incorporates a Volta integrated GPU (iGPU) alongside other application-specific accelerators. The CPU complex is a customized processor designed in accordance with ARM-v8.2 specifications. Internally, the processor cores are grouped into pairs, each cluster sharing a 2 MB L2 cache. Additionally, these clusters collectively share a 4 MB L3 victim cache. The iGPU is equipped with access to a 512 KB Last Level Cache (LLC) L2 cache. As visually depicted in Figure 4.1, both the CPU and GPU units share the same system RAM, encompassing 16GB of LPDDR4 memory. This shared memory hierarchy serves as the focal point of the experiments, despite the presence of other accelerators, such as the DLA (Deep Learning Accelerator) and PVA (Programmable Vision Accelerator), which also share this level of memory.

4.3.2 The profiling tool - NCU

The computational workloads, often referred to as *kernels*, are executed on the GPU after being divided into smaller tasks that can be concurrently processed. These tasks are then distributed across the Streaming Multiprocessors (SMs) of the GPU. SMs are processing clusters designed to handle blocks of threads, with these threads being further organized into *warps*, which constitute the smallest scheduling units within a GPU, adhering to the Single Instruction, Multiple Data (SIMD) paradigm.

In this chapter, the Nvidia Nsight Compute utility (**ncu**) is employed as a robust performance analysis tool tailored for CUDA application developers. This tool offers an extensive array of performance metrics and kernel execution statistics, empowering developers to pinpoint performance bottlenecks and fine-tune their code for optimal efficiency. By utilizing **ncu**, access to performance metrics, GPU hardware (HW) performance counters, and other execution statistics is granted, allowing for the creation of comprehensive profiles that delineate individual kernel characteristics. **ncu** is a relatively recent addition to the CUDA SDK, as its predecessor, the well-known **nvprof**, has recently been deprecated [131]. In comparison to its predecessor, **ncu** offers the capability to collect a considerably larger number of metrics. Furthermore, it autonomously computes the maximum, minimum, and average values for each metric during an offline profiling phase. Whenever applicable, the tool also automatically calculates the sum of observed measurements or percentages. **ncu** boasts an extensive repertoire of nearly 140,000 metrics, yet users have the flexibility to swiftly select or filter metrics of interest based on their interpretations of metric names. The tool uses the following naming convention:

unit_-(subunit?)-(pipestage?)-(interface)_qta_-(qualifiers?)

While **unit** and **subunit** enable the specification of the GPU's physical or logical component being profiled, **pipestage** pertains to the pipeline stage. Furthermore, **qta** denotes the dimensional units, and **qualifiers** serve to apply supplementary filters or predicates to the counters. The primary unit metrics are delineated as **counter_name.{sum,avg,min,max}**, permitting the calculation of summation, averages, minimums, and maximums. These metrics encompass additional aspects like ratios, throughput, and various *roll-up quantifiers*. A noteworthy set of counters falls under the category of **Cycle Metrics**, offering insights into the number of cycles within a specified unit clock domain. These metrics encompass elapsed cycles, active cycles (representing periods when the unit was actively processing data), stalled cycles, and more.

4.4 Setup and First Analysis

The initial profiling phase encompasses 14 distinct kernels with diverse computational and memory demands, spanning from fundamental operations to considerably more intricate programs. These kernels originate from both internal implementations and established benchmark suites. Initially, kernels were classified based on their DRAM bandwidth requirements, leading to the identification of three categories: memory-intensive, hybrid, and compute-intensive. Kernels with a bandwidth demand surpassing 70% are categorized as memory-intensive (M), those falling below 10% are designated as compute-intensive (C), while the others are classified as hybrid (H). Although this table resembles Table 3.2 in Section 3.4, it provides additional information regarding the bandwidth demand of the architecture under analysis and additional kernels. Additional information about the kernels is presented in Table 4.1. This initial kernel selection covers a broad input space: some of the kernels experience interference due to their high memory bandwidth requirements, while others are less affected. This diverse input space is representative of real-world applications and enhances the accuracy of the interference prediction model, even before estimating latency degradation. A larger set of kernels in the input space would increase the estimation confidence. However, isolating a kernel from its context (i.e., its surrounding CPU application) is a non-trivial task. Nonetheless, this isolation is essential for studying the effect of memory interference on kernel execution.

During this first phase, the analysis centers on the variations in metrics, which are read from hardware counters, as individual kernels execute while the intensity of CPU-generated interference varies. These metrics are acquired through the previously introduced `ncu`, which is initiated via the command line in conjunction with the test application responsible for dispatching the kernel set described earlier.

In order to generate variable CPU-side memory interference, an open-source software designed for heterogeneous SoCs has been set up, *HeSoC-mark* [13]¹ (details in Section 4.4.2). The Nvidia board under test is configured for maximum performance (MAXN).

Table 4.1: Kernels used for analysis on Jetson AGX Xavier architecture.

Kernel Name	Data Size In (MiB)	Data Size Out (MiB)	Bandwidth Demand (%)	Classification (Memory, Hybrid, Compute)	Origin
VADD	40 * 2	40	85.92	M	Synthetic / In House Implementation
SAXPY	40 * 2	40	83.55	M	Synthetic / In House Implementation
COPY	40	40	80.07	M	Synthetic / In House Implementation
RAYTRACE	-	1.5234	1.02	C	In House Implementation
DXTC	0.003 + 0.5	0.25	13.70	H	Cuda Samples
CONV	0.0001 + 40	40	21.34	H	In House Implementation
MVT	64 + 0.01 * 4	0.01 * 2	60.38	H	Polybench [88]
PF	0.5 + 255.5	0.5	50.73	H	Rodinia [51]
DOITGEN	16	16	85.12	H	Polybench
HOTSPOT	8 * 2	8	36.39	H	Rodinia
NBODY	4	2	27.09	H	Adapted from RoCm HIP Samples [132]
HISTO	24.3	0.003	55.93	H	In House Implementation
ATAX	262.14	0.032	76.61	M	Polybench
BFS	49.14	8.19	43.88	H	Rodinia

4.4.1 `ncu` and Kernels' Performance Metrics

The `ncu` tool empowers us to delve into the intricacies of kernel behavior through GPU-side hardware counters. Henceforth, in the continue of this thesis, the term used for these counters will be metrics. These metrics allow the system engineer to perform analysis on both compute and memory kernels' behaviours with respect to their completion latencies.

¹<https://git.hipert.unimore.it/mem-prof/hesoc-mark>

Given the vast number of `ncu` profilable metrics, in this part of the thesis, we focus on 45 metrics out of the approximately 140.000 available. This decision was made to prevent profiler overhead from affecting kernel execution times and to avoid potential board crashes when collecting an excessive number of metrics during a single profiling run. The initial filtering process followed the naming convention explained in Section 4.3.2. It centered on three main macro categories: the first category pertains to GPU memory accesses (L2 and DRAM, denoted by metrics starting with *lts*); the second category encompasses metrics related to compute instructions (indicated by those starting with *gr*, *gpu*, *smsp_inst*, or *_cycles*); and the final category provides a summary of branch divergence (initiated by metrics starting with *smsp_warp*). These categories were chosen based on their alignment with the principal components for GPGPU performance prediction and optimization, as indicated by previous research [133, 134].

Table 4.2: Metrics profiled for each kernel.

N°	Metrics	Description
1	<code>gpu_cycles_active.avg</code>	no. of cycles where GPU was active
2	<code>gpu_time_active.avg</code>	total duration in nanoseconds
3	<code>gr_cycles_active.avg</code>	no. of cycles where GR was active
4	<code>l1tex__t_bytes_pipe_lsu_mem_global_op_st.sum.per_second</code>	no. of bytes requested for global loads (per sec.) - Efficiency
5	<code>l1tex__t_sectors_pipe_lsu_mem_global_op_ld.sum</code>	no. of sectors requested for global loads
6	<code>lts__t_sector_hit_rate.pct</code>	proportion of L2 sector look-ups that hit (hit rate)
7	<code>lts__t_sectors.avg.pct.of_peak_sustained.elapsed</code>	no. of LTS sectors (L2 utilization)
8	<code>lts__t_sectors_aperture_sysmem_op_read.sum</code>	no. of LTS sectors accessing system memory for reads (system read bytes)
9	<code>lts__t_sectors_aperture_sysmem_op_read.sum.per_second</code>	no. of LTS sectors accessing system memory for reads (system memory read throughput)
10	<code>lts__t_sectors_op_read.sum.per_second</code>	no. of LTS sectors for reads
11	<code>lts__t_sectors_op_read.sum.per_second + lts__t_sectors_op_atom.sum.per_second + lts__t_sectors_op_read.sum.per_second</code>	L2 read throughput
12	<code>lts__t_sectors_op_write.sum.per_second</code>	no. of LTS sectors for writes
13	<code>lts__t_sectors_op_write.sum.per_second + lts__t_sectors_op_atom.sum.per_second + lts__t_sectors_op_read.sum.per_second</code>	L2 write throughput
14	<code>ltc__cycles_elapsed.{avg, sum}</code>	no. of cycles elapsed on LTC {avg, sum}
15	<code>lts__cycles_active.{avg, max, min, sum}</code>	no. of cycles where LTS was active {avg, max, min, sum}
16	<code>lts__cycles_elapsed.{avg, sum}</code>	no. of cycles elapsed on LTS {avg, sum}
17	<code>lts__d_sectors.{avg, max, min, sum}</code>	no. of sectors accessed in data banks {avg, max, min, sum}
18	<code>lts__d_sectors_fill_sysmem.{avg, max, min, sum}</code>	no. of sectors filled from system memory {avg, sum, min, max}
19	<code>lts__t_bytes.{avg, max, min, sum}</code>	no. of bytes requested {avg, sum, min, max}
20	<code>lts__t_request_hit_rate.pct</code>	proportion of L2 requests that hit
21	<code>lts__t_request_hit_rate.ratio</code>	ratio of the proportion of L2 requests that hit
22	<code>lts__t_requests.{avg, max, min, sum}</code>	no. of LTS requests {avg, max, min, sum}
23	<code>smsp_cycles_active.avg.pct.of_peak_sustained.elapsed</code>	SM efficiency
24	<code>smsp_inst_executed_op_global_ld.sum</code>	instruction executed for global loads
25	<code>smsp_inst_executed_pipe_lsu.avg.pct.of_peak_sustained_active</code>	Load Store Unit (LDST) utilization
26	<code>smsp_warp_issue_stalled_dispatch_stall_per_warp_active.pct</code>	proportion of warps per cycle
27	<code>smsp_warp_issue_stalled_dispatch_stall_per_warp_active.pct + smsp_warp_issue_stalled_misc_per_warp_active.pct</code>	stalls for other reasons
28	<code>smsp_warp_issue_stalled_long_scoreboard_per_warp_active.pct</code>	stalls for memory dependency

The profiled metrics are listed in Table 4.2. Some of these metrics have been consolidated into a single metric due to their combination for retrieving known **nvprof** metrics. For instance, metrics 11 and 13 in Table 4.2 correspond to **nvprof**’s *L2_read_throughput* and *L2_write_throughput*, respectively. It’s important to note that selected metrics may exhibit correlations, and at times, a specific *rollup metric* (e.g., sum rather than average) may convey more information than others. Since these details cannot be determined in advance, the analysis conducted in Section 4.6 will shed light on these matters.

4.4.2 Experimental Setup

As mentioned earlier, the *HeSoC-mark* tool is used to introduce interference from the CPU. Within the CPU complex, each core is assigned a unique number from 0 to 7. Core-0 is responsible for submitting work for individual kernels to the GPU, while cores 1 to 7 (numbered accordingly) are utilized to pin processes that generate significant memory pressure through the *meminterf* application in *HeSoC-mark*. This *meminterf* application is capable of generating high-memory traffic by repeatedly executing *memcpy* or *memset* functions with parameterized buffer sizes. For the experiments, the buffer size is set to 50 MB and opted to use *memset*. This setting effectively saturates the CPU-to-DRAM memory access traffic on the shared global memory with the integrated GPU, as highlighted in previous research [107].

Initially, in order to perform and establishing a performance baseline, a tests to collect metrics and execution times for each individual kernel without any interference from the CPU were conducted. Subsequently, the interference is gradually introduced by scaling up to a maximum of 7 tasks generating interference during the execution of individual kernels. Each CPU task represents an instance of the *meminterf* application pinned to a different core. This approach results in a linear increase in the magnitude of CPU-side memory interference. Previous research [13] has shown that CPU-to-DRAM bandwidth for sequential accesses on the Nvidia AGX Xavier is evenly distributed among sibling CPU cores. Sibling CPU cores are pairs of cores that share the same L2 cache, as depicted in Figure 4.1.

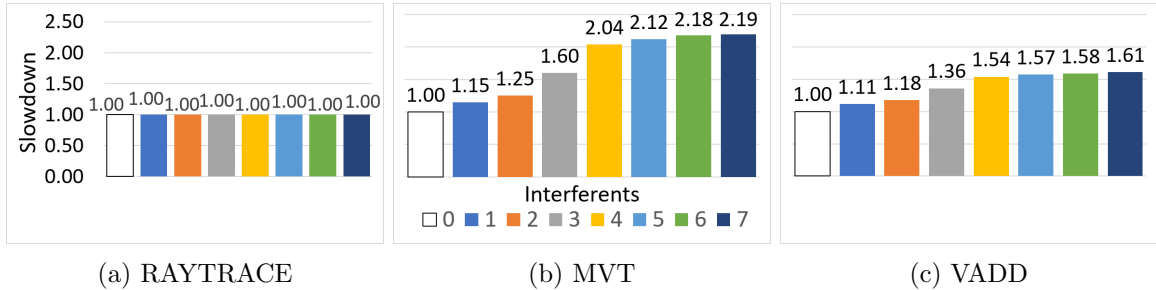


Figure 4.2: Slowdown experienced by different kernels as the number of interferents varies.

The experimental results reveal that the hybrid (MVT) and memory-intensive (VADD) kernels experience slowdowns due to the high level of global memory accesses generated by the CPU. In contrast, the fully compute-intensive kernel (RAYTRACE) remains unaffected. Partial data for three kernels is illustrated in Figure 4.2. Notably, the MVT kernel (Figure 4.2b) experiences the most significant slowdown among the 14 kernels used in the experiments, with a factor of 2.19 when 7 *meminterf* processes are co-running. This confirms that kernels classified as memory-intensive (M) and hybrid (H) experience slowdowns in the presence of high-intensity shared global memory accesses. Furthermore, it underscores that DRAM bandwidth alone is an insufficient predictor of latency deterioration. For instance, MVT experiences a more substantial slowdown compared to VADD, despite requiring nearly 20% less memory bandwidth. Such a behaviour might be explained in the significantly different

way in which these two kernels access memory, both in terms of cache line re-use and memory access patterns.

