6-1-2021

# Resource management and application customization for hardware accelerated systems

Zois Gerasimos Tasoulas
*Southern Illinois University Carbondale*, zoistas@hotmail.com

# RESOURCE MANAGEMENT AND APPLICATION CUSTOMIZATION FOR HARDWARE ACCELERATED SYSTEMS

by

Zois Gerasimos Tasoulas

Diploma, National Technical University of Athens, 2016

A Dissertation
Submitted in Partial Fulfillment of the Requirements for the
Doctor of Philosophy Degree

Department of Electrical, Computer and Biomedical Engineering
in the Graduate School
Southern Illinois University Carbondale
May 2021

**DISSERTATION APPROVAL**

RESOURCE MANAGEMENT AND APPLICATION CUSTOMIZATION FOR HARDWARE ACCELERATED SYSTEMS

by

Zois Gerasimos Tasoulas

A Dissertation Submitted in Partial

Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

in the field of Electrical and Computer Engineering

Graduate School
Southern Illinois University Carbondale
February 17, 2021

## AN ABSTRACT OF THE DISSERTATION OF

Zois Gerasimos Tasoulas, for the Doctor of Philosophy degree in Electrical and Computer Engineering, presented on February 17, 2021, at Southern Illinois University Carbondale.

TITLE: RESOURCE MANAGEMENT AND APPLICATION CUSTOMIZATION FOR HARDWARE ACCELERATED SYSTEMS

MAJOR PROFESSOR: Dr. Iraklis Anagnostopoulos

Computational demands are continuously increasing, driven by the growing resource demands of applications. At the era of big-data, big-scale applications, and real-time applications, there is an enormous need for quick processing of big amounts of data. To meet these demands, computer systems have shifted towards multi-core solutions. Technology scaling has allowed the incorporation of even larger numbers of transistors and cores into chips. Nevertheless, area constrains, power consumption limitations, and thermal dissipation limit the ability to design and sustain ever increasing chips. To overpass these limitations, system designers have turned towards the usage of hardware accelerators. These accelerators can take the form of modules attached to each core of a multi-core system, forming a network on chip of cores with attached accelerators. Another option of hardware accelerators are Graphics Processing Units (GPUs). GPUs can be connected through a host-device model with a general purpose system, and are used to off-load parts of a workload to them. Additionally, accelerators can be functionality dedicated units. They can be part of a chip and the main processor can offload specific workloads to the hardware accelerator unit.

In this dissertation we present: (a) a microcoded synchronization mechanism for systems with hardware accelerators that provide distributed shared memory, (b) a Streaming Multiprocessor (SM) allocation policy for single application execution on

GPUs, (c) an SM allocation policy for concurrent applications that execute on GPUs, and (d) a framework to map neural network (NN) weights to approximate multiplier accuracy levels. The aforementioned mechanisms coexist in the resource management domain. Specifically, the methodologies introduce ways to boost system performance by using hardware accelerators. In tandem with improved performance, the methodologies explore and balance trade-offs that the use of hardware accelerators introduce.

## ACKNOWLEDGMENTS

At this point, I would like to thank the persons that contributed invaluably to my studies in the Southern Illinois University, and the completion of this effort.

To begin with, I would like to thank my academic advisor, Dr. Iraklis Anagnostopoulos. His guidance and support were invaluable for the completion of this research. He made sure to provide me with all the necessary tools in order to successfully research the area of performance improvement. He was present and more than willing to help when I got stuck or doubts arose towards the direction of research. His experienced insight contributed majorly in the completion of the presented methodologies, and was critical to push research forward at moments of uncertainty. I thank him sincerely for all the help, knowledge and opportunities he provided me with. I am also thankful for the interesting, enriching and professional collaboration we had, throughout these five years.

Additionally, I would like to thank my lab mates. Together we shared many moments and experiences. We learned a lot from each other, and developed our skills. Their company made my PhD experience more pleasant and enriching.

I would like to thank my family. My mother for her constant support and for always being there for me, overcoming her difficulties to support me. My sisters, for their love and support through the period of the PhD. I thank them deeply for caring about me. I want to thank also my godfather and his family. They are probably the main reason I chose to study in the United States. They have helped me enormously and gave me so much. I am not sure if I can thank them enough or even return the help I have received.

Finally, I thank all of my friends, for the fun moments we shared. They made the PhD experience much more pleasant.

# DEDICATION

My dissertation is dedicated to my family, for supporting and inspiring me through all these years.

## TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# CHAPTER 1
# INTRODUCTION

## 1.1 INTRODUCTION

Modern computing and embedded systems are moving away from super scalar and multi-core architectures and follow the many-core paradigm or heterogeneous system approaches, in order to provide high throughput and meet application demands.

The many-core paradigm is characterized by the constant increase in the number of integrated processors, e.g., 48 [1] and 5772 [2] cores. To achieve high communication performance for the cores, many-core systems adopt schemes such as Network-on-Chip (NoC) connectivity to reduce communication cost and allow users to harness the full potential of the underlying cores. At the scale of decades or even hundreds of cores, memory access can introduce an important bottleneck. A large number of cores trying to access simultaneously memory can create a noticeable slow-down during execution. To overpass this problem, system designers have introduced Distributed Shared Memory (DSM) for many-core systems. By saving data in multiple memory locations, cores can reduce the probability of trying to access the same location, thus improving performance. Unfortunately, DSM has limited applicability due to its cost, for on-chip solutions, and the performance overhead that data consistency mechanisms introduce.

Graphics Processing Units (GPUs) are one of the most popular choices to accelerate execution on heterogeneous systems. Initially, GPUs where designed to help with graphics processing and rendering. Nevertheless, since the initial GPU models, computer engineers understood that the underlying computational power of GPUs can be harnessed for a great variety of applications. Nowadays GPUs are used to accelerate and execute applications from a wide range of domains, e.g., image recognition, neural networks and crypto-mining. GPUs derive their remarkable computational power from the thousands of processing cores that they incorporate, for example 5120 CUDA cores in NVIDIA Tesla

$V100$ PCle [3] and 4096 cores for AMD Radeon RX Vega 64 [4]. Although GPUs can yield high performance, their distinct architecture and programming model requires special programming practices and different approaches towards resource handling, compared to conventional CPUs.

GPUs adopt the Single Instruction Multiple Thread (SIMT) paradigm to achieve high parallelism. From a hardware perspective, GPUs incorporate multiple Streaming Multiprocessors (SMs) which in their turn consist of Streaming Processors (SPs). SPs are the operational units that execute application threads. From a software perspective, applications that intend to utilize a GPU are organized in computational kernels. These kernels are offloaded to the GPU. The threads of the kernels are organized in thread blocks that are sent to SMs. Within these blocks, threads are organized in groups, usually called warps. Due to the SIMT model, the SPs of an SM can only execute threads of a specific warp at each moment, spatial sharing of SPs among different warps is not possible.

An additional case of the heterogeneous paradigm is a host-target scheme. Specifically, a device or System on Chip (SoC), representing the host, can contain functionality specific modules, representing the target, that are used to offload specific workload and accelerate execution. One example of target modules are Neural or Tensor Processing Units (NPUs or TPUs). NPUs are hardware accelerators dedicated to improve Artificial Intelligence (AI) tasks, for example NN workloads, machine learning and machine vision. A subcategory of NNs are Convolutional Neural Networks (CNNs). The main operation during inference on CNNs is multiply-accumulate (MAC). To accelerate inference calculation, NPUs comprise many MAC units that can very efficiently execute the MAC operation.

The complexity of modern deep NNs requires NPU accelerators to integrate thousands of MAC units in order to provide significant performance improvement. For example, the embedded-oriented Samsung's NPU uses 1K MACs [5] and Google's Edge

TPU comprises 4K MACs [6].

## 1.2 MOTIVATION

In this section, we provide the motivation that led us engage with the aforementioned domains, and research techniques that improve system performance while considering trade-offs.

### 1.2.1 Synchronization for multi-core systems

As stated earlier, memory access can introduce significant delays in execution, for multi-core systems. In certain applications, multiple cores need to access the same data. For example, in an application that many cores read and write data, from and to a common data structure, certain guarantees need to be provided, either at the operating system (OS) level or the application level. If the access to the shared data structure is not protected and orchestrated with guarantees, consistency issues can arise. Even though multiple reads of shared data can be served by modern systems without errors, when we encounter scenarios of multiple writes or combinations of reads and writes, discrepancies can appear. The sequence at which the read and write operations are served, affect immediately the condition and consistency of the data structure.