4.5 Methodologies for Slowdown Prediction

The objective is to devise a method for predicting the slowdown experienced by a single kernel as the number of interfering elements varies. Literature provides several instances of performance predictors that employ machine learning techniques to estimate the expected performance of a specific problem [135, 136]. To address this regression problem, i employ three distinct machine learning methodologies:

- **Support Vector machines for Regression (SVR)** is a machine learning algorithm utilized for predicting continuous numerical values. In contrast to traditional regression models that aim to minimize the error between predicted and actual values, SVR seeks to identify a hyperplane in a higher-dimensional space that best fits the data while maximizing the margin between predicted values and the boundary. This methodology is suitable for handling high-dimensional data with a small number of training samples, and its robustness to outliers renders it an appealing choice for interference prediction.
- **Random Forest Regression (RFR)** is a machine learning algorithm that leverages ensembles of decision trees to address regression problems. The RFR algorithm constructs an ensemble of decision trees, with each tree trained on a random subset of input features and input data. These trees are built and trained using randomized versions of the dataset. Once the trees are constructed, predictions are made by averaging the predictions from each tree. RFR offers interpretability, allowing visualization and analysis of each decision tree within the ensemble. This feature empowers system engineers to gain insights into the relationships between input variables and the output variable.
- **A Deep Neural Network (DNN)** for regression is a machine learning algorithm used to predict continuous numerical values. It processes input data through a series of layers, each comprising interconnected neurons, to generate predictions.

4.5.1 Training and Test

In order to train machine learning methods, a dataset of examples is used, consisting of X that defines the characteristics of an example, and Y that defines the values to be predicted. The dataset comprises all 45 metrics related to the baseline of the entire set of 14 kernels and the number of interferent processes (ranging from 0 to 7) co-running during the execution of each kernel as values X . As the value Y for prediction, is used the corresponding slowdown that each individual kernel experienced during the preliminary analysis (Section 4.4). From the small number of usable experiments (112 cases) available for training, the models were trained by splitting the examples into a training set of 85%, with the remaining 15% serving as the test set. The best hyperparameters for the models were determined using *Optuna* [137]. Hyperparameters refer to all the quantities that remain fixed during the training phase and are set upstream by the user.

During the initial training phase, suboptimal results occurred due to the uneven distribution of Y , with a large number (61) of slowdown cases ≤ 1.1 . To improve training results, the 60% of slowdown values (36) that were below 1.1 were removed, leaving with a total of 76 cases. The distribution of these cases is displayed in Figure 4.3. To evaluate the predictive models with unseen cases after the training phase, a parameterized kernel called PARKER-NEL based on MVT is used, which experienced the highest slowdown during the profiling

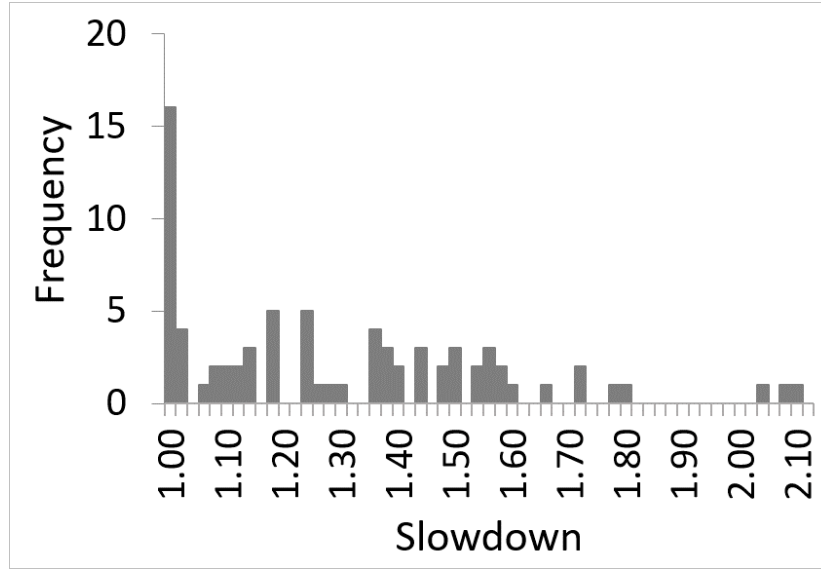


Figure 4.3: Slowdown distribution of the 14 kernels.

phase. PARKERNEL is governed by two parameters (p_0 and p_1). Each couple $\langle p_0, p_1 \rangle$ allowing to consider kernels exhibiting significantly different behaviors, as detailed in Section 4.5.2. Next, different models were employed to predict the slowdown, and these models were assessed using the generated kernels. For each model, an evaluation was performed to determine the minimum, maximum, average, and absolute average of the percentage error, comparing real performance to predicted performance during the prediction process.

4.5.2 PARKERNEL: A Parameterized Kernel

The parameterized kernel serves as the central component of the methodology used for evaluating the predictive models. It can be observed that, for each specific configuration of parameters p_0 and p_1 , PARKERNEL generates kernels that exhibit significantly distinct behavior from those created with a different parameter pair.

Listing 4.1: PARKERNEL implementation

```

1 int i = blockIdx.x * blockDim.x + threadIdx.x;
2 . . .
3 int j = 0;
4 for(j = 0; j < N; j++) {
5     unsigned int f = 0;
6     int m = 0;
7     // compute/memory intensity
8     for (f = 0 ; f < p0 ; f++)
9         m += f + threadIdx.x;
10
11     s[threadIdx.x] = m;
12     // Branch divergence
13     if(i % p1 == 0) {
14         x1[i] += a[i*N+j] * y_1[j];
15         x2[i] += a[j*N+i] * y_2[j];
16     }
17     else{
18         x1[i] -= a[i*N+j] / y_1[j];
19         x2[i] -= a[j*N+i] / y_2[j];
20     }

```

```

21 }
22 x1[i] += s[threadIdx.x];

```

As illustrated in Listing 4.1, parameter p_0 is employed to determine the balance between compute and memory instructions, whereas p_1 governs the degree of branch divergence within a single warp. When both p_0 and p_1 are set to 1, the generated kernel closely resembles the behavior of the MVT kernel. Lower values of p_0 result in kernels that are more memory-intensive, while higher values lean towards being compute-intensive. Similarly, lower values of p_1 lead to minimal branch divergence, while higher values cause kernels to experience more frequent stalls due to branch divergence. This is primarily due to the conditional predicate outlined in Listing 4.1, which depends on the GPU thread ID and has the potential to disrupt the GPU’s lockstep execution at the warp level.

To assess the coverage of the validation kernels in representing the entire output domain, which encompasses all possible slowdown values (Y), two visualizations were created. Firstly, the variation of slowdown with changes in the two parameters was observed while keeping the number of CPU interferences fixed at 7 (Figure 4.4). Additionally, the overall slowdown distribution across all the generated kernels with varying numbers of interferences was examined (Figure 4.5). The parameter ranges were defined as follows: $p_0 \in [1, 10, 20, 30, 40, 50, 75, 100, 125, 150, 175, 200, 250, 300, 350, 400, 500, 1000]$ and $p_1 \in [1, 3, 7, 9, 10, 12, 14, 17, 23, 30]$. Using these parameter steps and ranges in conjunction with the number of interferences, 180 distinct versions of PARKERNEL were generated. This resulted in a total of 1440 individual experiments as the number of interferences varied in each case.

The slowdown distribution for PARKERNEL, as illustrated in Figure 4.5, exhibits a broader coverage of potential slowdown values and contains fewer empty regions along the X-axis when compared to the slowdown distribution of the 14 kernels (Figure 4.3).

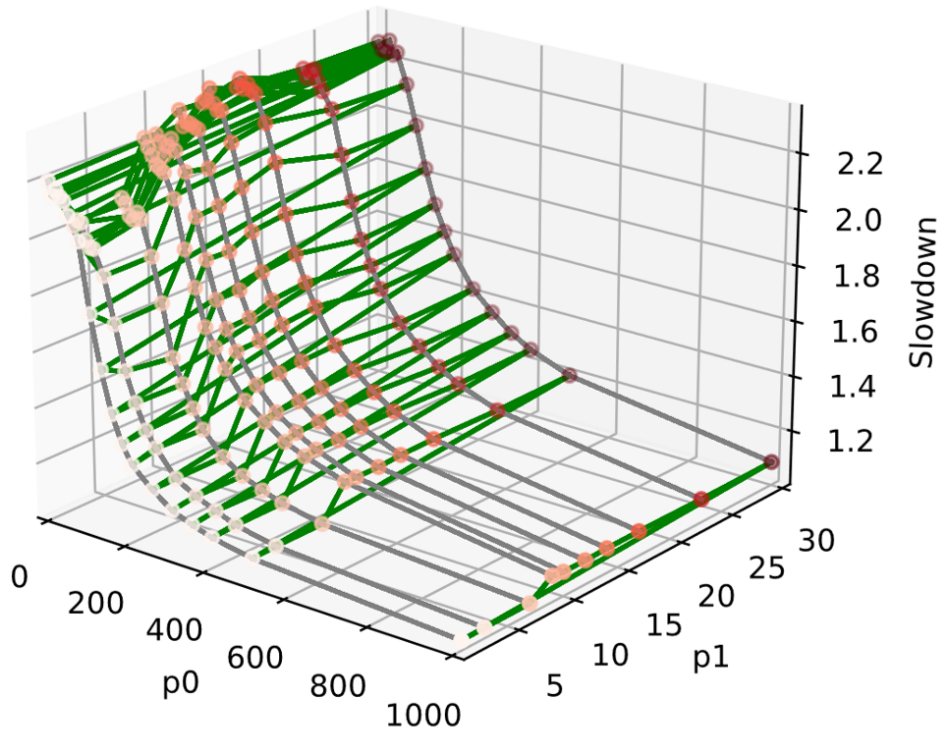


Figure 4.4: Slowdown distribution of PARKERNEL depending on p_0 and p_1 .

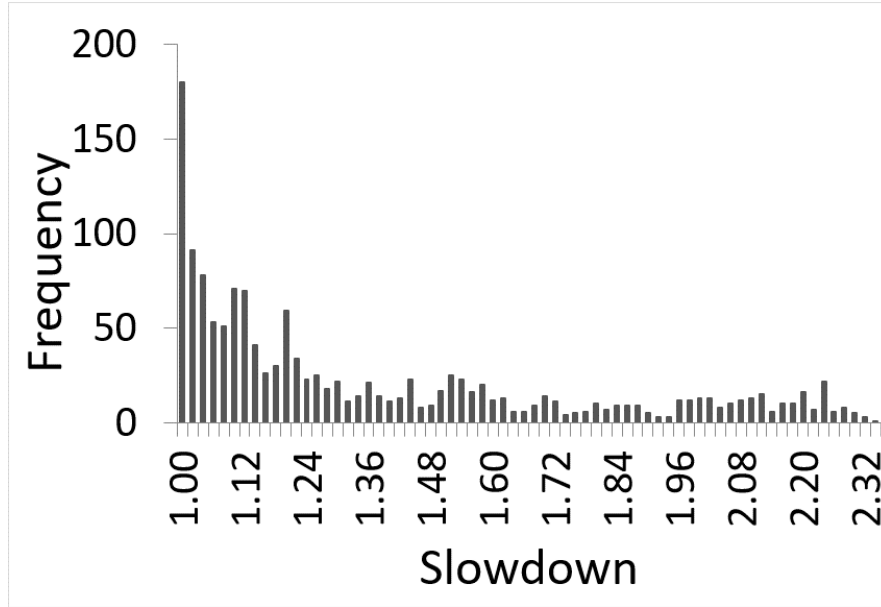


Figure 4.5: Slowdown distribution of PARKERNEL.

4.5.3 Methodologies and Results

In the context of predictive models for real-time systems, it's essential to establish a margin of acceptable error. This is because achieving an exact match between the prediction and the reference value is often impractical. In our presentation of results, it is adopted an arbitrary margin of 20%, a common practice in both industrial and academic applications for Worst-Case Execution Time (WCET) estimation [92, 93, 94]. If the prediction error falls within this 20% threshold, it is considered **in range** (see Fig. 4.7 and 4.6).

Support Vector machines for Regression - SVR

Figure 4.6 reveals the optimal SVR configuration with favorable performance on validation. It exhibits an average percentage error of -4% and an absolute percentage error of 8.33%. The most conservative underestimation stands at -35.15%, while the maximum overestimation reaches 16.29%. The SVR model results in 770 overestimations and 670 underestimations (as depicted in Figure 4.7).

Random Forest Regression - RFR

As depicted in Figure 4.6, the optimal Random Forest configuration displays an average percentage error of -2.83% and an absolute percentage error of 10.16%. The most conservative underestimation is -43.09%, and the maximum overestimation is 22.32%. The Random Forest Regression (RFR) model results in 866 overestimations and 574 underestimations, as illustrated in Figure 4.7.

Deep Neural Network - DeepNN

The optimal configuration, as depicted in Figure 4.8, for the training phase involved a neural network with three hidden layers. The first two hidden layers contained twice the size of the input (46 neurons), while the last hidden layer matched the input size. To predict a real value, a linear neuron served as the final output. Dropout layers were applied at each

layer to enhance reliability and mitigate overfitting. Figure 4.6 demonstrates that the best-performing Deep Neural Network (DeepNN) achieved an average percentage error of -3.30% and an absolute percentage error of 9.29%. The least conservative underestimation recorded was -34.24%, and the highest overestimation reached 13.46%. Regarding the frequency of overestimations and underestimations, the DeepNN generated 777 overestimations and 663 underestimations, as shown in Figure 4.7.

Combined

The combined approach is a strategy that amalgamates the individual predicted slowdowns generated by SVR, RFR, and DeepNN by consistently selecting the most pessimistic prediction among the three methods. As commonly observed in the literature, the combination of different machine learning methods, as long as they are independent, ensures an enhancement in overall performance. In this specific context, this approach also introduces a bias concerning predictions leaning toward either optimistic or pessimistic outcomes. Figure 4.6 reveals that although there is a 0.86% increase in the absolute average error when focusing on the best of the three methods (SVR), the employment of a combined approach results in a 2% reduction in the number of underestimations for the minimum percentage underestimation values. Additionally, the outcomes exhibit a normal distribution centered around 0, resulting in 913 overestimations, 527 underestimations, and 1298 predictions falling within the correctness range of 0-20% (Figure 4.7), signifying that 90% of the predictions lie within the specified correctness range.

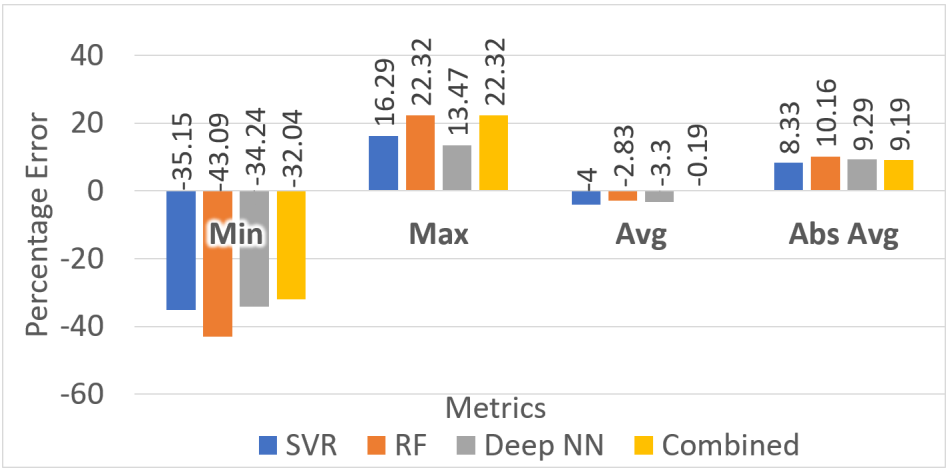


Figure 4.6: Prediction results.

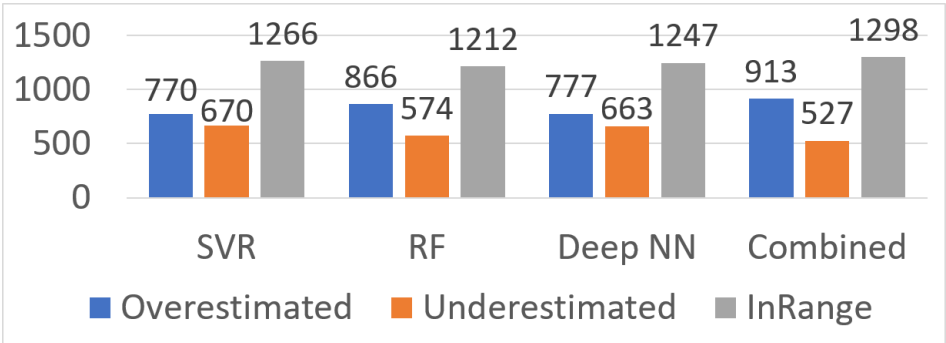


Figure 4.7: Underestimation and overestimation count from the predictive models.

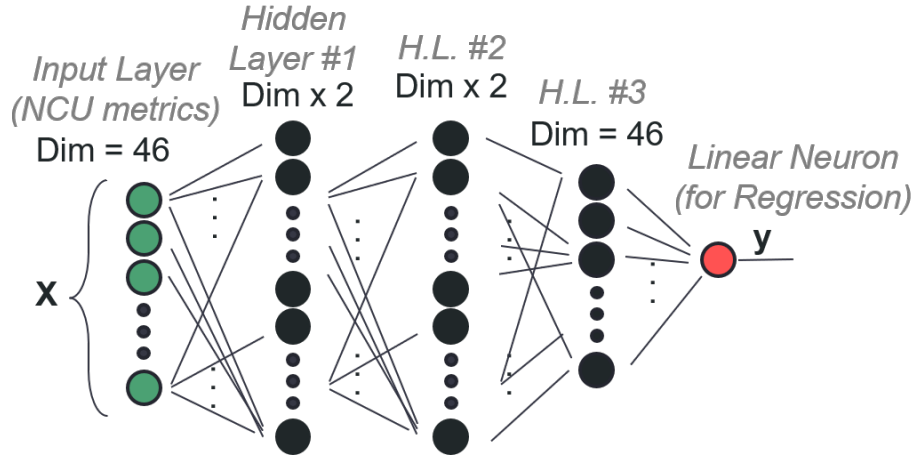


Figure 4.8: The best configuration of the neural network .

4.6 Metrics Analysis

Having established the reasonable performance of the predictive model outlined in the previous section, the next step is to analyze and interpreting the significance of metrics in the context of kernel latency deterioration. To achieve this three distinct methodologies were employed:

- *Principal Component Analysis* (PCA) [138]: PCA is a dimensionality reduction technique designed to transform high-dimensional data into a more manageable set of linearly uncorrelated variables known as principal components. These components are arranged based on their contributions to the total data variance, with the first principal component explaining the most variance. This method is widely used to simplify complex datasets and reveal underlying patterns in the data.
- *GINI importance*: GINI importance quantifies the significance of each feature in a Random Forest model. It relies on the Gini index, a measure of impurity employed in decision tree algorithms for splitting nodes. The GINI importance of a feature is determined by randomly shuffling the values of that feature in the dataset and assessing the resulting decrease in the Gini index. A larger decrease in the Gini index indicates a more important feature. GINI importance is frequently utilized to identify crucial features in a Random Forest model and to perform feature selection for further analysis.
- *Spearman correlation factor*: The Spearman correlation factor, or Spearman's rank correlation coefficient, is a statistical measure that gauges the strength and direction of the relationship between two variables. It is computed by ranking the values of each variable and then calculating the correlation coefficient between their ranks. Spearman is especially useful for assessing the correlation among variables that lack a linear relationship or have non-normal distributions. Its values range between $[-1, +1]$, where -1 denotes a perfect negative correlation, 0 indicates no correlation, and $+1$ signifies a perfect positive correlation. In this section it is used to evaluate the non-linear correlation between each individual metric and the kernel slowdown resulting from memory interference. In this analysis, metrics with a Spearman coefficient greater than or equal to 0.21 or less than or equal to -0.21 are considered most important. The threshold of 0.21 is derived from the absolute average value calculated across the entire correlation dataset.

Table 4.3: Metrics that are more influential during slowdown prediction.

N° in Tab. 4.2	Metric Name	PCA	GINI	Spearman
-	Interferents_nr (number of interferents by CPU side)	2.682	0.469	0.529
28	smsp__warp_issue_stalled_long_scoreboard_per_warp_active.pct	19.688	0.135	0.504
27	smsp__warp_issue_stalled_dispatch_stall_per_warp_active.pct + smsp__warp_issue_stalled_misc_per_warp_active.pct	11.630	0.095	-0.378
26	smsp__warp_issue_stalled_dispatch_stall_per_warp_active.pct	3.859	0.040	-0.378
25	smsp__inst_executed_pipe_lsu.avg.pct_of_peak_sustained_active	1.048	0.025	-0.343
7	lts__t_sectors.avg.pct_of_peak_sustained_elapsed	-	-	0.371
9	lts__t_sectors_aperture_sysmem_op_read.sum.per_second	-	-	0.304
10	lts__t_sectors_op_read.sum.per_second	-	-	0.301
4	lltex__t_bytes_pipe_lsu_mem_global_op_st.sum.per_second	-	-	0.286
12	lts__t_sectors_op_write.sum.per_second	-	-	0.285
24	smsp__inst_executed_op_global_ld.sum	2.168	0.022	-
22	lts__t_requests.sum	-	0.018	-
22	lts__t_requests.min	-	0.017	-
22	lts__t_requests.max	-	0.016	-
6	lts__t_sector_hit_rate.pct	0.737	-	-
19	lts__t_bytes.min	50.826	-	-
21	lts__t_request_hit_rate.ratio	5.439	-	-
23	smsp__cycles_active.avg.pct_of_peak_sustained_elapsed	1.458	0.022	-

For each of these methodologies, the focus was on metrics exhibiting the highest correlations. This involved creating a set for each method, with each set containing the top 10 values according to GINI, PCA, and Spearman analyses. Subsequently, an intersection set was obtained by identifying metrics common to all three. Within this intersection set, five metrics consistently appeared across all three sets. These metrics pertained to the number of CPU-side interferents, memory dependency stalls, the proportion of warps per cycles, and Load Store Unit utilization. Further, a union set was formed by combining the individual top 10 metrics from each analysis methodology. This process revealed 18 metrics deemed the most influential using different analytical methods. The final 18 metrics, along with the values derived from the methodologies used to assess metric importance, are presented in Table 4.3. To validate the metrics importance assessment, the three predictive methodologies were retrained and combined using a pessimistic approach that exclusively considered these 18 metrics. The validation was conducted in the same manner as previously, utilizing PARKERNEL-generated kernels. In this setting, the combined method yielded a mean and absolute error of 2.79% and 10.24%, respectively. It estimated the minimum and maximum slowdown to be -32.21% and 21.95%, with 995 overestimations and 445 underestimations. Thus, the prediction quality remained nearly consistent with what was reported in Section 4.5. This outcome validates and emphasizes that these 18 metrics are paramount for characterizing the behavior of GPU kernels and CPU-side memory interference.

4.7 Conclusion

In this chapter was presented an in-depth analysis of GPU kernel behavior under varying memory-intensive CPU-side interference scenarios. To predict GPU kernel slowdown resulting from intensive CPU-side memory activity, various machine learning (ML) methodologies, including SVR, RFR, DeepNN, and a combined approach, were employed. The ML methodologies underwent training with fine-grained profiling data, made accessible through Nvidia’s profiling tool, `ncu`. The combined approach yielded favorable outcomes, with a mean and absolute error percentage of -0.19% and 9.19%, respectively. Additionally, well-established dataset analysis techniques such as PCA, GINI, and Spearman were employed to identify 18

out of the initial 45 `ncu` metrics as the most influential predictors. This finding was further substantiated by retraining the predictive model using only the 18 selected metrics. Consequently, the predictive model trained with this restricted feature set demonstrated an average and absolute percentage error of +2.79% and 10.24%, respectively.

A combined predictive model of this nature can prove invaluable for assessing system behavior under varying computational workloads between the CPU and GPU. Consequently, it has the potential to become an essential tool for response time analysis and memory interference-aware scheduling. Regarding future prospects, the pivotal metrics identified through the proposed approach offer an opportunity to enhance existing state-of-the-art heterogeneous task models within real-time literature. Specifically to the DAG-based task models, which represent jobs as interconnected nodes within a directed acyclic graph. Each node is associated with various timing parameters such as period, WCET, deadline, offset, and more. In heterogeneous DAGs, nodes are designated to run on specific accelerators. Herein lies the significance of these findings: the identified key metrics for predicting memory interference can be incorporated to enrich the associated parameters for each node, empowering system engineers to consider memory interference when deploying task sets on real hardware.

Chapter 5

Dissecting Memory Explicit Copy vs Unified Virtual Memory

5.1 Introduction

Today, numerous CUDA applications are developed to address highly parallelizable problems. Unfortunately, the development of CUDA applications requires an in-depth and advanced understanding of both the development architecture and the related programming APIs, posing a significant challenge for less experienced programmers.

Nevertheless, there are two significant challenges within the prevalent GPU programming paradigm that substantially hinder extended utilization. Firstly, the distinct separation of physical memory between a GPU and a CPU mandates explicit memory oversight in the traditional GPU programming model. This necessitates programmers to conscientiously manage the copying of data between CPU and GPU memories to ensure it resides in the correct location for usage (i.e., copy-then-execute). Therefore, there is a need to perform a data transfer between the host (CPU) and the device executing the code (GPU), commonly referred to as Host to Device (HtoD). Conversely, when the algorithm has completed execution on the GPU and the final results need to be passed to the host for further processing, it is commonly known as Device to Host (DtoH) transfer. Secondly, the conventional GPU programming model lacks the capability to execute a kernel if it requires more memory than what is available in the GPU's memory space, thereby precluding memory over-subscription.

To address these issues, Nvidia provides a solution that simplifies a significant part of the development and the afore mentioned issues, known as Unified Virtual Memory (UVM).

In this chapter, I will therefore delve into two types of memory management: UVM and the explicit copies, with the aim to guide the developer in balancing the convenience of UVM against potential performance increments or, more often, decrements. This consideration extends beyond the ease of program writing, as many auto-parallelization frameworks on GPUs or external libraries leverage UVM. This is because translating CPU code into kernel code is more easily automated with UVM. Moreover, similar to what has been done in the previous Chapter 4, the aim is to lay the first groundwork for predictive models based on hardware metrics to aid the developer to understand the complex balance among memory consumption, coding style complexity and performance of arbitrary CUDA kernels.

5.2 Memory Management in CUDA

It is necessary to highlight two major distinctions between different types of GPUs: integrated and discrete. Discrete GPUs (such as GTX, RTX, and earlier models) communicate with the

rest of the system through a PCI-ex bus, while on embedded chips (integrated GPUs), there is no PCI-ex bus for this type of connection. Integrated GPUs are directly connected to the same shared main memory with the CPU, as shown in Figure 4.1 for the architecture of Nvidia Jetson Xavier. This type of connection enables the use of the same memory banks.

Understanding this memory separation and the two worlds of embedded and discrete devices provide the necessary context to thoroughly evaluate the two data transfer strategies. Explicit copies with *cudaMemcpy* requires explicit memory allocation on both the host (CPU) and the device (GPU) side, and populating, through copies, from host to device (HtoD) and vice versa (DtoH) at the end of kernel execution.

Explicit memory transfer offers a finer control, allowing the developer to set precise specifications of the data to be transferred, more specifically he or she can control when and how data is copied from HtoD or DtoH. Explicit copies are managed by the GPU Copy Engine (CE), which executes a DMA (Direct Memory Access) transfer with respect to CPU L2, but effectively operating a GPU LLC pre-fetching [13]. However, it is essential to consider the transfer cost associated with this methodology, especially when handling large amounts of data or frequent transfers, as the transfer overhead must be taken into account to pass data from HtoD and DtoH using a PCI-ex bus. Performance considerations aside and on the point of view of the developer, explicit or manual memory management in such heterogeneous systems adds complexity during the software development phase and requires detailed attention to avoid potential errors.