Synchronization mechanisms have been developed in order to avoid the aforementioned problems and provide a consistent image of the saved data. Primitive synchronization techniques such as coarse-grain locks, provide the necessary guarantees but limit significantly performance and scalability. One of the simplest synchronization mechanisms is a lock, requiring every core that needs access to the structure to acquire it before implementing transactions with the data structure. Although a lock can be simple to implement and is supported by many modern computer systems, it limits performance. When multiple cores compete to acquire the lock, processing time is lost during the competition. Additionally, while a core holds the lock, the rest of the cores have to

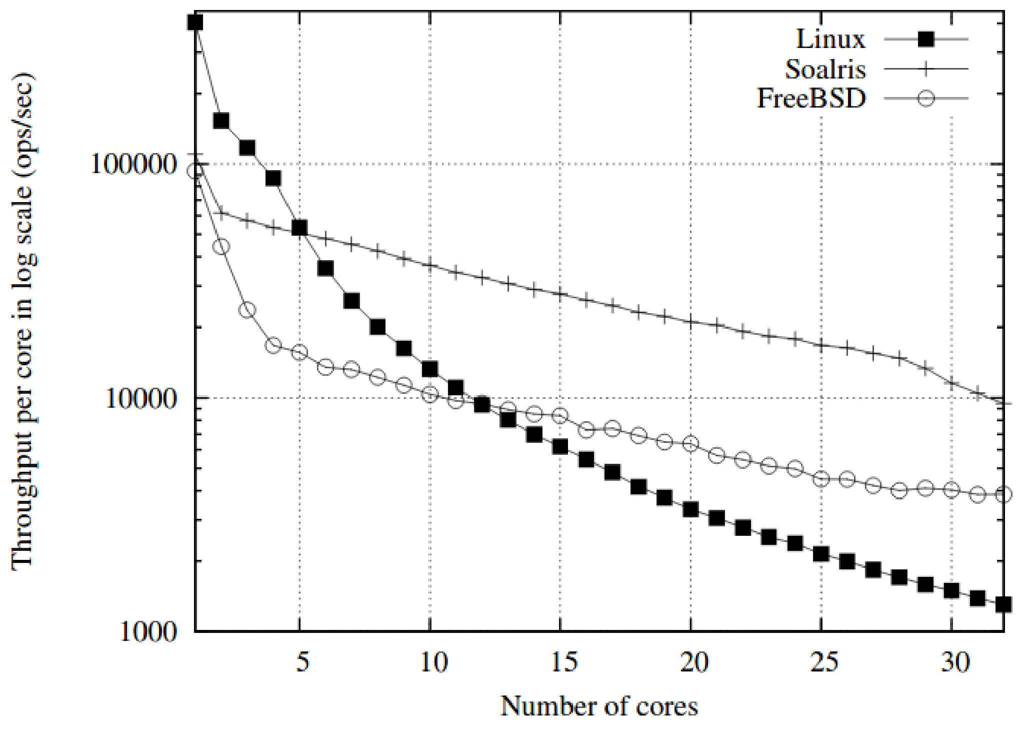


Figure 1.1: Throughput per core for scaling number of cores [7]

remain idle until the lock is freed. This is translated to even more lost processing time. Adding to all that, a single key protecting a data structure is a significant hurdle when applications need to scale. In case an application needs to scale and more cores have to be used to process data, there is a point where performance can break down, due to a huge amount of latency per core in order to acquire the lock. To highlight the importance of efficient synchronization mechanisms we provide Figure 1.1. In this Figure we can observe that popular OSs experience a performance breakdown when the number of cores on a system increases, thus demonstrating scalability bottlenecks. This observation supports our claim that improved synchronization mechanisms are necessary to guarantee data integrity, combined with high performance. Finally, coarse-grain locks are vulnerable in other aspects too, for instance they introduce a single point of failure that can be a vulnerability for certain applications.

Additional synchronization mechanisms include memory coherency mechanisms. Coherency techniques guarantee that the system manages to maintain a coherent state of the data, among different memory locations. From the programmer's point of view, such mechanisms are very convenient because the underlying system is responsible for the data coherency. Usually the programmer does not need to give any instruction, the system orchestrates the necessary mechanism that will guarantee data coherency among multiple core memories. Unfortunately, even though memory coherency mechanisms are convenient for the programmers, they present certain limitations. These mechanisms do not scale well and have a performance impact. Considering computing systems with many cores, using memory coherency mechanisms would introduce a significant delay when data has to be kept consistent among all of the cores of the system.

As a consequence of the limitations that the initial synchronization techniques demonstrate, there is a demand for synchronization mechanism that can allow high performance while guaranteeing correct data. Improvements in computer architecture, such as DSM, allow researchers and system engineers to experiment with more elaborate synchronization schemes and achieve improved performance for many-core systems. **This is the reason hardware accelerators present a promising alternative, in order to overcome current limitations and offer higher throughput for multi- and many-core systems. The area of utilizing hardware accelerators to implement synchronization techniques has to be further researched because it can yield interesting and useful results.**

### 1.2.2 Streaming Multiprocessor allocation for single application execution

GPUs are excessively used to accelerate applications on various domains. Utilizing the resources of a GPU at their maximum potential is not a trivial task, due to the unique GPU architecture and differences in the GPU programming model, compared to general purpose CPU programming. Allocating all the available SMs to an application can result in suboptimal performance.

In Figure 1.2 we can see that certain applications, e.g. FFT and BLK, drop or

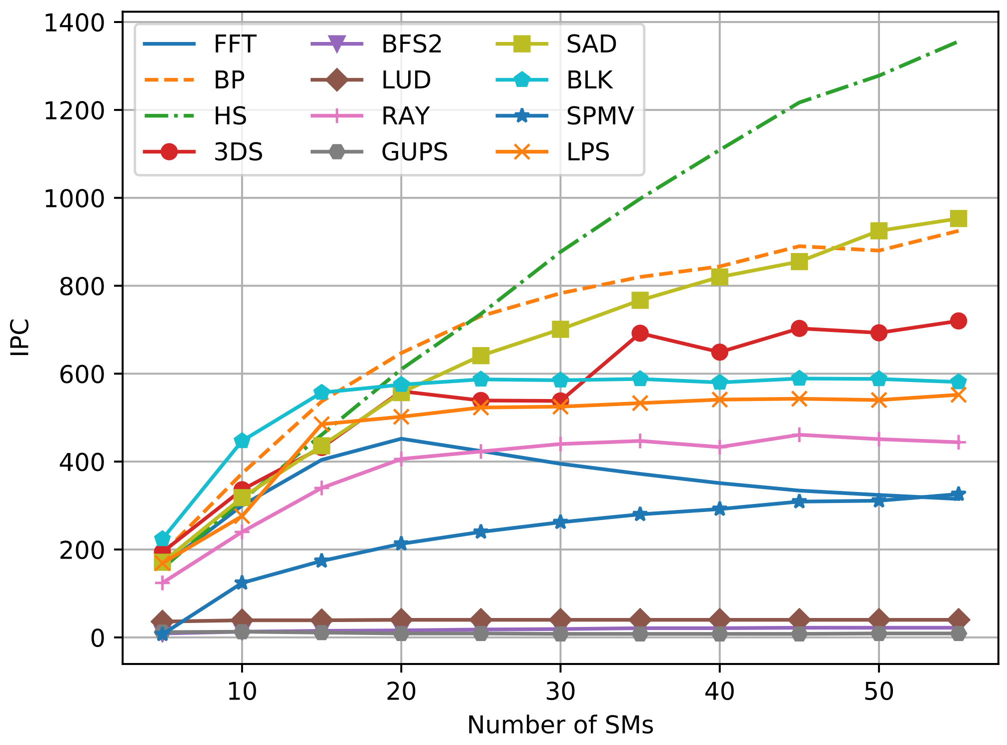


Figure 1.2: IPC per application [8]

maintain constant their performance when SMs beyond a threshold are offered to them. Providing more information about the content of Figure 1.2, we executed the Rodinia benchmarks [9] individually on a GPU, allocating each time a different number of SMs, from 5 to 55, with a step of 5 SMs. For each configuration we plot the Instructions Per Cycle (IPC) of each application.