The UVM-based approach implements a unified memory view through a single address space accessible by every processor and device. This enables transparent access from both the CPU and GPU, eliminating the need for explicit data transfers, hence significantly simplifying the programming model. More specifically, through UVM the application developer has to explicitly omit copying data *to* and *from* the GPU device. On a more technical perspective, UVM tracks a pool of managed memory that is shared between the host and the device. Implementation-wise, such managed memory is accessible to both the CPU and GPU by addressing the same pointer. The GPU driver automatically migrates data (i.e. memory pages) allocated in Unified Memory from/to host and device in a completely transparent manner on the point of view of the programmer. An example of CUDA code required for launching a kernel using the classic memory management method is visible in Listing 5.1, while in Listing 5.2 the version of the same code using UVM memory management is visible.

Listing 5.1: Example of classic cuda memory management

```

1 struct dataElem {
2     int prop1;
3     int prop2;
4     char *name;
5 }
6
7 void launch(dataElem *elem) {
8     dataElem *d_elem;
9     char *d_name;
10    int namelen = strlen(elem->name) + 1;
11
12    // Allocate storage for struct and name
13    cudaMalloc(&d_elem, sizeof(dataElem));
14    cudaMalloc(&d_name, namelen);
15
16    // Copy up each piece separately, including new "name" pointer value
17    cudaMemcpy(d_elem, elem, sizeof(dataElem), cudaMemcpyHostToDevice);
18    cudaMemcpy(d_name, elem->name, namelen, cudaMemcpyHostToDevice);
19    cudaMemcpy(&(d_elem->name), &d_name, sizeof(char*), cudaMemcpyHostToDevice);
20 }

```

```

20
21 // kernel launch, CPU and GPU use different copies of "elem"
22 Kernel<<< . . . >>>(d_elem);
23
24 // use elaborated data ...
25 }
26 ...

```

Listing 5.2: Example of UVM memory management

```

1 struct dataElem {
2     int prop1;
3     int prop2;
4     char *name;
5 }
6 void launch(dataElem *elem) {
7     dataElem *d_elem;
8     char *d_name;
9     int namelen = strlen(elem->name) + 1;
10
11 // Allocate storage for struct and name
12 cudaMallocManaged(&d_elem, sizeof(dataElem));
13 cudaMallocManaged(&d_name, namelen);
14
15 // kernel launch
16 Kernel<<< ... >>>(d_elem);
17 cudaDeviceSynchronize();
18
19 // use elaborated data ...
20 }
21 ...

```

When a kernel accessing memory managed data is launched, warps accessing such a resource will trigger a page-fault to be handled by the GPU Memory Management Unit (MMU), which in turn will cause the appropriate address translation and page migration on the GPU LLC.

The implications of developing GPU-accelerated applications using UVM are twofold:

(1) Unified Memory significantly simplifies parallel programming on these kind of heterogeneous platforms by transforming device memory management from a mandatory task into an optimization step. This innovation allows programmers to focus primarily on the development of parallel GPU kernels, without the initial complexity of handling device memory allocation and transfers. Consequently, this advancement not only streamlines the learning process for beginners and/or non-expert developers but also eases the transition of existing codebases to GPU utilization; as a consequence of this, UVM simplifies the implementation of source-to-source translators and automatic parallelization frameworks [139, 140].

(2) From a performance standpoint, the dichotomy between Unified Virtual Memory (UVM) and explicit copies, especially when considering both integrated and discrete GPUs, opens up numerous and intriguing research questions. In an integrated/embedded environment, where there is a single system memory space, using UVM for the same application shows a clear saving in terms of memory occupancy, since it's not necessary to have two buffers (host and device) for each buffer used by the kernels. Moreover, serving page faults "on the fly" is definitely faster on embedded systems than on discrete ones: this becomes obvious when looking at the total system RAM bandwidth of a modern Tegra platform compared to PCI-Express 3.0 or even PCI-Express 4.0 standards. On the other hand, the explicit copy system also acts as a prefetcher for the GPU Last Level Cache (LLC), allowing for a

better hit-ratio at kernel launch. Therefore, it becomes interesting and non-trivial to study the characteristics of each kernel with the granularity allowed by the hardware counters, in order to predict which of these effects will prevail over the others in these two memory models.

In other words, two different behaviors emerge based on the type of architectures. When using discrete devices, if the required data is not locally present in the GPU’s main memory, the page faulting mechanism generates a memory request to the CPU’s main memory. Subsequently, the memory manager initiates a request for the data from the host by accessing the PCI-ex. On the other hand, in embedded architectures, the memory manager retrieves the data without the overhead associated with accessing through the PCI-ex. This process is referred to by Nvidia as *zero-copy*.

As a result of this data movement behavior between memories, frequent access to data not allocated on a specific processor increases the overhead for data exchange execution and, consequently, waiting stalls due to movements. Additionally, through the use of UVM, the memory manager makes heuristic choices to avoid excessive page faults and maintain data locality, unlike explicit copy management, where chunk of data are already present in GPU VRAM depending on the developer’s choice.

However, it is crucial to examine UVM performance in specific scenarios, as memory access can result in cases of overhead or performance improvement. The overhead of UVM page migrations may or may not be hidden during computations, hence leading to potentially unfavorable cases depending on the programs to be executed. Throughout this chapter, executions using traditional memory management, i.e., through explicit copies, are indicated as *COPY*, while those related to unified memory management are labeled as *UVM*.

5.3 Analysis

The comparative analysis of these two memory management mechanism will be guided through benchmarking, profiling all available kernel metrics on each real architecture under consideration, using the same methodologies as presented in Chapter 4. This involves the use of tools such as Nsight Compute (NCU) and Nsight System (NSYS) on the same set of kernels presented in Section 3.4, with launch configurations detailed in Table 4.1. Utilizing analysis techniques like PCA and Spearman correlation I will provide additional insights for memory behaviour of explicit and implicit data copies.

To discern differences between the two methods, the execution timings of the entire pipeline are also analyzed. The profiling timings begin with the initialization (cuda malloc) of data operations for GPU kernel execution and continue until the processed data is returned to the CPU and accessed for UVM, ensuring the same end copying behavior as the COPY method. For each experiment, a speed-up is calculated for each individual kernel execution (k) with both methods (COPY and UVM). This speed-up is computed as the ratio between the two execution timings, as indicated in Equation 5.1.

$$Speed_Up(k) = \frac{T_{COPY}(k)}{T_{UVM}(k)} \quad (5.1)$$

The expression *entire execution pipeline* refers to the fact that for COPY, we measure the latency from the first HtoD cudaMemcpy to the latest DtoH copies after the kernel execution. In order to provide a fair benchmarking mechanism, this translates in UVM in profiling not only the kernel execution that will access the same amount of memory accessed as in the COPY, but we will also execute a pointer walk through the same output buffer after the kernel execution. This is done because it is assumed that kernel output data must be readily available on CPU side.

Another important aspect to highlight is how explicit copies have been set up. In this case, when allocating host and device memory: the developer selects between two options: (1) pageable memory and (2) pinned memory. (1) represents the default option, that is implemented by simply taking a regular CPU-side memory buffer (e.g., allocated with a *malloc*) and allocating the respective destination buffer on the device (using *cudaMalloc*). Since the GPU is not able to access such data from pageable host memory, for the subsequent *cudaMemcpy* to device memory the CUDA driver must first allocate a temporary page-locked (CPU-side) buffer to then use it as a temporary storage for the actual copy. This might significantly hinder the performance of the GPU DMA copy engine. Therefore, for these sets of benchmarks and related analysis (and similarly to the experiments set up in the previous chapters), pinned memory (2) has been used to handle allocations in the COPY method. From the point of view of the programmer, pinned allocations are implemented by allocating CPU-side memory with the *cudaMallocHost* function. This effectively maximizes the CPU-GPU memory bandwidth for data transfers by fully exploiting DMA with no need for staging or temporary host buffers. This is important for the fairness and significance of the presented benchmarks.

Four different real architectures were used for benchmarking COPY vs UVM: two embedded Nvidia boards, namely Orin and AGX Xavier, and two discrete graphics cards, RTX 2080Ti and RTX 2070, both from laptops. Each individual kernel has been timed and profiled 150 times, with the initial 50 runs designated as warm-up and subsequently discarded. This approach ensures stabilized values and helps avoid polluted caches that could lead to potentially unwanted hits or performance boosts. The relative *Speed-Up* for each kernel on each analyzed architecture is presented in Table 5.1.

Table 5.1: Kernels’ speed-up on different real architectures.

Kernel Name	Speed-Up			
	RTX 2070	RTX 2080Ti	Xavier	Orin
VADD	0.294	0.267	0.464	0.507
SAXPY	0.291	0.270	0.345	0.508
COPY	0.218	0.204	0.231	0.357
RAYTRACE	0.629	0.547	0.696	0.848
DXTC	0.788	0.922	0.893	0.853
CONV	0.235	0.200	0.502	0.419
MVT	0.881	0.850	1.289	1.269
PF	2.680	1.858	2.493	2.948
DOITGEN	0.519	0.540	0.391	0.314
HOTSPOT	0.294	0.249	0.553	0.402
NBODY	0.991	0.987	0.998	0.982
HISTO	0.461	0.404	1.239	1.106
ATAX	1.602	1.672	1.100	1.192
BFS	0.923	0.988	0.997	1.036

As indicated by the *Speed-Up* values in Table 5.1, it is highlighted how using UVM degrades the performance of tasks that are synthetic and memory-bound, such as VADD, SAXPY, and COPY, which involve minimal computation, especially due to cold caches and non-existent cache line reuse. For this type of task, UVM is forced to copy the data page by page, impacting cache locality performance. Meanwhile, with explicit copies, using the GPU’s CopyEngine, which always passes through the L2 cache, data locality for warp execution is improved,

resulting in fewer stalls.

This slowdown is mitigated for tasks that feature a degree of compute instructions able to mask the page-faulting mechanism latencies. Trivially, page migration has a significantly different cost depending on using a discrete or integrated device. More specifically, such a cost is lower on embedded architectures due to shared memory between the CPU and GPU and the absence of access and communication via a PCI-ex bus.

As indicated by the performance improvements highlighted in Table 5.1, the PF and ATAX kernels show a significant increase in performance when UVM is employed on all platforms, both embedded and discrete. In this case such a completion time improvement is attributed to the behavior of the memory manager with UVM, which, during the page-fault phase, manages to mitigate latencies and data access during kernel execution through advantageous memory access patterns for the task type. In other words, cache locality and cache line re-use prevails over the page migration costs on both discrete and embedded.

For the MVT and HISTO kernels, performance improvements are observed only on embedded architectures (Orin and Xavier). In this case, we can see how the reduced page migration time on embedded architecture favours the UVM methodology over explicit copies.

For the NBODY and BFS kernels, on the other hand, there is a slight degradation of performance that could potentially be negligible, standing around 2% and 8%, respectively in all the tested platforms. This suggests that in both embedded and discrete, for these kernels, the price to pay for having the advantages of managed memory is reasonably small.

The benchmark presented in this chapter just adds to the quite extensive literature on managed memory in GPU-accelerated platforms for specific applications [141, 142, 143, 55, 144]. However, it is still unclear how to understand where lies the balance between prefetched memory versus page migration costs, or even better, estimating how much performance a developer might be willing to sacrifice if the simplified UVM model is adopted. The rest of this chapter will present a novel, HW-metric based and data driven analysis to answer these research questions.

5.4 PCA and Spearman Coefficient

To obtain more valuable insights into the *Speed-Up* calculated using the UVM method, I applied the methodologies of PCA and Spearman coefficient as presented in Chapter 4, specifically in section 4.6.

In Chapter 4, which focuses on memory interference between CPU and GPU, a reasonable estimation of parameters influencing a potential predictive model was conducted, confirming that by using different analysis techniques, it is possible to retrieve valuable metrics that manage to summarize the same information as a larger database. In this section, the metrics that will be most influential for creating predictive models will be highlighted. However, from the first benchmark executed, as shown in Table 5.1 and the analyses conducted in other research [141, 142, 143, 55, 144], a more complex trend emerges compared to the simple concept of "memory/compute intensive" kernels. A manual a priori selection of metrics to profile is therefore challenging. Consequently, all available metrics for the individual architectures under examination were profiled. Out of approximately 6270 metrics, 95 were profiled using the NSYS tool. These include information about compilation architecture, GPU name, Nvidia driver, CUDA driver, launch parameters of individual kernel grids (X, Y, Z), and data transfer times DtH and HtD, among others. The remaining metrics pertain to information about kernel execution on the execution architecture (as in Chapter 4). The number of profileable metrics across different architectures using the `ncu` tool may vary, as some metrics might not be profileable on a particular architecture.

To conduct this analysis, a script was created to test each profileable metric using the `ncu` tool. If a metric is profileable, it is saved in a dictionary for subsequent profiling of each individual kernel. Subsequently, a second script performs profiling using both `ncu` and NSYS tools, compiling the host code with a flag to select one of the two implemented methodologies (UVM or COPY).

Each file containing kernel metrics is then analyzed to extract additional statistics such as minimum, maximum, average, and standard deviation of each metric over 100 executions of the individual kernel. These statistics are subsequently highlighted in the names of the metrics in the subsequent tables, using labels $\{(Min), (Max), (Avg), (Std)\}$.

Finally, files generated by NSYS and `ncu` for each kernel execution are combined to create a unified dataset that will be analyzed using PCA and correlation coefficient for each metric with respect to the *Speed-Up*.

PCA

The Principal Component Analysis (PCA) aims to transform a high-dimensional dataset into a more manageable set of linearly uncorrelated variables. By reducing the dataset to a smaller set of relevant metrics, it can be used to feed and train various machine learning methods. Through PCA, 40 metrics have been identified with 85% of the variance explained. As highlighted in Table 5.2 and in Table 5.3 along with the respective descriptions of the meaning of each metric, these 40 metrics synthesize the initial database. As we delve into more detailed analysis (Table 5.3), among the initial metrics, there are metrics regarding the Load Store Unit pipeline, memory clock frequency, hit ratio for write operations in the cache, grid launch size, the number of instructions executed by the divergence unit (ADU), and the average memory barrier operations, among others. Many of these metrics, obtained from PCA, are related to findings found in other research [143], which is, to the best of my knowledge, the largest and most complete collection of UVM-related measures. This latter contribution, however, does not explicitly consider HW metrics as presented in this section.

Spearman Correlation

The Spearman correlation factor, or Spearman’s rank correlation coefficient, is a statistical measure that gauges the strength and direction of the relationship between two variables and assessing the correlation among variables that lack a linear relationship or have non-normal distributions. In this section Spearman method is used to evaluate the non-linear correlation between each individual metric and the kernel *Speed-Up*.

In this analysis, due the high number of metrics in the dataset to analyze, I only took the overall average values “(Avg)” of metrics derived from the Spearman correlation. The top 100 results of the Spearman correlation are presented in Table 5.5 (continued into Table 5.6).

Concerning the top 10 metrics identified through correlation with the *Speed-Up*:

- `smsp__average_inst_executed_pipe_lsu_per_warp.ratio`: represents the average ratio of instructions executed per clock cycle by the load/store unit (LSU) per warp.
- `sm{smsp}__sass_inst_executed_op_st.sum`: represent the average number of instructions executed per memory operation and store operations, respectively, per SM and per warp.
- `smsp__average_inst_executed_pipe_lsu_per_warp.pct`: represents the average percentage of instructions executed by the load/store unit (LSU) compared to the total instructions executed per warp.

- `smsp{sm}_sass_inst_executed_op_st.max`: represent the maximum number of instructions executed per store operation per warp and per SM.
- `smsp_warp_issue_stalled_drain_per_warp_active.pct`: represents the average percentage of stall cycles due to issue or drain stalls per active warp per SM.
- `l1tex_data_pipe_lsu_wavefronts_cmd_read{.sum,.avg}`: These metrics concern the total and average number of read command requests by the L1 load/store data pipe per wavefront.

In this ranking of the top 100 Spearman metrics, some of them correspond to the results obtained through Principal Component Analysis (PCA), highlighting that the same four metrics were previously identified through PCA. The metrics common to both methods are: `smsp_average_warp_latency_ratio(Avg)`, `sm_mio_pq_read_cycles_active_pipe_lsu.max(Avg)`, `DtoHSizeMed(MB)`, `sm_mio_pq_read_cycles_active_pipe_lsu.sum(Avg)`.

Another metric that can be easily controlled or intuitively understood is the kernel launch configuration. This metric, along with others concerning launch parameters, is included among the metrics highlighted by Spearman correlation. For example, the `DtoHSizeMed(MB)` is negatively correlated with the *Speed-Up*, indicating that the average size of data transmitted from the device to the host plays a crucial role in the *Speed-Up* of tasks. If the `DtoHSizeMed(MB)` decreases, the *Speed-Up* of the task increases, based on the negative correlation value of -0.657.

Taking a closer look at Table 5.5, many of the metrics displayed therein are clearly correlated with each other. For instance, `DtoHTimeMax,Med` (16, 17) essentially represent the same concept, as do the metrics `DtoHSizeAvg,Med,Min,Max` (from 65 to 68), which also share nearly identical correlation values. This sheds light on the importance of performing an analysis using PCA to alleviate the workload on any potential predictive model.

By examining the results of both PCA and the Spearman Table 5.5, and observing metrics with high rankings, it becomes apparent that some of these metrics are easily understandable and extractable, while others can be readily controlled by the programmer during development. However, there are also other types of metrics that are considerably more complex to extract and manage. For example, metrics like `sm_inst_executed_pipe_adu.sum`, which refer to instructions executed by the ADU (Address Divergence Unit), as well as those ranking high in Spearman, such as those concerning the quantity and dispatching of instructions per warp, and subsequently, those regarding how many of these instructions halt due to data dependencies or lack thereof (`{sm}smsp_sass_inst_executed_op_st.max`, `sm_sass_inst_executed_op_st.max`, `smsp_warp_issue_stalled_drain_per_warp_active.pct`, etc.), are non-trivial to extract. In addition, metrics starting with "l1txt_" and "lts_", which refer to L1 and L2 caches respectively, provide valuable information for machine learning purposes. However, extracting them manually by a programmer without a data-driven approach and profiling tests becomes extremely complex.

5.4.1 Conclusion

In this Chapter, which presents the initial insights of a preliminary work, the foundations have been laid for a deeper analysis of the behavior of two different memory management methods in CUDA applications, namely UVM and COPY. Through an initial analysis of various types of acceleration architectures, including discrete and embedded GPUs, timings and metrics related to both methodologies have been profiled across different types of tasks.

This has revealed that there are cases, and therefore specific tasks, that may benefit from using the UVM memory management method. Subsequently, the profiled metrics have been examined through PCA and Spearman to gain further insights into this dataset and the relative speed-up of tasks. These metrics could be employed for future in-depth analyses using other methodologies, such as neural networks and other machine learning methods. As a future research perspective, suitable predictive models will be sought to forecast slowdown or speed-up in UVM usage scenarios that are tailored for specific architectures.

Table 5.2: PCA analysis on metrics.

N°	Metrics Name	PCA Value
1	sm_mio_pq_read_cycles_active_pipe_lsu.sum(Min)	21.97
2	Mem_Clock(MHz)	12.138
3	lts_t_sector_op_write_hit_rate.ratio(Min)	9.340
4	gridY	6.455
5	l1tex_t_sector_pipe_lsu_mem_global_op_st_hit_rate.ratio(Max)	5.232
6	sm_inst_executed_pipe_adu.sum(Max)	4.699
7	lts_t_requests_srcnode_gpc_aperture_sysmem_op_membar.avg(Min)	4.407
8	DtoHSizeMed(MB)	3.510
9	fe_output_ops_type_bundle.cmd_go_idle.avg(Min)	2.761
10	smsp_average_warps_issue_stalled_selected_per_issue_active.ratio(Avg)	2.025
11	sm_threads_launched.sum(Min)	1.991
12	smsp_sass_average_data_bytes_per_sector_mem_global_op_ld.ratio(Max)	1.903
13	smsp_average_warps_issue_stalled_selected_per_issue_active.pct(Max)	1.495
14	smsp_average_warps_issue_stalled_selected_per_issue_active.pct(Min)	1.291
15	smsp_sass_average_data_bytes_per_wavefront_mem_shared.pct(Min)	1.168
16	l1tex_t_sector_pipe_lsu_mem_global_op_ld_hit_rate.ratio(Max)	0.875
17	HtoDSizeStdDev(MB)	0.716
18	smsp_warps_launched.avg(Min)	0.499
19	smsp_average_warps_issue_stalled_short_scoreboard_per_issue_active.ratio(Std)	0.441
20	GPU_Max_Clock(MHz)	0.409
21	smsp_average_warp_latency.ratio(Std)	0.347
22	DtoHCount	0.282
23	smsp_average_warps_issue_stalled_selected_per_issue_active.pct(Std)	0.248
24	lts_t_requests_srcunit_tex_aperture_device_op_read_lookup_miss.avg(Min)	0.162
25	smsp_average_warps_issue_stalled_selected_per_issue_active.pct(Avg)	0.122
26	smsp_average_warps_issue_stalled_selected_per_issue_active.ratio(Min)	0.109
27	lts_t_requests_aperture_device_op_membar.avg(Min)	0.084
28	l1tex_t_output_wavefronts_pipe_tex.min(Std)	0.063
29	smsp_average_warps_issue_stalled_branch_resolving_per_issue_active.ratio(Std)	0.049
30	lts_t_requests_srcnode_gpc_aperture_sysmem_op_membar.avg(Max)	0.045
31	lts_t_sectors_aperture_sysmem_op_membar_lookup_hit.sum(Min)	0.040
32	smsp_inst_executed_op_global_ld_pred_off_all.avg(Min)	0.036
33	lts_t_requests_aperture_device_op_write.avg(Min)	0.031
34	lts_t_sectors_srcunit_tex_aperture_device_op_read.avg(Min)	0.027
35	lts_t_requests_aperture_device_op_read.avg(Min)	0.020
36	lts_t_requests_aperture_device_lookup_hit.avg(Min)	0.015
37	smsp_warps_issue_stalled_drain.avg(Min)	0.007
38	lts_t_sectors_aperture_device.avg(Min)	0.007
39	smsp_average_warps_issue_stalled_wait_per_issue_active.ratio(Std)	0.004
40	PipeTimeAvg(ms)	0.000

Table 5.3: Metrics meaning.

N° in Tab. 5.2	Description
1	Sum of active cycles for reading from the priority queue in the LSU (Load Store Unit) pipeline (Min)
2	Memory clock frequency in MHz
3	Hit rate for write operations in the temporal sector cache (Min)
4	Launch kernel grid configuration (Y)
5	L1 cache hit rate for store operations in the global memory sector cache
6	Instruction executed by the ADU (Address Divergence Unit)
7	L2 Cache tag stage average requests for memory barrier operations in the system memory through the General Processing Cluster (GPC) that contains SMs
8	Average size of data transfers from device to host
9	The FE (Front End) unit (responsible for the overall flow of workloads sent by the driver) for operation to go idle
10	Ratio of selected stalled warps per active issue (SMSP: each SM is partitioned into four processing blocks)
11	Sum of launched threads
12	Ratio of average data size per sector in global memory load operations
13	Average percentage of selected stalled warps per active issue
14	Average percentage of selected stalled warps per active issue
15	Percentage of average data size per wavefront in shared memory operations
16	Hit rate for load operations in the global memory sector cache
17	Standard deviation of sizes of data transfers from host to device (HtD)
18	Average of launched warp of the partitioned four processing blocks in SM
19	The Average ratio for stalled warps per active issue
20	Maximum GPU clock frequency (MHZ)

Table 5.4: Metrics meaning (continue of Table 5.3).

N° in Tab. 5.2	Description
21	Ratio of average warp latency
22	Count of data transfers from device to host
23	Average percentage of selected stalled warps per active issue
24	Average request of "miss" operation in reading L2 cache
25	Average percentage of selected stalled warps per active issue
26	Average percentage of selected stalled warps per active issue
27	Average requests for memory barrier operations through L2 Cache
28	L1 minimum output generated by the wavefronts (all work items are processed in parallel)
29	Ratio of branch-resolving stalled warps per active issue
30	L2 cache average requests for memory barrier operations in the system memory through the GPC source node
31	The sum of sectors for memory barrier operations in the system memory with cache hits
32	Represents the average number of instructions executed for global load operations with prediction disabled across all SMs
33	The average number of requests for write operations in the L2 cache
34	Represents the average number of sectors for read operations in the L2 cache with the source unit texture
35	The average number of requests for read operations in the L2 cache through the device
36	The average number of requests for lookup operations in the L2 cache with cache hits through the device aperture
37	The average number of warps that are stalled due to draining on the Streaming Multiprocessors
38	The average number of sectors for operations in the L2 cache through the device aperture
39	The ratio of the average number of warps that are stalled in wait per active issue on the SMSP
40	The average time to execute and end the entire pipeline

Table 5.5: First 100 metrics of correlation over *Speed-Up*.

Nr.	Metric	Spearman Correlation
1	smsp__average_inst_executed_pipe_lsu_per_warp.ratio(Avg)	0.789
2	sm__sass_inst_executed_op_st.sum(Avg)	0.760
3	smsp__sass_inst_executed_op_st.sum(Avg)	0.760
4	smsp__average_inst_executed_pipe_lsu_per_warp.pct(Avg)	0.750
5	smsp__sass_inst_executed_op_st.max(Avg)	0.738
6	sm__sass_inst_executed_op_st.max(Avg)	0.732
7	smsp__warp_issue_stalled_drain_per_warp_active.pct(Avg)	-0.702
8	lltex__data_pipe_lsu_wavefronts.cmd_read.sum(Avg)	0.702
9	lltex__data_pipe_lsu_wavefronts.cmd_read.avg(Avg)	0.701
10	lltex__data_pipe_lsu_wavefronts.cmd_read.max(Avg)	0.696
11	sm__sass_inst_executed_op_st.min(Avg)	0.694
12	smsp__average_warp_latency_issue_stalled_no_instruction.ratio(Avg)	0.691
13	lltex__data_pipe_lsu_wavefronts.avg(Avg)	0.690
14	smsp__average_warp_latency_issue_stalled_not_selected.pct(Avg)	0.690
15	smsp__average_warp_latency_issue_stalled_not_selected.ratio(Avg)	0.690
16	DtoHTimeMax(ns)	-0.689
17	DtoHTimeMed(ns)	-0.689
18	smsp__average_warp_latency_issue_stalled_no_instruction.pct(Avg)	0.689
19	sm__sass_inst_executed_op_memory_32b.max(Avg)	0.688
20	DtoHTimeAvg(ns)	-0.688
21	tpc__lltex_data_bank_reads.max(Avg)	0.688
22	smsp__sass_inst_executed_op_memory_32b.max(Avg)	0.687
23	lltex__data_pipe_lsu_wavefronts.max(Avg)	0.686
24	lltex__data_pipe_lsu_wavefronts.sum(Avg)	0.685
25	sm__sass_thread_inst_executed_op_memory_pred_on.max(Avg)	0.685
26	smsp__average_inst_executed_per_warp.ratio(Avg)	0.684
27	smsp__average_warp_latency.pct(Avg)	0.684
28	smsp__average_warp_latency.ratio(Avg)	0.684
29	tpc__lltex_data_bank_reads.min(Avg)	0.682
30	tpc__lltex_data_pipe_lsu_wavefronts.max(Avg)	0.682
31	lltex__average_t_sectors_per_request.ratio(Avg)	0.681
32	tpc__lltex_data_pipe_lsu_wavefronts.avg(Avg)	0.681
33	tpc__lltex_data_bank_reads.sum(Avg)	0.681
34	lltex__average_t_sectors_per_request_pipe_lsu.ratio(Avg)	0.681
35	DtoHTimeMin(ns)	-0.680
36	smsp__sass_thread_inst_executed_op_memory_pred_on.max(Avg)	0.680
37	smsp__average_warp_latency_issue_stalled_wait.ratio(Avg)	0.680
38	smsp__average_warp_latency_issue_stalled_selected.ratio(Avg)	0.679
39	smsp__average_inst_executed_pipe_fma_per_warp.ratio(Avg)	0.679
40	smsp__average_warp_latency_issue_stalled_dispatch_stall.ratio(Avg)	0.679
41	lltex__data_bank_reads.max(Avg)	0.679
42	smsp__average_warp_latency_issue_stalled_wait.pct(Avg)	0.679
43	tpc__lltex_data_pipe_lsu_wavefronts.sum(Avg)	0.679
44	lltex__data_pipe_lsu_wavefronts.cmd_write.max(Avg)	0.678
45	smsp__average_warp_latency_issue_stalled_dispatch_stall.pct(Avg)	0.678
46	tpc__lltex_data_pipe_lsu_wavefronts.min(Avg)	0.674
47	smsp__sass_inst_executed_op_ld.max(Avg)	0.673
48	sm__sass_inst_executed_op_ld.max(Avg)	0.673
49	lltex__data_pipe_lsu_wavefronts.cmd_write.avg(Avg)	0.670
50	smsp__average_inst_executed_per_warp.pct(Avg)	0.669

Table 5.6: Remaining 50 metrics of correlation over *Speed-Up* (continue of Table 5.5).