The observations from Figure 1.2 demonstrate that there is a motive, in terms of performance, to limit the number of available SMs for certain applications. Aside of the performance gain, when SMs are not being used, they can be clock-gated. By clock-gating SMs, we can limit their active cycles, thus reducing their material degradation. Degradation in chips is caused by transistor degradation. Activity on SMs translates to switches of transistor state. These switches mean that electric current passes through the transistors. As a result of this current activity, there is a small but permanent degradation in the material of the transistors. Clock-gating SM results in divergence of

SM usage and activity. This can lead to varying degradation among the SMs. Material degradation over time leads to aging which in turn causes among other issues, slower frequency and reliability issues. Recent SM allocation policies do not consider the activity history of SMs while allocating them. Partially this can be because most allocation policies allocate all the available SMs for each application.

**Combining the analysis from the previous paragraphs, there is research space to be explored in terms of SM allocation policies. Allocating different number of SMs for each application can lead to improved overall performance. Additionally to improving performance, reducing the number of allocated SMs enables the option of reducing the activity divergence among SMs.** Nevertheless, the SM allocation mechanism should be aware of the activity condition of each SM, in order to be able balance activity among them. Imbalances in SM activity are unwanted in a GPU because they lead to significant differences on SMs over the time. These differences can reduce performance, cause premature system failures and reduce reliability. Concluding, there is a motivation to develop an SM allocation policy that will provide improved performance while balancing activity among SMs.

### 1.2.3 Streaming Multiprocessor allocation for concurrent applications execution

Until recently, two or more applications were not able to run simultaneously on a GPU. This means that computational resources can remain underutilized. As mentioned in Section 1.2.2, GPU resources can remain underutilized by certain applications because there are not enough tasks to take full advantage of all the SMs.

To elaborate, underutilization can occur because different applications have different computational needs. Some applications are computationally intensive, improving their performance when more SMs are made available to them. On the other hand, other applications can be memory bounded, which means that their performance is coupled with the available memory bandwidth. Even if more SMs are available to applications that belong to the latter type, their performance will not improve. As a consequence, there is an incentive to determine the computational needs of applications. GPU performance can be improved by resource sharing when a system needs to execute multiple applications that demonstrate different computational needs. Instead of executing sole applications on the GPU, SMs can be shared between multiple applications in order to harness as much computational power as possible.

Nevertheless, running different applications on the SMs causes different activity rates among the SMs. Activity on SMs translates to switches of transistor state. When SMs are allocated to multiple applications executing concurrently, their activity can vary significantly. This can lead to varying degradation among the SMs. The variance among SM degradation can be further exacerbated by the effects of process variation (PV), these are material differences caused during the manufacturing process.

**Combining the aforementioned facts, there is a motivation to craft an allocation policy for SMs on GPUs. Taking advantage of concurrent execution of applications on a GPU, system performance can be improved.** In addition to improving performance, the developed SM allocation policy should not overlook activity divergence among SMs. Together with improved performance, the allocation policy should aim at activity balancing among SMs, in order to avoid future inequalities of SM condition.

### 1.2.4 Weight oriented approximation for neural network inference acceleration

As stated previously, NPUs are composed of thousands of MAC units in order to achieve high performance. Nevertheless, the power consumption of thousands of MAC units can prove to be a constrain for embedded devices with limited power capacities. Unless power consumption can be withheld between certain limits, embedded devices cannot harness the maximum potential in performance improvement, offered by NPUs.

A way to achieve low power consumption combined with high performance during NN inferences is through leveraging approximate computations. NNs demonstrate inherent error resilience [10, 11] thus can leverage the benefits of approximate computing without significant impacts on the end result. On its core, the approximate computing paradigm is based on the use of approximate circuits to execute addition and multiplication operations. The approximate circuits trade result accuracy for reductions in execution time and power consumption. The aforementioned reductions are achieved by simpler circuitry, which is the source of result approximation as well. Given that CNNs comprise thousands of MAC operations, the use of approximate computations can contribute in significant power savings, as long as the overall result error is withheld is acceptable limits. Additional power gains can be achieved if the approximate MAC unit offers multiple approximation levels that can be altered during run-time. The existence of multiple approximation levels allows the system engineer to alter the level of approximation during inference. As a result, the introduced error in the result can be better controlled. For example, layers that are critical and have a greater impact on the inference result can be calculated with low level of approximation or exactly. Less significant layers can utilize higher approximation, thus yielding more power gains.

Existing analyses explore the impact of error in NNs [12], make use of fixed approximation multipliers [10, 11], or develop a layer-based approximation that tries to find the right approximation level separately for different convolution layers [13]. The aforementioned approaches can be time consuming due to required retraining of the NN or may not leverage the full potential of approximate MAC units, due to fixed approximation levels and lack for input-adaptive run-time support. A way to surpass the above issues can be weight oriented approximation. In other words, instead of choosing an approximation level for whole convolutional layers, we can explore a finer grain mapping. Specifically, within the same convolutional layer, we can assign different weights

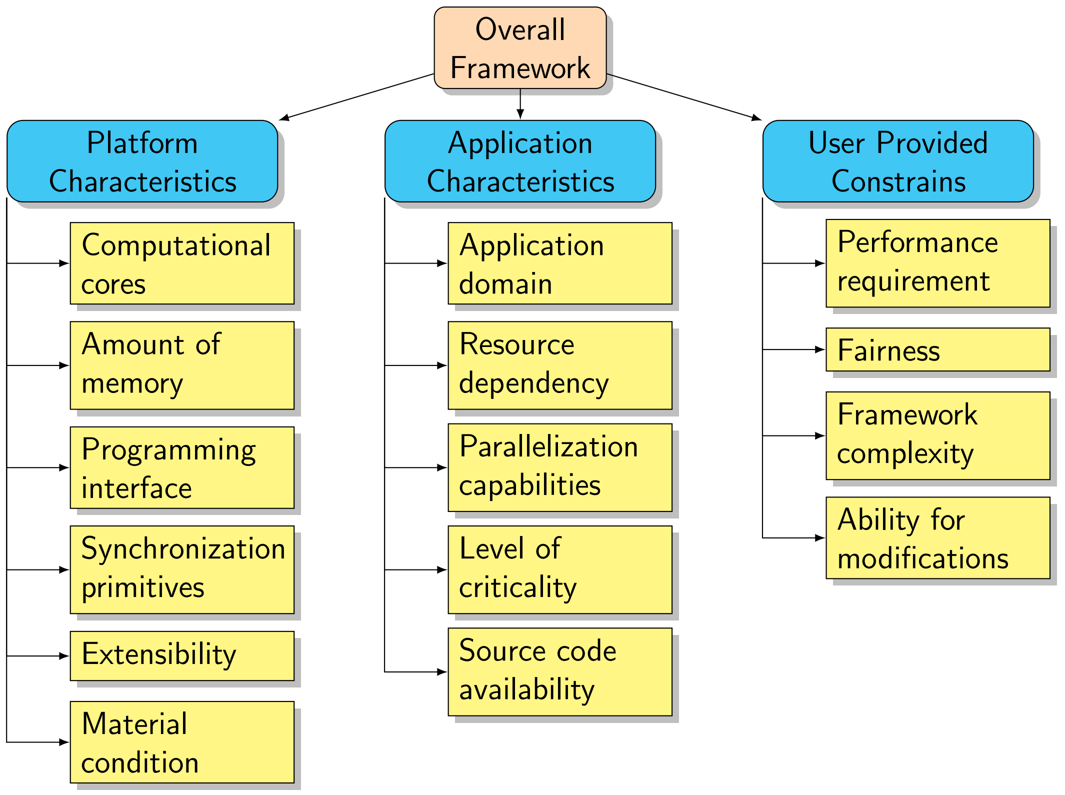


Figure 1.3: Framework organization overview

to various approximation levels, provided that the approximate MAC allows run-time approximation level configuration.

**If we follow the facts stated in this section, we deduce that NN inference performance can be improved by using hardware accelerators, i.e., NPUs. Specifically, for the embedded device domain, there is an incentive to explore resource management for NPUs in order to achieve power savings. The area of weight oriented approximation is promising power savings, while at the same time inference error can be held within acceptable limits.**

## 1.3 DISSERTATION OVERVIEW

The issues discussed in the previous section, Section 1.2, reside under the umbrella of run-time resource management for hardware-accelerated systems. Specifically, depending on the application domain, different hardware accelerators are suitable in order to achieve better performance or have energy savings. The variety of hardware accelerators and software applications, demands separate solutions that take into consideration how to best leverage the benefits that each accelerator offers, for example GPUs offer massive parallelisms whereas NPUs offer energy efficient calculations. Furthermore, each application domain presents separate challenges and each type of accelerator introduces different trade-offs. The methodologies presented in this dissertation take all these challenges and trade-offs into consideration and propose solutions to mitigate their effect. In Figure 1.3 we depict the generic directives that guide a framework design for hardware accelerated systems. It includes constrains and specifications regarding the baseline hardware platform, the application dependencies and requirements when executed on hardware accelerated systems, as well as user defined requirements. Albeit accelerators and applications belong in different domains and demonstrate structural differences, the sequence of steps and considerations when designing a framework are similar, for all the types of accelerators used in this work. Thus, the same process can be followed to determine how to best leverage the benefits of a hardware accelerator while mitigating to the best extend the trade-offs introduced by its usage.