Nr.	Metric	Spearman Correlation
51	smsp__average_inst_executed_pipe_alu_per_warp.ratio(Avg)	0.669
52	sm__mio_pq_write_cycles_active_pipe_lsu.max(Avg)	0.669
53	l1tex__data_bank_reads.sum(Avg)	0.668
54	smsp__inst_executed_pipe_lsu.max(Avg)	0.667
55	smsp__l1tex_lsuin_requests.max(Avg)	0.665
56	smsp__thread_inst_executed_pipe_lsu_pred_on.max(Avg)	0.665
57	sm__thread_inst_executed_pipe_lsu_pred_on.max(Avg)	0.664
58	sm__mio_inst_issued.max(Avg)	0.664
59	sm__mio_pq_read_cycles_active.max(Avg)	0.660
60	sm__mio_pq_write_cycles_active_pipe_lsu.sum(Avg)	0.660
61	sm__mio_pq_read_cycles_active_pipe_lsu.max(Avg)	0.659
62	l1tex__lsuin_requests.max(Avg)	0.658
63	sm__inst_executed_pipe_lsu.max(Avg)	0.658
64	l1tex__lsu_writeback_active.max(Avg)	0.658
65	DtoHSizeAvg(MB)	-0.657
66	DtoHSizeMed(MB)	-0.657
67	DtoHSizeMin(MB)	-0.657
68	DtoHSizeMax(MB)	-0.657
69	sm__sass_inst_executed_op_memory_32b.sum(Avg)	0.655
70	smsp__sass_inst_executed_op_memory_32b.sum(Avg)	0.655
71	l1tex__data_pipe_lsu_wavefronts_cmd_read.min(Avg)	0.655
72	l1tex__lsu_writeback_active.sum(Avg)	0.654
73	l1tex__data_pipe_lsu_wavefronts.min(Avg)	0.654
74	lts__t_requests_srcnode_gpc_op_read_lookup_hit.avg(Avg)	0.653
75	smsp__average_inst_executed_pipe_fma_per_warp.pct(Avg)	0.650
76	lts__t_requests_srcnode_gpc_lookup_hit.avg(Avg)	0.649
77	smsp__sass_inst_executed_op_st.min(Avg)	0.648
78	lts__t_requests_srcunit_tex_op_read_lookup_hit.max(Avg)	0.647
79	smsp__warp_issue_stalled_drain_per_warp_active.ratio(Avg)	-0.646
80	lts__t_requests_srcunit_tex_lookup_hit.max(Avg)	0.645
81	sm__sass_inst_executed_op_ld.sum(Avg)	0.644
82	smsp__sass_inst_executed_op_ld.sum(Avg)	0.644
83	sm__mio_pq_read_cycles_active_pipe_lsu.sum(Avg)	0.643
84	smsp__mio2rf_writeback_active_pipe_lsu.max(Avg)	0.642
85	sm__sass_thread_inst_executed_op_memory_pred_on.sum(Avg)	0.641
86	smsp__sass_thread_inst_executed_op_memory_pred_on.sum(Avg)	0.641
87	sm__inst_executed_pipe_lsu.sum(Avg)	0.640
88	smsp__inst_executed_pipe_lsu.sum(Avg)	0.640
89	lts__t_requests_srcnode_gpc_lookup_hit.min(Avg)	0.640
90	lts__t_requests_srcunit_tex_lookup_hit.min(Avg)	0.640
91	sm__mio_pq_read_cycles_active.sum(Avg)	0.639
92	lts__t_requests_lookup_hit.min(Avg)	0.639
93	lts__t_tag_requests_hit.min(Avg)	0.639
94	smsp__average_warp_latency_issue_stalled_branch_resolving.ratio(Avg)	0.639
95	smsp__sass_inst_executed_op_memory_32b.avg(Avg)	0.639
96	smsp__amortized_warp_latency.pct(Avg)	0.639
97	smsp__amortized_warp_latency.ratio(Avg)	0.639
98	sm__sass_inst_executed_op_memory_32b.min(Avg)	0.639
99	sm__thread_inst_executed_pipe_lsu_pred_on.sum(Avg)	0.638
100	smsp__thread_inst_executed_pipe_lsu_pred_on.sum(Avg)	0.638

Chapter 6

Thesis Conclusion

The first explored and detailed point in this thesis involved the study of optimization techniques developed over the years. The substantial growth in industrialization and the prevalence of the Internet of Things (IoT) have resulted in the widespread use of various types of devices in both professional and non-professional settings. These devices, including wearables, handheld gadgets, and embedded systems like smartphones, etc., often encounter constraints related to energy consumption. These devices are equipped with a System on Chip (SoC) that integrates various specialized processing engines designed for specific tasks, including modules for neural network processing, data transmission over networks and IoT devices, and programmable massively parallel accelerators like Graphics Processing Units (GPUs). This initial analysis of optimization techniques was fundamental to the subsequent studies conducted in this thesis. Delving into the composition of a GPU and understanding various Last Level Cache (LLC) replacement policies, as well as new optimization techniques and the two main types of computational workloads, compute and graphical workloads, laid the foundation for further research. Subsequently, we explored various macro-areas of optimization. The first of these focused on memory access optimization, which included the study of Memory Locality-aware Cache Replacement (MLCR), the introduction of the Tile Sequencer, and an improved variant of PseudoLRU based on insertion/promotion vectors (IPV). Another macro-area was the selective caching, where different techniques for reusing cached data, avoiding victimizing a cache line that could potentially be reused in the near future by the executing task, were examined. Regarding compute optimization, it was highlighted that both compute and graphics workloads can exhibit periodic behavior. This redundancy is then exploited by using the memoization through Look-Up Tables, Tile-Based Deferred Rendering (TBDR), Parallel Frame Rendering (PFR), RAVEN and also in the Ray Tracing field.

The second focal point of this thesis, driven by a lack of comprehensive studies on underutilized GPU resources, centered on maximizing the potential of GPUs. This objective can only be attained through the use of a partitionable GPU capable of executing multiple tasks simultaneously. By customizing the well-known GPGPU-Sim simulator, has been tackled this initial challenge. However, the ability to run various tasks concurrently opened the door to latency and interference issues arising from simultaneously shared hardware components. These inter-task interferences cannot be disregarded, which leads to the next part of the focus: developing a model capable of capturing these interferences based on the behaviors of the different concurrently executed tasks. Starting from a profiling of individual kernels in isolation, has been created a model capable of predicting execution latencies and the required memory bandwidth as the number of assigned SMs varies in isolated scenarios. The model exhibited errors ranging from underestimations of 30% to overestimations of up to 10%, highlighting the challenge of estimating the behavior of high-bandwidth-demand kernels. The subsequent step involved the development of a predictive model to forecast latencies in the presence of

interferences. The errors derived from the predictive model were based on 107 experiments involving the concurrent execution of 2, 3, 4, 5, or 6 kernels, resulting in a total of 420 single kernel predictions. Of these, 135 instances were underestimations, and 285 were overestimations, with an average error rate of 10.7% and a variance of 2.71%. The maximum sporadic overestimation was 76%, occurring in only 1 out of 420 cases, while the minimum sporadic underestimation was 20%, also occurring in 1 out of 420 cases, which is roughly 0.2%. To summarize the findings, a predictive model capable of estimating the number of execution cycles required for the completion of a single kernel, whether in isolation or concurrently with other kernels, has been successfully developed. This achievement resulted in an average absolute error of 11.23%.

To effectively harness the models presented in this part of the thesis, potential future developments could involve conducting additional tests and studies in the field of task schedulability in real-time systems. With a focus on partitionable GPUs, the developed models could be employed to proactively predict the schedulability of a task, running concurrently with other tasks. This prediction would take into account the type of tasks already in execution, the interferences at play, and the potential slowdowns the task might experience when scheduled. These models can also be utilized for offline analysis of the task schedulability problem to optimize the hardware potential provided by GPUs and minimize latencies resulting from concurrent executions.

The third focus was centered around heterogeneous embedded platforms. There exists a significant knowledge gap in the current state of research concerning these architectures, which are employed in various latency-sensitive applications like the automotive domain, IoT real-time systems, smart cities, and more. To date, no comprehensive study has delved into the behavior of these embedded accelerators when subjected to high computational workloads. Understanding how the parallel computational engines in these modern platforms may interact and subsequently affect performance and predictability remains a substantial research challenge. For example, scenarios where GPU-accelerated tasks run concurrently with other CPU tasks are commonplace in a wide range of applications, as previously mentioned. In order to fill this gap, has been conducted an in-depth analysis of GPU kernel behavior under varying memory-intensive CPU-side interference scenarios. To predict GPU kernel slowdown resulting from intensive CPU-side memory activity, various machine learning (ML) methodologies, including SVR, RFR, DeepNN, and a combined approach, were employed. The ML methodologies underwent training with fine-grained profiling data, made accessible through Nvidia’s profiling tool, *ncu*. The combined approach yielded favorable outcomes, with a mean and absolute error percentage of -0.19% and 9.19%, respectively. Additionally, well-established dataset analysis techniques such as PCA, GINI, and Spearman were employed to identify 18 out of the initial 45 *ncu* metrics as the most influential predictors. This finding was further substantiated by retraining the predictive model using only the 18 selected metrics. Consequently, the predictive model trained with this restricted feature set demonstrated an average and absolute percentage error of 2.79% and 10.24%, respectively. A combined predictive model of this nature can prove invaluable for assessing system behavior under varying computational workloads between the CPU and GPU. Consequently, it has the potential to become an essential tool for response time analysis and memory interference-aware scheduling. Regarding future prospects, the contend that the pivotal metrics identified through the proposed approach offer an opportunity to enhance existing state-of-the-art heterogeneous task models within real-time literature. Specifically to the DAG-based task models, which represent jobs as interconnected nodes within a directed acyclic graph. Each node is associated with various timing parameters such as period, WCET, deadline, offset, and more. In heterogeneous DAGs, nodes are designated to run on specific accelerators. Herein lies the significance of these findings: the identified key metrics for predicting memory interference

can be incorporated to enrich the associated parameters for each node, empowering system engineers to consider memory interference when deploying task sets on real hardware.

In the concluding chapter, the groundwork has been laid for a comprehensive analysis of two distinct memory management methods in CUDA applications: UVM and COPY. This preliminary investigation has yielded valuable insights into their performance across diverse acceleration architectures, encompassing both discrete and embedded GPUs, and spanning a variety of task scenarios. The findings indicate that certain tasks stand to benefit from the utilization of UVM memory management, as evidenced by the significant performance improvements observed in PF and ATAX kernels across both architectures. Conversely, for MVT and HISTO kernels, performance enhancements are observed solely on embedded platforms (Orin and AGX Xavier), attributed to the reduced page migration time inherent to embedded systems.

However, in cases such as NBODY and BFS, the adoption of UVM does not result in a significant degradation of performance, despite a mere 8% timing slowdown. This suggests that the investment of time and resources into developing a bespoke memory management solution may not necessarily translate into a substantial enhancement in execution performance, thus prompting a reconsideration of the value proposition associated with such implementations.

Subsequently, a deeper analysis of the profiled metrics was undertaken through PCA and Spearman analysis, providing further insights into task performance and relative speed-up. Through PCA, 40 metrics were identified that explain 85% of the variance, effectively synthesizing the initial oversized database of over 6270 metrics. Notable among these metrics are those pertaining to the Load Store Unit pipeline, memory clock frequency, cache write operation hit ratio, grid launch size, instructions executed by the divergence unit (ADU), and average memory barrier operations.

By employing Spearman ranking and focusing on the correlation between metrics and the task’s speed-up, the top 100 Spearman metrics were highlighted, some of which corresponded to results obtained through PCA. Notably, four metrics were identified through both PCA and Spearman analysis, including `smsp_average_warp_latency_ratio(Avg)`, `sm_mio_pq_read_cycles_active_pipe_lsu.max(Avg)`, `DtoHSizeMed(MB)`, `sm_mio_pq_read_cycles_active_pipe_lsu.sum(Avg)`.

These metrics represent valuable inputs for future research endeavors, facilitating the exploration of alternative methodologies such as neural networks and other machine learning techniques. Looking ahead, future research efforts will concentrate on developing predictive models tailored to forecast slowdown or speed-up in UVM usage scenarios tailored to various architectures.

6.1 Future Work

There are several avenues for future exploration. Firstly, based on the findings of Chapter 3, my intention as a future work is to evaluate the performance of the developed predictive model against other existing models which are close to the implemented approach. Additionally, there is potential to expand the findings by emulating a different range of GPU architectures, ensuring that the predictive model remains effective across various architectures. Furthermore, it may be necessary to revisit the predictive models built with GPGPU-Sim, as future GPUs may behave substantially differently. Additionally, many of the memory interference issues, in the case of partitioned GPUs, could be studied directly on real hardware, taking into account the current limitations of partitioning technologies such as NVIDIA MPS and MiG. Moreover, i aim to investigate the effect of SM masking, as proposed in [145], with the recent release of libsmctrl. This library allows for arbitrary partitioning of SM on NVIDIA hardware, and as demonstrated by the authors of libsmctrl, giving system engineers control over the scheduling

of blocks from different kernels can lead to performance improvements in real-world scenarios. In this context as well, i intend to examine how closely our simulated model aligns with the metrics obtained using this new GPU partitioning approach.

In Chapter 4, the next possible step would involve automating the analysis process for other embedded architectures. This comprehensive approach aims to create a unified model capable of anticipating execution latencies and interference patterns solely based on architectural metrics, thus streamlining the predictive process.

Looking forward, I will delve deeper into Chapter 5, focusing on developing a predictive machine learning model. This model aims to preemptively determine the optimal memory management methodology and quantify the potential performance gains offered by simplified UVM management compared to conventional COPY approaches.

The methodologies developed across the chapters present an opportunity to advance the current state-of-the-art in heterogeneous task modeling within real-time research. Particularly, in the context of DAG-based task models, which conceptualize jobs as interconnected nodes within directed acyclic graphs. The significance of these findings lies in their potential to enhance task modeling by incorporating key metrics for predicting memory interference into the associated parameters for each node. This integration enables system engineers to account for memory interference when deploying task sets on real hardware, thus contributing to more accurate and efficient system designs.

Bibliography

- [1] X. Larrucea, A. Combelles, J. Favaro, and K. Taneja, “Software engineering for the internet of things,” *IEEE Software*, vol. 34, no. 1, pp. 24–28, 2017.
- [2] J. M. Cebri’n, G. D. Guerrero, and J. M. Garcia, “Energy efficiency analysis of gpus,” in *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pp. 1014–1022, IEEE, 2012.
- [3] P. Varga, J. Peto, A. Franko, D. Balla, D. Haja, F. Janky, G. Soos, D. Ficzer, M. Maliosz, and L. Toka, “5g support for industrial iot applications—challenges, solutions, and research gaps,” *Sensors*, vol. 20, no. 3, p. 828, 2020.
- [4] Y. Yamato, T. Demizu, H. Noguchi, and M. Kataoka, “Automatic gpu offloading technology for open iot environment,” *IEEE Internet of Things Journal*, vol. 6, no. 2, pp. 2669–2678, 2018.
- [5] M. K. Elteir, S. Lazem, and M. Azab, “Unleashing the hidden powers of low-cost iot boards: Gpu-based edutainment case study,” *Journal of King Saud University-Computer and Information Sciences*, 2020.
- [6] A. Masola, C. Gabbi, A. Castellano, N. Capodieci, and P. Burgio, “Graphic interfaces in adas: From requirements to implementation,” in *Proceedings of the 6th EAI International Conference on Smart Objects and Technologies for Social Good*, pp. 193–198, 2020.
- [7] T. Allen and R. Ge, “Characterizing power and performance of gpu memory access,” in *2016 4th International Workshop on Energy Efficient Supercomputing (E2SC)*, pp. 46–53, IEEE, 2016.
- [8] A. Masola and N. Capodieci, “Optimization strategies for gpus: an overview of architectural approaches,” *International Journal of Parallel, Emergent and Distributed Systems*, pp. 1–15, 2023.
- [9] C. A. Navarro, N. Hitschfeld-Kahler, and L. Mateu, “A survey on parallel computing and its applications in data-parallel problems using gpu architectures,” *Communications in Computational Physics*, vol. 15, no. 2, pp. 285–329, 2014.
- [10] M. J. Flynn, “Very high-speed computing systems,” *Proceedings of the IEEE*, vol. 54, no. 12, pp. 1901–1909, 1966.
- [11] S. Hong and H. Kim, “An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness,” in *Proceedings of the 36th annual international symposium on Computer architecture*, pp. 152–163, 2009.

- [12] B. Wu, Z. Zhao, E. Z. Zhang, Y. Jiang, and X. Shen, “Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on gpu,” *ACM SIGPLAN Notices*, vol. 48, no. 8, pp. 57–68, 2013.
- [13] N. Capodieci, R. Cavicchioli, I. S. Olmedo, M. Solieri, and M. Bertogna, “Contending memory in heterogeneous socs: Evolution in nvidia tegra embedded platforms,” in *2020 IEEE 26th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pp. 1–10, IEEE, 2020.
- [14] P. Gera, H. Kim, H. Kim, S. Hong, V. George, and C.-K. Luk, “Performance characterisation and simulation of intel’s integrated gpu architecture,” in *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 139–148, IEEE, 2018.
- [15] C. Li, Y. Yang, H. Dai, S. Yan, F. Mueller, and H. Zhou, “Understanding the trade-offs between software-managed vs. hardware-managed caches in gpus,” in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 231–242, 2014.
- [16] I. S. Olmedo, N. Capodieci, J. L. Martinez, A. Marongiu, and M. Bertogna, “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.
- [17] M. Doggett, “Texture caches,” *IEEE Micro*, vol. 32, no. 3, pp. 136–141, 2012.
- [18] S. Cook, *CUDA programming: a developer’s guide to parallel computing with GPUs*. Newnes, 2012.
- [19] Z. S. Hakura and A. Gupta, “The design and analysis of a cache architecture for texture mapping,” in *Proceedings of the 24th annual international symposium on Computer architecture*, pp. 108–120, 1997.
- [20] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, “Demystifying gpu microarchitecture through microbenchmarking,” in *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pp. 235–246, IEEE, 2010.
- [21] X. Mei and X. Chu, “Dissecting gpu memory hierarchy through microbenchmarking,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, pp. 72–86, 2016.
- [22] F. Paulsson and S. Senanayake, “A quantitative evaluation and proposition of cache policies in mobile gpus,” *LU-CS-EX 2016-17*, 2016.
- [23] S. Kumar and P. Singh, “An overview of modern cache memory and performance analysis of replacement policies,” in *2016 IEEE International Conference on Engineering and Technology (ICETECH)*, pp. 210–214, IEEE, 2016.
- [24] H. S. Stone, “A logic-in-memory computer,” *IEEE Transactions on Computers*, vol. 100, no. 1, pp. 73–78, 1970.
- [25] P. C. Patton, “Trends in data organization and access methods,” *Computer*, vol. 3, no. 6, p. 18–24, 1970.

- [26] K. So and R. N. Rechtschaffen, “Cache operations by mru change,” *IEEE Transactions on Computers*, vol. 37, no. 6, pp. 700–709, 1988.
- [27] M. Chaudhuri, “Pseudo-lifo: the foundation of a new family of replacement policies for last-level caches,” in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 401–412, IEEE, 2009.
- [28] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.
- [29] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, “Adaptive insertion policies for high performance caching,” *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, pp. 381–391, 2007.
- [30] V. Bharadwaj and V. Kachapur, “Nru cache replacement policy for multicore processors using multi2sim,” *International Research Journal of Engineering and Technology (IRJET)*, 2018.
- [31] A.-C. P. Yang, T.-C. Wang, and J.-P. Sheu, “A least frequently used (lfu) algorithm for page replacement,” *ACM Transactions on Computer Systems (TOCS)*, vol. 1, no. 2, pp. 157–171, 1981.
- [32] N. N. Mekhiel, “Multi-level cache with most frequently used policy: A new concept in cache design,” in *Proceedings of the International Conference on Computer Applications in Industry and Engineering*, 1995.
- [33] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer, “High performance cache replacement using re-reference interval prediction (rrip),” *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, pp. 60–71, 2010.
- [34] R. Karedla, J. S. Love, and B. G. Wherry, “Caching strategies to improve disk system performance,” *Computer*, vol. 27, no. 3, pp. 38–46, 1994.
- [35] H. Gao and C. Wilkerson, “A dueling segmented lru replacement algorithm with adaptive bypassing,” in *JWAC 2010-1st JILP Workshop on Computer Architecture Competitions: cache replacement Championship*, 2010.
- [36] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, “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.
- [37] B. Forsberg, L. Benini, and A. Marongiu, “Taming data caches for predictable execution on gpu-based socs,” in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 650–653, IEEE, 2019.
- [38] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder, “The gpu as numerical simulation engine,” in *SIGGRAPH 2003*, 2003.
- [39] N. Capodieci, R. Cavicchioli, and A. Marongiu, “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.
- [40] M. Garland, M. Kudlur, and Y. Zheng, “Designing a unified programming model for heterogeneous machines,” in *SC’12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, IEEE, 2012.

- [41] K. Bhargavi and B. S. Babu, “Accelerating the big data analytics by gpu-based machine learning: A survey,” in *International Symposium on Sensor Networks, Systems and Security*, pp. 63–83, Springer, 2017.
- [42] T. Kalaiselvi, P. Sriramakrishnan, and K. Somasundaram, “Survey of using gpu cuda programming model in medical image analysis,” *Informatics in Medicine Unlocked*, vol. 9, pp. 133–144, 2017.
- [43] O. Navarro-Hinojosa, S. Ruiz-Loza, and M. Alencastre-Miranda, “Physically based visual simulation of the lattice boltzmann method on the gpu: a survey,” *The Journal of Supercomputing*, vol. 74, no. 7, pp. 3441–3467, 2018.
- [44] G. Ceballos, A. Sembrant, T. E. Carlson, and D. Black-Schaffer, “Behind the scenes: Memory analysis of graphical workloads on tile-based gpus,” in *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 1–11, IEEE, 2018.
- [45] G. Ceballos, A. Sembrant, T. E. Carlson, and D. Black-Schaffer, “Analyzing graphics workloads on tile-based gpus,” in *2017 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 108–109, IEEE, 2017.
- [46] C.-C. Hsiao, S.-L. Chu, and S.-S. Dai, “Efficient rendering and cache replacement mechanisms for hierarchical tiling in mobile gpus,” in *2012 IEEE Global High Tech Congress on Electronics*, pp. 170–175, IEEE, 2012.
- [47] F. K. H. Abadi and S. Safari, “Performance and area aware replacement policy for gpu architecture,” in *2014 4th International Conference on Computer and Knowledge Engineering (ICCKE)*, pp. 497–503, IEEE, 2014.
- [48] D. A. Jiménez, “Insertion and promotion for tree-based pseudolru last-level caches,” in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 284–296, 2013.
- [49] N. Agarwal, D. Nellans, E. Ebrahimi, T. F. Wenisch, J. Danskin, and S. W. Keckler, “Selective gpu caches to eliminate cpu-gpu hw cache coherence,” in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 494–506, IEEE, 2016.
- [50] Y. Huangfu and W. Zhang, “Profiling-based l1 data cache bypassing to improve gpu performance and energy efficiency,” *ACM SIGBED Review*, vol. 12, no. 1, pp. 7–11, 2015.
- [51] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, “Rodinia: A benchmark suite for heterogeneous computing,” in *2009 IEEE international symposium on workload characterization (IISWC)*, pp. 44–54, Ieee, 2009.
- [52] W. Jia, K. A. Shaw, and M. Martonosi, “Characterizing and improving the use of demand-fetched caches in gpus,” in *Proceedings of the 26th ACM international conference on Supercomputing*, pp. 15–24, 2012.
- [53] B. Li, J. Sun, M. Annavaram, and N. S. Kim, “Elastic-cache: Gpu cache architecture for efficient fine-and coarse-grained cache-line management,” in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 82–91, IEEE, 2017.

- [54] S. Kumar, H. Zhao, A. Shriraman, E. Matthews, S. Dwarkadas, and L. Shannon, “Amoeba-cache: Adaptive blocks for eliminating waste in the memory hierarchy,” in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 376–388, IEEE, 2012.
- [55] T. Allen and R. Ge, “In-depth analyses of unified virtual memory system for gpu accelerated computing,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15, 2021.
- [56] J. Power, A. Basu, J. Gu, S. Puthoor, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood, “Heterogeneous system coherence for integrated cpu-gpu systems,” in *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 457–467, IEEE, 2013.
- [57] N. Suzuki, H. Kim, D. De Niz, B. Andersson, L. Wrage, M. Klein, and R. Rajkumar, “Coordinated bank and cache coloring for temporal protection of memory accesses,” in *2013 IEEE 16th International Conference on Computational Science and Engineering*, pp. 685–692, IEEE, 2013.
- [58] S. Mittal, “A survey of techniques for cache partitioning in multicore processors,” *ACM Computing Surveys (CSUR)*, vol. 50, no. 2, pp. 1–39, 2017.
- [59] P.-H. Wang, C.-H. Li, and C.-L. Yang, “Latency sensitivity-based cache partitioning for heterogeneous multi-core architecture,” in *Proceedings of the 53rd Annual Design Automation Conference*, pp. 1–6, 2016.
- [60] M. Hall and J. P. McNamee, “Improving software performance with automatic memoization,” *Johns Hopkins APL Technical Digest*, vol. 18, no. 2, p. 255, 1997.
- [61] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, “Eliminating redundant fragment shader executions on a mobile gpu via hardware memoization,” *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 529–540, 2014.
- [62] A. Lauritzen, “Deferred rendering for current and future rendering pipelines,” *SIGGRAPH Course: Beyond Programmable Shading*, pp. 1–34, 2010.
- [63] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, “Parallel frame rendering: Trading responsiveness for energy on a mobile gpu,” in *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pp. 83–92, IEEE, 2013.
- [64] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, “Teapot: A toolset for evaluating performance, power and image quality on mobile graphics systems,” in *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pp. 37–46, 2013.
- [65] M. Anglada, E. de Lucas, J.-M. Parcerisa, J. L. Aragón, P. Marcuello, and A. González, “Rendering elimination: Early discard of redundant tiles in the graphics pipeline,” in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 623–634, IEEE, 2019.
- [66] C. Hwang, S. Pushp, C. Koh, J. Yoon, Y. Liu, S. Choi, and J. Song, “Raven: Perception-aware optimization of power consumption for mobile games,” in *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, pp. 422–434, 2017.