The introduction chapter is completed at this point, the remaining chapters in the dissertation are organized as follows:

- Chapter 2 presents the novel contributions of this dissertation, as well as a concise description of the already available works in the areas we researched.
- Chapter 3 presents in detail a synchronization model for multi core systems that have hardware accelerators attached to each core. Each hardware accelerator contains distributed shared memory and enables microcode usage.
- Chapter 4 introduces a framework for SM allocation under the scenario of executing single applications. The framework balances optimal performance for applications executing on a GPU while it also reduces aging divergence among the SMs of the

GPU.

- Chapter 5 presents an SM allocation policy for the scenario of concurrently executing applications on GPUs. The policy decides on how to partition SM between two applications in order to minimize slowdown caused by resource sharing. Additionally, the policy takes into consideration aging divergence among the SMs and tries to minimize it.

- Chapter 6 introduces a time-efficient framework that can map CNN weights to approximation levels of a reconfigurable approximate multiplier. The framework achieves energy savings while it introduces a small accuracy loss on inferences.

- Finally, Chapter 7 is the conclusion of this dissertation. It summarizes the main contributions and provides future possible extensions for the presented methodologies.

# CHAPTER 2
# CONTRIBUTION

## 2.1 NOVELTIES

In this section we highlight the novel contributions presented in this dissertation. The four main contributed methodologies can be summarized into: i) a synchronization model for multi-core systems; ii) an SM allocation policy for single application execution on GPUs; iii) an SM allocation policy for concurrently executing applications on GPUs; and iv) a weight oriented mapping framework for approximate multipliers with multiple approximation levels.

The novel contributions for the synchronization model are:

- A microcoded synchronization model for concurrent data structures that achieves higher performance, fair access to the shared data structures, less idle cycles per core, and lower power consumption.
- The development of a client-server model that leverages DSM, and is based on message-passing.
- The use of a hardware dual-microcoded controller (DMC) to accelerate the processing of client requests to the server.
- The proposed synchronization model achieves up to 88× less idle cycles, serves requests at least 1.39× faster and consumes at leas 5× less power, compared to single lock and other state-of-the-art synchronization mechanisms.

The novel contributions for the SM allocation policy for single application execution on GPUs are:

- A methodology to determine a set of optimal and close-to-optimal configurations for the kernels of an application.

- A kernel-level allocation policy that makes the methodology more adaptive to the differences between kernels of an application.
- An algorithm for SM allocation that keeps track of the activity factor history of the SMs and re-adjusts the SMs of the GPU at run-time in order to achieve higher throughput and minimize activity imbalances.
- A decision mechanism to clock-gate SMs based on the characteristics of the executing kernel. SMs that do not contribute in higher performance of a kernel are clock-gated in order to reduce their activity. Thus, relative delay incurred by aging is decreased.
- The proposed allocation policy improves GPU throughput by at least 13.8% and reduces aging divergence among the SMs by up to 89.6%, compared to state-of-the-art aging aware scheduling policies.

The novel contributions for the SM allocation policy for concurrently executing applications on GPUs are:

- A methodology to choose how applications should be paired together before executing them on a GPU.
- A concrete algorithm to decide how to partition SMs between two co-executing applications, based on profiling data collected off-line.
- A process to decide which SMs to pick for each executing application, in order to mitigate aging imbalances among the SMs of a GPU.
- An overall methodology to improve performance and manage aging imbalances while executing concurrent applications on GPUs. The proposed methodology takes into consideration the effects of PV.

- The proposed allocation policy achieves up to 27% higher GPU throughput for applications executing concurrently, compared to other aging-aware scheduling policies. Additionally, it can achieve 5.9× lower aging deviation, compared to other state-of-the-art aging aware scheduling policies.

Finally, the novel contributions for the weight oriented mapping framework for approximate multipliers with multiple approximation levels are:

- a time efficient methodology that maps different approximation levels of a multiplier to NN weights per convolutional layer.
- a run-time framework that given a trained NN can map weights to approximation levels. The framework receives as an input an error threshold that needs to be satisfied. Within the allowed error margin, the framework achieves power gains while it incurs negligible time overhead.
- The proposed framework, combined with a reconfigurable approximate MAC unit, can achieve up to 20.2% energy gains while reducing accuracy by 0.5% to 2.0%, comparing to inferences calculated with exact circuits. The required overhead to apply the weight exploration of the proposed framework can reach up to 2 hours which is an acceptable trade-off, considering alternative methods.

The next section presents existing approaches that deal with the issues of leveraging performance improvement form hardware accelerators. We highlight the strengths of the existing approaches, as well as the parts that can be improved or can be addressed with the methodologies presented in this dissertation.

## 2.2 RELATED WORK

There is rich literature on resource management for hardware accelerated systems. In this section we present existing works that dealt with the topics of synchronization techniques for concurrent data structures, SM allocation and SM aging for single and multiple applications, as well as weight oriented acceleration for NN inferences on approximate hardware.

### 2.2.1 Synchronization for multi-core systems

There is high interest in the evaluation of performance and synchronization techniques for data structures. However, software is not yet able to scale at the same pace, [14]. As presented in [15, 7], non-scalable locks can severally degrade the performance of commercial software.

A detailed presentation of different aspects of synchronization techniques and a theoretical approach to multi-core synchronization scalability was made in [16]. The author presents many different ideas and approaches towards synchronization and analyzes why some techniques fail to scale after a certain number of cores. Many synchronization concepts unfortunately cannot be implemented due to hardware limitations. A thorough comparison of synchronization techniques takes place in [17]. The authors evaluate the performance of various synchronization models on different types of computer systems and state the type of system that is appropriate to achieve better performance for the various models.

The authors in [18], explored and evaluated the performance of message passing algorithms in embedded systems. Message passing can be an alternative to shared memory techniques and can provide solutions for certain architectures and configurations. As stated in [16], message passing solutions can also scale-up for a certain amount of cores. In [19, 20], the authors proved that by using microcode they can provide efficient solutions for dynamic memory management on distributed shared memory systems. However, the aforementioned solutions use single-locks for synchronization. The use of microcoded synchronization as we propose, could farther improve performance.

In [21], the authors evaluate and compare many synchronization techniques but they only use queue type data structures. Additionally, they do not provide any results for the fairness of the synchronization technique and their solution implies the existence of compare-and-swap (CAS) supporting hardware. Furthermore, in [22], the authors compare state of the art synchronization algorithms but they do not have a big variety of structures that they evaluate and the platform used for their experiments supports cache coherency. That can be a limiting factor for the applicability of many synchronization techniques. Also, they mainly focus on inter-thread synchronization. Finally, in [23], although the authors provide a lock free synchronization technique for multicore systems, their evaluation is based on a general purpose chips. Their solution implies that the hardware supports CAS operations and requires extra hardware components to be present in the processor or in the directory module.

### 2.2.2 Streaming Multiprocessor allocation for single application execution

Various research works have focused on improving GPU throughput. The authors in [24] pinpoint that the GPU resources are not fully utilized during run-time and the throughput of the system is less than the highest achievable throughput. In order to overcome this problem, they propose FineReg, a GPU architecture that improves the overall throughput by increasing the number of concurrent threads. Even though their method considers optimization at the kernel level, FineReg requires hardware modifications. Similarly, the methodologies presented in [25, 26] increase GPU throughput but require significant hardware modifications and they do not consider the effect of increased relative delay. Furthermore, the authors in [27] proposed a methodology to increase the warp size in order to address the memory divergence problem and reduce the effect of branching, while the authors in [28] leverage the performance cost of accessing memory. Although these methods improve performance, they do not consider aging imbalances among the resources of the GPU. In the long term, this can lead to performance degradation for the GPU.