- [67] A. Hore and D. Ziou, “Image quality metrics: Psnr vs. ssim,” in *2010 20th international conference on pattern recognition*, pp. 2366–2369, IEEE, 2010.
- [68] H. Zhou, Y. Fu, and C. Liu, “Supporting dynamic {GPU} computing result reuse in the cloud,” in *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, 2015.
- [69] H. Tabani, J.-M. Arnau, J. Tubella, and A. Gonzalez, “An ultra low-power hardware accelerator for acoustic scoring in speech recognition,” in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 41–52, IEEE, 2017.
- [70] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke, “Paraprox: Pattern-based approximation for data parallel applications,” in *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pp. 35–50, 2014.
- [71] J. Gunther, S. Popov, H.-P. Seidel, and P. Slusallek, “Realtime ray tracing on gpu with bvh-based packet traversal,” in *2007 IEEE Symposium on Interactive Ray Tracing*, pp. 113–118, IEEE, 2007.
- [72] T. Aila and S. Laine, “Understanding the efficiency of ray traversal on gpus,” in *Proceedings of the conference on high performance graphics 2009*, pp. 145–149, 2009.
- [73] L. Liu, W. Chang, F. Demoullin, Y. H. Chou, M. Saed, D. Pankratz, T. Nowicki, and T. M. Aamodt, “Intersection prediction for accelerated gpu ray tracing,” in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 709–723, 2021.
- [74] Y. He, E. B. Lum, E. Enderton, H. P. Moreton, and K. Fatahalian, “Adaptive shading in a graphics processing pipeline,” Jan. 24 2017. US Patent 9,552,667.
- [75] E. B. Lum, R. L. Dimitrov, I. L. Ubieta, P. J. Neill, Y. Uralsky, and A. Meixner, “Variable fragment shading with surface recasting,” May 31 2016. US Patent 9,355,483.
- [76] C. A. Burns and W. A. Hunt, “The visibility buffer: a cache-friendly approach to deferred shading,” *Journal of Computer Graphics Techniques (JCGT)*, vol. 2, no. 2, pp. 55–69, 2013.
- [77] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, “Boosting mobile gpu performance with a decoupled access/execute fragment processor,” *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3, pp. 84–93, 2012.
- [78] J. Gaur, R. Srinivasan, S. Subramoney, and M. Chaudhuri, “Efficient management of last-level caches in graphics processors for 3d scene rendering workloads,” in *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 395–407, IEEE, 2013.
- [79] E. De Lucas, P. Marcuello, J.-M. Parcerisa, and A. Gonzalez, “Visibility rendering order: Improving energy efficiency on mobile gpus through frame coherence,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 2, pp. 473–485, 2018.
- [80] Y. Sun, N. B. Agostini, S. Dong, and D. Kaeli, “Summarizing cpu and gpu design trends with product data,” *arXiv preprint arXiv:1911.11313*, 2019.

- [81] A. K. et al., “Tango: A deep neural network benchmark suite for various accelerators,” *CoRR*, vol. abs/1901.04987, 2019.
- [82] S. e. a. Pai, “Improving GPGPU concurrency with elastic kernels,” in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS ’13*, (New York, NY, USA), p. 407–418, Association for Computing Machinery, 2013.
- [83] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo, “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.
- [84] S. Williams, A. Waterman, and D. Patterson, “Roofline: an insightful visual performance model for multicore architectures,” *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.
- [85] E. Konstantinidis and Y. Cotronis, “A quantitative roofline model for gpu kernel performance estimation using micro-benchmarks and hardware metric profiling,” *Journal of Parallel and Distributed Computing*, vol. 107, pp. 37–56, 2017.
- [86] A. Masola, N. Capodiecici, R. Cavicchioli, I. S. Olmedo, and B. Rouxel, “Memory-aware latency prediction model for concurrent kernels in partitionable gpus: Simulations and experiments,” in *Workshop on Job Scheduling Strategies for Parallel Processing*, pp. 46–73, Springer, 2023.
- [87] K. Gupta, J. A. Stuart, and J. D. Owens, “A study of persistent threads style gpu programming for gpgpu workloads,” in *2012 Innovative Parallel Computing (InPar)*, pp. 1–14, 2012.
- [88] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, “Auto-tuning a high-level language targeted to gpu codes,” in *2012 innovative parallel computing (InPar)*, pp. 1–10, Ieee, 2012.
- [89] H.-E. Zahaf, I. S. Olmedo, J. Singh, N. Capodiecici, and S. Faucou, “Contention-aware gpu partitioning and task-to-partition allocation for real-time workloads,” in *Proceedings of the 29th International Conference on Real-Time Networks and Systems*, pp. 226–236, 2021.
- [90] T. Leibovich-Raveh, D. J. Lewis, S. A.-R. Kadhim, and D. Ansari, “A new method for calculating individual subitizing ranges,” *Journal of Numerical Cognition*, vol. 4, no. 2, pp. 429–447, 2018.
- [91] A. Jog, O. Kayiran, T. Kesten, A. Pattnaik, E. Bolotin, N. Chatterjee, S. W. Keckler, M. T. Kandemir, and C. R. Das, “Anatomy of gpu memory system for multi-application execution,” in *Proceedings of the 2015 International Symposium on Memory Systems*, pp. 223–234, 2015.
- [92] F. Wartel, L. Kosmidis, A. Gogonel, A. Baldovino, Z. Stephenson, B. Triquet, E. Quinones, C. Lo, E. Mezzetta, I. Broster, *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.

- [93] K. P. Silva, L. F. Arcaro, and R. S. De Oliveira, “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.
- [94] B. Rouxel, S. Skalistis, S. Derrien, and I. Puaut, “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.
- [95] Q. Hu, J. Shu, J. Fan, and Y. Lu, “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.
- [96] A. Jog, E. Bolotin, Z. Guz, M. Parker, S. W. Keckler, M. T. Kandemir, and C. R. Das, “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.
- [97] H. Dai, Z. Lin, C. Li, C. Zhao, F. Wang, N. Zheng, and H. Zhou, “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.
- [98] T. Yandrofski, J. Chen, N. Otterness, J. H. Anderson, and F. D. Smith, “Making powerful enemies on nvidia gpus,” in *2022 IEEE Real-Time Systems Symposium (RTSS)*, pp. 383–395, IEEE, 2022.
- [99] X. Zhao, M. Jahre, and L. Eeckhout, “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.
- [100] Q. Xu, H. Jeon, K. Kim, W. W. Ro, and M. Annavaram, “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.
- [101] W. Thomas, S. Toraskar, and V. Singh, “Dynamic optimizations in gpu using roofline model,” in *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5, IEEE, 2021.
- [102] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte, “The case for gpgpu spatial multitasking,” in *IEEE International Symposium on High-Performance Comp Architecture*, pp. 1–12, IEEE, 2012.
- [103] Y. Ukidave, C. Kalra, D. Kaeli, P. Mistry, and D. Schaa, “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.
- [104] P. Aguilera, K. Morrow, and N. S. Kim, “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.

- [105] E. C. Marangoz, K.-D. Kang, and S. Shin, “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.
- [106] S. Bak, G. Yao, R. Pellizzoni, and M. Caccamo, “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.
- [107] R. Cavicchioli, N. Capodieci, and M. Bertogna, “Memory interference characterization between cpu cores and integrated gpus in mixed-criticality platforms,” in *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pp. 1–10, IEEE, 2017.
- [108] Q. Zhu, B. Wu, X. Shen, K. Shen, L. Shen, and Z. Wang, “Understanding co-run performance on cpu-gpu integrated processors: observations, insights, directions,” *Frontiers of Computer Science*, vol. 11, pp. 130–146, 2017.
- [109] D. Shingari, A. Arunkumar, and C.-J. Wu, “Characterization and throttling-based mitigation of memory interference for heterogeneous smartphones,” in *2015 IEEE International Symposium on Workload Characterization*, pp. 22–33, IEEE, 2015.
- [110] M. Ditty, A. Karandikar, and D. Reed, “Nvidia’s xavier soc,” in *Hot chips: a symposium on high performance chips*, 2018.
- [111] C. J. Burges, “A tutorial on support vector machines for pattern recognition,” *Data Mining and Knowledge Discovery*, pp. 121–167, 1998.
- [112] L. Breiman, “Random forests,” *Machine Learning*, pp. 5–32, 2001.
- [113] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” *Data Mining and Knowledge Discovery*, pp. 436–444, 2015.
- [114] Z. Li and L. Wang, “Memory-aware scheduling for mixed-criticality systems,” in *Computational Science and Its Applications–ICCSA 2016: 16th International Conference, Beijing, China, July 4–7, 2016, Proceedings, Part II 16*, pp. 140–156, Springer, 2016.
- [115] Z. Lei, X. Lei, and J. Long, “Memory-aware scheduling parallel real-time tasks for multicore systems,” *International Journal of Software Engineering and Knowledge Engineering*, vol. 31, no. 04, pp. 613–634, 2021.
- [116] A. Masola, N. Capodieci, B. Rouxel, G. Franchini, and R. Cavicchioli, “Machine learning techniques for understanding and predicting memory interference in cpu-gpu embedded systems,” in *2023 IEEE 29th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pp. 147–156, IEEE Computer Society, 2023.
- [117] A. Podzimek, L. Bulej, L. Y. Chen, W. Binder, and P. Tuma, “Analyzing the impact of cpu pinning and partial cpu loads on performance and energy efficiency,” in *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pp. 1–10, IEEE, 2015.
- [118] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda, “Reducing memory interference in multicore systems via application-aware memory channel partitioning,” in *Proceedings of the 44th annual IEEE/ACM international symposium on microarchitecture*, pp. 374–385, 2011.

- [119] H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni, “Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms,” in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 155–166, IEEE, 2014.
- [120] W. Zhao, Q. Chen, and M. Guo, “Ksm: Online application-level performance slowdown prediction for spatial multitasking gpgpu,” *IEEE Computer Architecture Letters*, vol. 17, no. 2, pp. 187–191, 2018.
- [121] W. Zhao, Q. Chen, H. Lin, J. Zhang, J. Leng, C. Li, W. Zheng, L. Li, and M. Guo, “Themis: Predicting and reining in application-level slowdown on spatial multitasking gpus,” in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 653–663, IEEE, 2019.
- [122] W. Ali and H. Yun, “Protecting real-time gpu kernels on integrated cpu-gpu soc platforms,” *arXiv preprint arXiv:1712.08738*, 2017.
- [123] N. Capodieci, R. Cavicchioli, P. Valente, and M. Bertogna, “Sigamma: Server based integrated gpu arbitration mechanism for memory accesses,” in *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, pp. 48–57, 2017.
- [124] P. Houdek, M. Sojka, and Z. Hanzálek, “Towards predictable execution model on arm-based heterogeneous platforms,” in *2017 IEEE 26th International Symposium on Industrial Electronics (ISIE)*, pp. 1297–1302, IEEE, 2017.
- [125] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley, “A predictable execution model for cots-based embedded systems,” in *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 269–279, IEEE, 2011.
- [126] A. Saeed, D. Mueller-Gritschneider, F. Rehm, A. Hamann, D. Ziegenbein, U. Schlichtmann, and A. Gerstlauer, “Learning based memory interference prediction for co-running applications on multi-cores,” in *2021 ACM/IEEE 3rd Workshop on Machine Learning for CAD (MLCAD)*, pp. 1–6, IEEE, 2021.
- [127] N. Mishra, J. D. Lafferty, and H. Hoffmann, “Esp: A machine learning approach to predicting application interference,” in *2017 IEEE International Conference on Autonomic Computing (ICAC)*, pp. 125–134, IEEE, 2017.
- [128] S. Kim and Y. Kim, “Co-scheml: interference-aware container co-scheduling scheme using machine learning application profiles for gpu clusters,” in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 104–108, IEEE, 2020.
- [129] M. Ayub and T. Helmy, “Concurrent kernel execution and interference analysis on gpus using deep learning approaches,” *Journal of King Saud University-Computer and Information Sciences*, vol. 34, no. 10, pp. 10193–10204, 2022.
- [130] R. Bramley, Y. Huang, G. Duan, N. Saxena, and P. Racunas, “On the measurement of safe fault failure rates in high-performance compute processors,” in *2020 IEEE International Test Conference (ITC)*, pp. 1–10, IEEE, 2020.
- [131] A. Saiz, P. Prieto, P. Abad, J. A. Gregorio, and V. Puente, “Top-down performance profiling on nvidia’s gpus,” in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 179–189, IEEE, 2022.

- [132] P. Bauman, N. Chalmers, N. Curtis, C. Freitag, J. Greathouse, N. Malaya, D. McDougall, S. Moe, R. van Oostrum, N. Wolfe, *et al.*, “Introduction to amd gpu programming with hip,” *Presentation at Oak Ridge National Laboratory.*, 2019.
- [133] J. H. Ryoo, S. J. Quirem, M. Lebeane, R. Panda, S. Song, and L. K. John, “Gpgpu benchmark suites: How well do they sample the performance spectrum?,” in *2015 44th International Conference on Parallel Processing*, pp. 320–329, IEEE, 2015.
- [134] Y. Kim and A. Shrivastava, “Cumapz: A tool to analyze memory access patterns in cuda,” in *Proceedings of the 48th design automation conference*, pp. 128–133, 2011.
- [135] G. Franchini, V. Ruggiero, F. Porta, and L. Zanni, “Neural architecture search via standard machine learning methodologies,” *Mathematics in Engineering*, pp. 1–21, 2023.
- [136] S. Bonettini, G. Franchini, D. Pezzi, and M. Prato, “Explainable bilevel optimization: An application to the helsinki deblur challenge,” *Inverse Problems in Imaging*, 2023.
- [137] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, “Optuna: A next-generation hyperparameter optimization framework,” in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019.
- [138] M. Andrzej and R. Waldemar, “Principal components analysis (pca),” *Computers & Geosciences*, pp. 303–342, 1993.
- [139] A. Mishra, L. Li, M. Kong, H. Finkel, and B. Chapman, “Benchmarking and evaluating unified memory for openmp gpu offloading,” in *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC*, pp. 1–10, 2017.
- [140] R. Usha, P. Pandey, and N. Mangala, “A comprehensive comparison and analysis of openacc and openmp 4.5 for nvidia gpus,” in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–6, IEEE, 2020.
- [141] Z. Wang, Z. Wang, C. Liu, and Y. Hu, “Understanding and tackling the hidden memory latency for edge-based heterogeneous platform,” in *3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20)*, 2020.
- [142] J. Choi, H. You, C. Kim, H. Young Yeom, and Y. Kim, “Comparing unified, pinned, and host/device memory allocations for memory-intensive workloads on tegra soc,” *Concurrency and Computation: Practice and Experience*, vol. 33, no. 4, p. e6018, 2021.
- [143] Y. Gu, W. Wu, Y. Li, and L. Chen, “Uvmbench: A comprehensive benchmark suite for researching unified virtual memory in gpus,” *arXiv preprint arXiv:2007.09822*, 2020.
- [144] S. Chien, I. Peng, and S. Markidis, “Performance evaluation of advanced features in cuda unified memory,” in *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, pp. 50–57, IEEE, 2019.
- [145] J. Bakita and J. H. Anderson, “Hardware compute partitioning on nvidia gpus,” in *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 54–66, IEEE, 2023.