As a way to better understand the behavior of GPU applications, the authors in [29] presented a methodology to classify applications depending on the usage of the memory controller and cache. This classification was used to execute concurrent GPU applications in order to minimize the slowdown. However, this methodology uses a coarse-grain approach by utilizing the overall behavior of the application without taking into consideration the differences between the kernels of the application. Also, the latter method works only for concurrent applications on GPUs. Moreover, the authors in [30] extract the optimal points of operation for several GPU applications and use this information in order to allocate the appropriate number of SMs. However, the latter methodology, similar to [29], does not utilize kernel-based information, leaving more room for improvements. A different approach to improve performance is followed in [31]. The authors present a compiler framework that improves application performance by by-passing caches in the GPU. This methodology does not consider the imbalances of execution that affect the aging of the computational components. Thus it fails to distribute aging among the cores of a GPU in a balanced way. In [32], authors improve GPU performance by proposing an alternative thread block scheduling and execution of concurrent kernels on the same SM. This methodology as well, does not explore the effects of kernel differences on resource pressure which can lead in aging imbalances.

Regarding modeling aging on electronic devices, [33, 34] present how it can affect performance. Numerous solutions have been proposed to reduce aging effects on GPU components and address the imbalance of aging rate among its components. However, the focus of existing solutions is mostly on balancing aging, while the optimization of the GPU throughput is not considered. Specifically, the authors in [35] present a register file design for GPUs that leverages data redundancy in order to mitigate NBTI effects. Additionally, in [36] the authors present a methodology that groups SPs and then clock-gates the group with the highest aging impact. In that way, the authors achieve lower aging-incurred relative delay without a significant overhead on execution. The authors extended their methodology in [37] by considering process variation as well. However, in both approaches the focus is given only on the reduction of aging, while optimizing GPU throughput is not considered. Also, the authors in [38] introduce a methodology that finds the number of SMs that provide the optimal performance. They choose to execute the applications with fewer SMs that the total available, in order to reduce the aging-incurred *threshold voltage ($V_{th}$)* shift on transistors of a GPU, while also reducing power consumption. Even though this approach increases the GPU throughput and balances aging, it works on the application level (coarse-grain) without taking into consideration kernel-based information (fine-grain).

Furthermore, the authors in [39] present a resource allocation policy that reduces the variation on SPs speed caused by process variation and aging due to activity. This work though, focuses on improvements at the SP level and does not consider the kernel structure of applications. In addition, the performance improvement is derived by the prolonged life and higher frequency of operations. The allocation policy does not consider the characteristics of each application in order to provide an allocation scheme of SPs that will boost performance. Moreover, the authors in [40] propose an aging-aware compilation method that combats delay-induced faults caused by the NBTI phenomenon on GPUs. This method though does not improve performance and in many cases adds an overhead in applications execution time. Overall, the aforementioned research works focus either on the SP level or register files of GPUs, or they do not serve the double goal of performance improvement coupled with aging divergence reduction.

### 2.2.3 Streaming Multiprocessor allocation for concurrent applications execution

Regarding GPU throughput optimization, a methodology to reduce contention on GPUs by assigning the appropriate memory bandwidth is presented in [41]. In [42], the authors explain how performance can be increased while executing simultaneous kernels and preempting them. However, the above mentioned research works do not consider aging effects and process variation that significantly affect the GPU throughput in the long run.

Modeling of aging on electronic devices and its effect on performance are presented in [34, 43]. The authors in [44] present a methodology to accurately calculate the average stress ratio of each transistor, considering the effects of different input configurations. Authors in [45] present a framework that calculates BTI variabilities caused by thermal variabilities among the components of a system. Furthermore, there are several proposed techniques to mitigate aging effects on electronic components. The authors in [46] propose heterogeneous components for mobile-platforms to counter aging effects. In [38], the authors present a profiling-based technique that mitigates aging on GPUs while improving power consumption. However, GPU performance is not optimized, and the authors focus only on single application execution. As presented in [47], aging depends on data patterns, and application schedulers can impact many-core systems greatly by determining the sequence of the applications [33]. If two processors experience the same stress at different time points, the aging rate is different in each case [33]. This makes the calculation of aging effects difficult on systems that employ a great number of processors and suffer from contention on shared resources.

Regarding PV, the authors in [48, 49] provide two models to calculate the effects of PV on multi-core systems, starting from the transistor level and moving up to the chip level. Additionally, in [50] the authors present a methodology to calculate PV correlations among different parts of a chip. The authors in [51] demonstrate the impacts of PV on frequency and throughput, whereas in [52] the authors present how PV affects performance for multi-core chips. Moreover, the authors in [53] present a workload partitioning algorithm that considers PV. The algorithm works on the SM level, but it does not mitigate aging imbalances.

### 2.2.4 Weight oriented approximation for neural network inference acceleration

In order to achieve accuracy reconfiguration during NN inference, the authors in [13] proposed a heterogeneous architecture built upon several static approximate multipliers [54]. Specifically, they apply at run-time a layer-wise approximation and they power-gate any approximate multipliers that are not used. However, this approach requires a heterogeneous architecture design, weight tuning, and it also has a high area overhead resulting in throughput loss due to the underutilized hardware. In [55], Simulated Annealing is used to produce approximate reconfigurable multipliers for NN inference by combining gate-level pruning [56] and wire-by-switch replacement [57]. Nevertheless, similar to [13], the approximate multipliers generated are optimized for the Mean Relative Error (MRE) metric and they apply only layer-wise approximation limiting the potential benefits. In [58] approximate reconfigurable circuits are generated using wire-by-switch replacement and by identifying closed logic island regions. The authors in [59] used reconfigurable bloom filters in order to support approximate layer-based pattern matching.

Previous methodologies have also tried to control the accuracy of the approximations at run-time by enabling reconfiguration [57, 60, 61, 62, 63, 64]. Particularly, the methods in [60, 61, 62] apply power gating to achieve reconfiguration. Considering that thousands of MACs are integrated in NN accelerators, such fine-grained power-gating approach is inefficient. In [65], the authors presented a NN accelerator which integrates approximate multipliers along with a compensation module in order to reduce energy consumption. Similarly, the authors in [11] analyzed the impact of error in NNs by utilizing approximate multipliers to different convolutional layers. However, the latter two approaches considered the LeNet NN, which is swallow comparing to current sate-of-art architectures, and the developed multipliers offer a single level of approximation thus not being flexible for deeper NNs. Moreover, the error compensation proposed in [65] requires the addition of an extra accumulation row in the MAC array, increasing thus its size as well as its computational latency. Additionally, the method in [11] requires retraining after the approximation has been performed in order to help NN adapt to the changes. The authors in [10] proposed approximate multipliers based on the concept of computation sharing to reduce energy consumption. Nevertheless, the introduced concept of Multiplier-less Artificial Neuron also requires network retraining in order to correct the accuracy loss due to the use of approximation. In [66], the authors study the impact of approximate multiplications on Capsule Networks (CapsNets) and compare their error resilience with convolutional NNs. However, they follow a layer-wise approach and the utilized multipliers support a single operational mode. Finally, as a way to accelerate the exploration process of approximate circuits for deep NNs, the authors in [67] proposed an emulation method optimized for GPUs. However, their solution considered approximate circuits with single approximation levels.

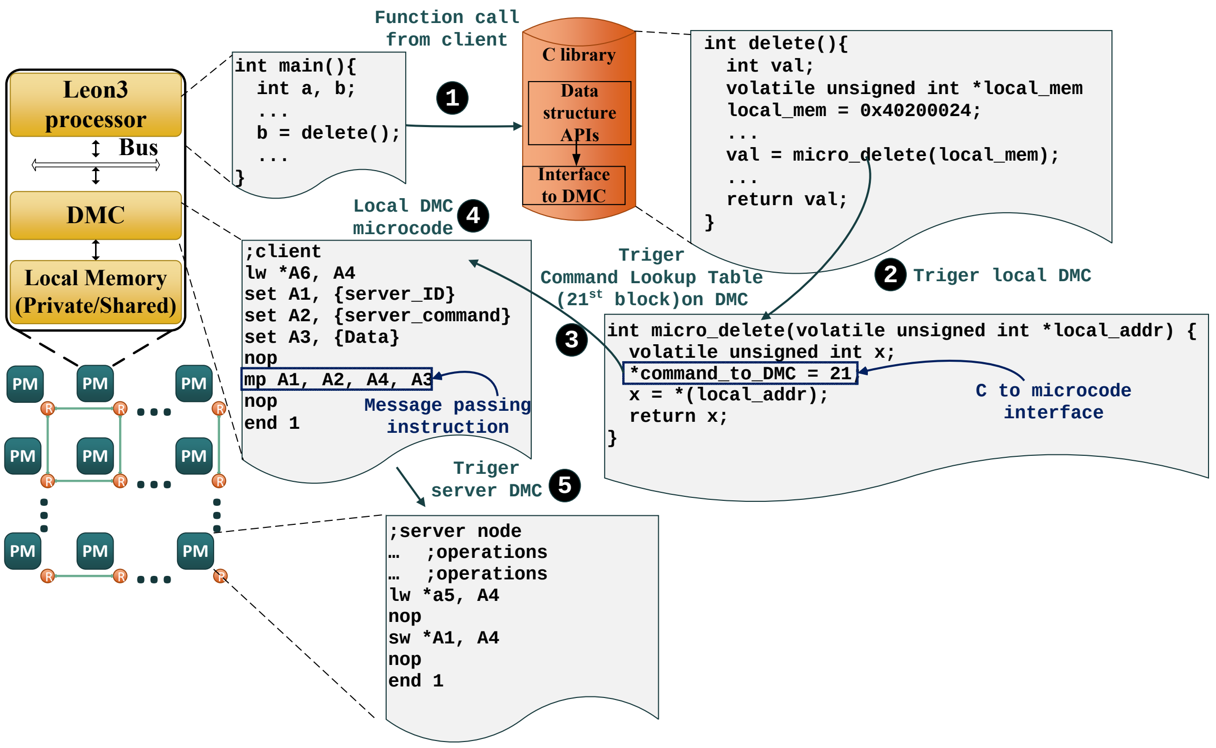
# CHAPTER 3
# MESSAGE-PASSING SYNCHRONIZATION FOR DISTRIBUTED SHARED MEMORY ARCHITECTURES

In this chapter we present in details the orchestration of the microcoded synchronization mechanism. We also present extensive experimental results that compare the developed mechanism with other synchronization mechanisms.

## 3.1 METHODOLOGY

The development of concurrent data structures provides various challenges, especially in systems with limited synchronization primitives. Assume that we have a DSM platform which is composed of Processor-Memory (PM) nodes interconnected via a packet-switched mesh network, as depicted in Figure 6.1. Additionally, each node employs a DMC, a programmable hardware accelerator [69] which allows the programmer to implement custom microcoded functions and trigger them by corresponding `C`-level APIs. Specifically, the DMC consists of two mini-processors. Mini-processor A which is responsible for inter-core tasks and memory accesses of the local core, and mini-processor B, responsible for accessing remote cores and serving shared memory requests by remote cores. The utilized platform follows the principle of industry-driven architectures [1, 70, 71] that adopt the DSM architecture with limited synchronization primitives, e.g., no Compare-And-Swap (CAS) support.

The proposed synchronization model is based on the idea that a single core plays the role of the "server" and it is the only one that accesses the data structure directly. The rest of the cores that need to access the concurrent data structure are "clients". Instead of accessing the data structure, they send requests to the server and wait for its response, if necessary. The server, as soon as it receives a request, performs the operation on behalf of the client and then sends a response to the client from which the request came from. In



```
int main(){
  int a, b;
  ...
  b = delete();
  ...
}
```

```
int delete(){
  int val;
  volatile unsigned int *local_mem
  local_mem = 0x40200024;
  ...
  val = micro_delete(local_mem);
  ...
  return val;
}
```

```
int micro_delete(volatile unsigned int *local_addr) {
  volatile unsigned int x;
  *command_to_DMC = 21
  x = *(local_addr);
  return x;
}
```

```
;client
lw *A6, A4
set A1, {server_ID}
set A2, {server_command}
set A3, {Data}
nop
mp A1, A2, A4, A3
nop
end 1
```

```
;server node
…   ;operations
…   ;operations
lw *a5, A4
nop
sw *A1, A4
nop
end 1
```

Figure 3.1: DSM platform and proposed synchronization [68]

the proposed method, the data structure is initially stored in the local private memory of the server, in order to save time, as there is no need to translate memory addresses. If the server needs more space, then it utilizes the shared memory of the system (starting from its local shared) defined during the initialization of the system.

The steps of the proposed synchronization model are displayed in Figure 6.1. The application developer calls in the `C` level the appropriate function to either add (`insert()/push()`) or remove (`delete()/pop()`) an element from the structure (❶). This call triggers our `C` library that contains the functions and the appropriate Processor-to-DMC interfaces for the used data structures. This function in its body contains a call to the microcoded function. At that point (❷), the microprocessor, attached to the core requesting a transaction (client), is notified and gets the control of the transaction. It is important to mention here that the microcoded functions are stored as command blocks in the control store of the accelerator and they are dynamically loaded by using a specific load command and the id of that block (❸). When the microcoded block has been loaded (❹), the mini-processor A fetches the microinstructions. Then, through the mini-processor B of the client, the communication with the server is initiated. Communication is based on message-passing, which means that the client sends a message to the server with its request and any additional required arguments. Specifically, message-passing communication was implemented with the usage of the `mp reg1, reg2, reg3, reg4` command. This message passing command allows to send a message directly from the mini-processor of a client core to the mini-processor of the server. The arguments of the command in order of presence are:

i) the destination node (server) of the message;

ii) the number of the microcoded block, stored in the server's control store, to be executed;

iii) the address where the returned data is expected; and

iv) the data of the client.

At that point (❺), the DMC of the server is triggered and starts executing the corresponding microcoded function (insert or delete). It is important to mention here that all the actions of the server are performed by the hardware accelerator, at the microcode level, and not by the processor (`C` level). Particularly, according to the received message, the DMC of the server triggers the appropriate command block in the control store and performs the requested operation. Finally, when the client receives the extracted data, in case of `delete()/pop()`, it finalizes the execution of the microcode function and the control returns to the main processor.

Apparently, the server serializes all operations. However, the presented model has a number of advantages that compensate for the decreased parallelism that it provides. As depicted in Figure 6.1, the concurrent data structure is initially allocated in the local private memory of the server, making the access to the data structure faster. Therefore, the number of memory accesses in remote memories is limited for both the server and the clients. Apart from the memory allocation issues, the primary reasons for performance improvement of the proposed synchronization method are:

i) In the case of an `insert()/push()` operation, when the client sends its request to the server, there is no need to wait for a response. Thus, in contrast with the typical lock-based implementations, an insert operation is getting blocked by an insert from another client, only if the server's mini-processor buffer is full and the message passing command blocks.

ii) The reduction in the instruction overhead (the synchronization details are hidden from the high `C`-level).

iii) The exploitation of the DMC for performing memory operations, thus alleviating the main processor's workload.

The proposed method can be applied to data structures with low level of parallelism, e.g., queues, stacks and heaps, which are widely used and found in applications and operating systems, but it cannot be straight-forward applied effectively [17] to data structures that allow multiple-write and/or multiple-read operations at the same time.

## 3.2 EVALUATION

The hardware platform used to implement the synchronization models is described in [20, 69]. Each node consists of a LEON3 processor, a hardware accelerator DMC and memory, shared between the nodes. The nodes are interconnected by Nostrum [72], a packet-switched mesh network, and see a continuous logical address space for the shared memory. To access the shared memory, nodes perform an address translation, accessing a lookup table to obtain the physical address and the number of the node holding this part of the memory.

To evaluate the proposed synchronization model, we compare it with four synchronization models that focus on pure or distributed shared memory systems and work at the `C`-level

i) a coarse grain model of a single-lock;

ii) a client-server model, called delegation model presented in [16];

iii) a clustered client-server model, inspired by the idea of flat-combining presented in [73]; and

iv) a modified DSM-sync model, presented in [17], with two h-factor values, 1 and 10, where each core acts as a server, in a round robin way, for $h$ requests.

Each of these methods was implemented on the same hardware platform and to be fair, the DMC was used to accelerate memory operations in all cases. Additionally, we chose node $(0, 0)$ (the first node in a mesh topology) as the server for our implementation and the one initializing the structure.

Table 3.1 presents the actions for each synchronization model. $T_{LEON3}$ is the command execution time on LEON3 processor, including cache lookup time or time spent at the bus until reaching the microprocessor, while $T_{v2p}$ is the time for virtual-to-physical address translation. $T_{lsm}$ stands for the time to access local shared memory and $T_{rsm}$ is the time to access remote shared memory. $T_{rem}$ is the time to launch a remote read request, $T_{poll}$ is the polling time for the lock and $n_l$ are the times that a core polls for it. $T_{lpm}$ is the time to access local private memory. $T_{com} = T_{csd} + T_{cds}$ is the communication latency where $T_{csd}$ is the latency from source to destination and $T_{cds}$ is the latency from destination to source. Parameter $\beta = 0$ means the structure is located in local shared memory, while $\beta = 1$ corresponds to remote shared memory and $\alpha = 1$ for memory read and 0 for a memory write. Last, $\gamma = 0$ means that the structure is in, local or remote, shared memory whereas for $\gamma = 1$ it is located in local private memory.

Table 3.1: Total time per request execution for all synchronization models

| Synchron. model | Total time |
|---|---|
| Single lock model | $T_{total} = \underbrace{T_{LEON3} + T_{v2p} + T_{rem} + T_{poll} * n_l + T_{com}}_{\text{acquiring lock}}$ $+ \underbrace{T_{LEON3} + T_{v2p} + \beta * T_{rsm} + (1-\beta) * T_{lsm}}_{\text{inserting/extracting element}}$ |
| Client-server model [16] | $T_{total} = \underbrace{T_{LEON3} + T_{v2p} + T_{rsm}}_{\text{checking if a request exists}} + \underbrace{T_{LEON3} + T_{v2p} + T_{rsm}}_{\text{reading/writing requested element}}$ $+ \underbrace{T_{LEON3} + \gamma * T_{lpm} + (1-\gamma) * (\beta * T_{rsm} + (1-\beta) * T_{lsm})}_{\text{accessing the data structure}}$ |
| Clustered client-server model [73] | $T_{total} = \underbrace{T_{LEON3} + T_{v2p} + T_{rsm}}_{\text{checking if a request exists}}$ $+ \underbrace{T_{LEON3} + T_{v2p} + T_{rem} + T_{poll} * n_l + T_{com}}_{\text{acquiring lock}}$ $+ \underbrace{T_{LEON3} + T_{v2p} + T_{rsm}}_{\text{reading/writing requested element}}$ $+ \underbrace{T_{LEON3} + T_{v2p} + \beta * T_{rsm} + (1-\beta) * T_{lsm}}_{\text{accessing the data structure}}$ |
| DSM-sync model [17] | $T_{total} = \underbrace{T_{LEON3} + T_{v2p} + T_{rsm}}_{\text{checking if a request exists}} + \underbrace{T_{LEON3} + T_{v2p} + T_{rsm}}_{\text{reading/writing requested element}}$ $+ \underbrace{T_{LEON3} + T_{v2p} + \beta * T_{rsm} + (1-\beta) * T_{lsm}}_{\text{accessing the data structure}}$ |
| Proposed model | $T_{total} = \underbrace{T_{LEON3}}_{\text{initiating the microcoded model}} + \underbrace{T_{v2p} + T_{rem}}_{\text{accessing remote core}}$ $+ \underbrace{\gamma * T_{lpm} + (1-\gamma) * (\beta * T_{rsm} + (1-\beta) * T_{lsm}) + T_{csd} + a * T_{cds}}_{\text{accessing data structure}}$ |

To evaluate and compare the proposed microcoded synchronization model, we utilized the data structures of stack, queue, deque [74], and binary max heap. The metrics we used are

i) the total execution time;

ii) progress, defined as the average number of cycles per request;

iii) core utilization, defined as the number of idle cycles of a core while waiting for its request to be completed; and

iv) power gain, defined as the power consumption gain achieved over the single-lock.

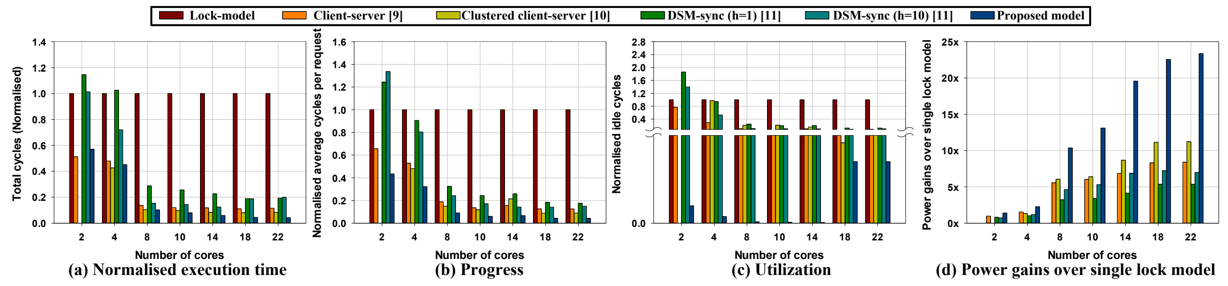


Figure 3.2: Stack [68]

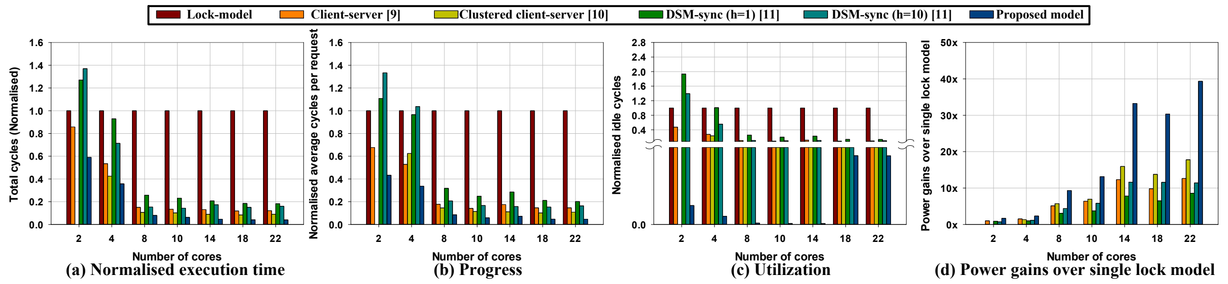


Figure 3.3: Queue [68]

As input, we utilized the benchmarks presented in [17], which consist of sequences of pairs of insertion and extraction operations.

Figures 3.2(a), 3.3(a), 3.4(a) and 3.5(a) depict total execution time. Beyond 4 cores, *the lock model experiences a breakdown in performance due to lock congestion* validating previous approaches [16]. We observe that the clustered client-server model performs better than the client-server model and many times it outperforms DSM-sync implementations. The clustered client-server model combines principles from the client-server and lock model and its efficiency is explained by the fact that clients are

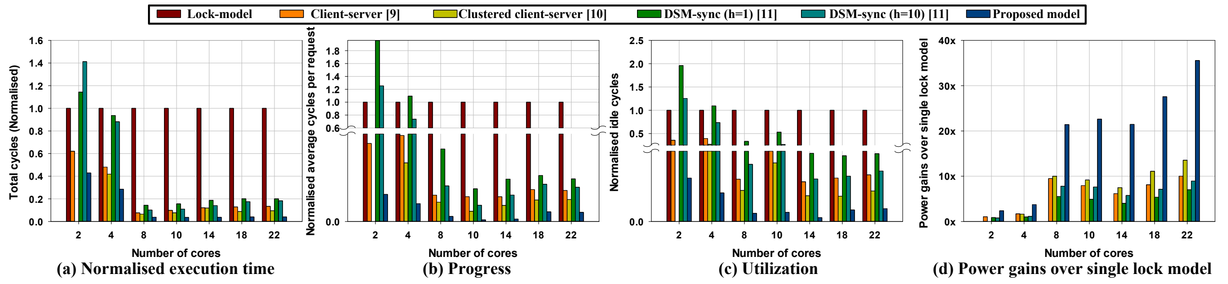


Figure 3.4: Deque [68]

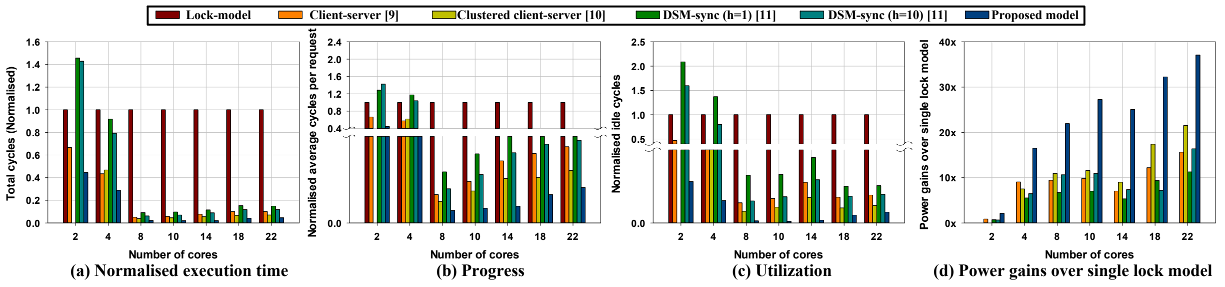


Figure 3.5: Binary max heap [68]

grouped under two servers and only these compete for the lock, leading to low congestion. The DSM-sync models have lower performance which is a result of server transitions, leading to wasted cycles. The microcoded model performs better than the rest of the synchronization techniques, for the queue with 14 cores, performs 3.7× faster than the DSM-sync model with h-factor = 10, and for 22 cores 2.2× faster than the clustered client-server model. This performance is achieved by utilizing the hardware accelerator for longer periods, bypassing main cores and using message passing to achieve communication between remote nodes.

Regarding progress, the microcoded model achieves gains compared to the other models. In Figures 3.2(b), 3.3(b), 3.4(b), and 3.5(b) we measured the average number of cycles for each request. The baseline is the number of cycles for the single lock model, which is used in many conventional systems. The proposed method performs a request 1.39× faster than the DSM-sync model with h-factor = 10 for the stack and demonstrates even better results compared to the lock model.

The core utilization is presented in Figures 3.2(c), 3.3(c), 3.4(c), and 3.5(c), where the baseline is the number of cycles for the single lock implementation. The proposed microcoded model uses the message passing command, which saves cycles by bypassing the upper execution level and allows a direct communication with the node hosting the data structure. Specifically, for the deque structure and 14 cores, the microcoded model has 88× less idle cycles compared with the lock model, 9.8× compared with DSM-sync model and 5.6× less idle cycles than the clustered client-server model.

Figures 3.2(d), 3.3(d), 3.4(d), and 3.5(d) present the power gains over the single-lock model. The proposed model demonstrates greater power gains due to the significant performance improvement and due to the fact that it utilizes the hardware accelerator for longer periods, as a result, processors remain idle. Specifically, the proposed model achieves an average gain of 5× for stack, 5× for queue, 8× for deque and 10× for binary max heap.

Overall, the proposed method achieves better results for the list-type structures and the tree-type structure, and provides promising performance, fair progress and greater power gain. According to [17] developing fully non-blocking techniques for tree-type structures is a cumbersome task and requires support of advanced synchronization primitives. The proposed method offers an efficient alternative for these structures, does not require advanced synchronization support and imposes an area overhead of up to $351k$ NAND gates per node [69].

## 3.3 SUMMARIZING

In this chapter we presented an efficient, hardware-accelerated, scalable synchronization model for distributed shared memory systems. Specifically, the presented synchronization mechanism is a contribution towards the implementation of concurrent data structures in architectures that provide limited synchronization primitives support, but have DSM available. Experimental results show that the proposed message-passing based client-server model provides increased performance, better throughput, better core utilization and greater power gains even in cases of high contention.

# CHAPTER 4
# KERNEL-BASED RESOURCE ALLOCATION FOR IMPROVING GPU THROUGHPUT WHILE MINIMIZING THE ACTIVITY DIVERGENCE OF SMS

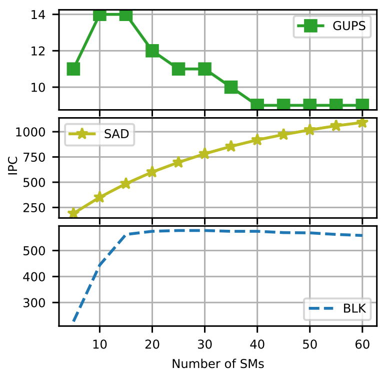
In this chapter we present the developed methodology for SM allocation considering execution of single applications on a GPU. The methodology aims at improving overall system throughput while balancing activity among SMs. Balanced activity will lead to equal aging among the SMs, which is a desired property for systems. The methodology takes decisions at the kernel level of each application, thus providing a fine-grain approach that yields better results than other, state-of-the-art methodologies. In this chapter, we present as well, extensive experiments that evaluate the efficiency of the developed methodology.

## 4.1 PRELIMINARIES

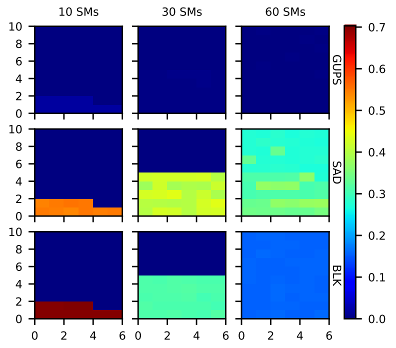
### 4.1.1 Motivation

In this section, we present the motivation that led us investigate methods to increase GPU throughput while minimizing the divergence of aging among the SMs of the GPU.

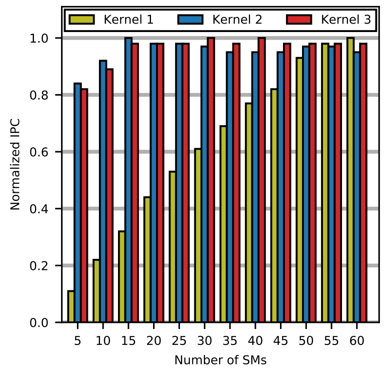
Observation 1: *Allocating all the SMs of a GPU to an application is not always beneficial in terms of GPU throughput.* The default scheduler of many commercial GPUs allocates for each application all the available SMs and distributes the tasks (thread blocks) among them. In Figure 4.1a we present the IPC for three benchmarks of the Rodinia suite [9], GUPS, SAD and BLK. We executed the benchmarks giving them each time a different number of available SMs (60 SMs was the total number of SMs on the GPU). The observation that we make is that the three benchmarks have different behavior as the number of available SMs changes. Specifically, GUPS has optimal IPC with 10 and 15 SMs but its IPC drops after that point. On the contrary, SAD



(a) IPC of three benchmarks for various numbers of available SMs



(b) Activity factor of each SM for executions with different numbers of available SMs



(c) Normalized IPC for the kernels of SAD benchmark

Figure 4.1: Motivational observations [75]

continuously increases its IPC as more SMs are available. Finally, BLK increases its performance to reach the optimal point at 25 SMs and 30 SMs. After that point there is a small drop at its IPC as the number of available cores increases. This can be explained due to the different nature of the benchmarks. Some benchmarks execute more computational instructions, thus their performance increases when more SMs are available. Other benchmarks depend significantly on loading/storing data from/to memory, thus allocating more SMs to them does not improve performance. As a result, certain benchmarks can improve their performance when fewer SMs are allocated to them.

*Observation 2: Reducing the SMs per application may increase GPU throughput but creates imbalance in the activity of the SMs.* In Figure 4.1b, for the same benchmarks that we plotted their IPC, we plot their *activity factor*. As activity factor we define the fraction of active cycles over total cycles for an SM. We plot the activity factor of each SM after executing each of the three benchmarks with 10, 30 and 60 SMs. We generally observe that the more SMs a benchmark has available, the lower the average activity factor of SMs is. Also the activity is distributed in a balanced way among the SMs. As the number of allocated SMs is reduced, certain areas of the GPU have higher activity where other areas have very low activity. The comment that we make though is that for certain benchmarks, for example GUPS, the change of activity among SMs is negligible for the different execution scenarios. Additionally for BLK, the increase in average activity among the SMs from 60 to 30 SMs is not significant. In this work, we quantify the aging effects as relative delay incurred to the SMs of the GPU, due to the degradation of the materials (Section 4.1.2). Increased relative delay due to aging causes performance degradation [36, 37, 38] as well as it severely affects the lifetime of a system [33, 76]. Additionally, imbalanced activity among the SMs of the GPU creates highly diverse aging rate. Combining the observations for IPC and activity behavior, we reach the following conclusions:

i) There is a double incentive to limit the available SMs for certain benchmarks. Their IPC can increase and we also get the opportunity to leave some SMs unused, thus reducing their activity while the average activity of the GPU does not increase.

ii) If we choose to reduce the available SMs for an application, we need to find a mechanism for allocating SMs in order to distribute the activity equally among them. If an application utilizes fewer SMs than all the available, activity divergence will appear among the SMs of the GPU causing imbalanced aging in the long term.

Observation 3: *Kernels of an application can have diverse behavior, further affecting overall GPU throughput and increasing the activity imbalance of the SMs of the GPU.* In Figure 4.1c we plot the IPC for each of the three kernels of SAD. The IPC of each kernel is normalized to the value of the optimal IPC for the same kernel. We observe that the performance of each kernel can also vary, within the same benchmark. For SAD, kernel 1 continuously increases its IPC as more SMs are being available, reaching its optimal value for 60 SMs. On the contrary, kernel 2 has optimal IPC for 15 SMs and kernel 3 has optimal IPC for 30 and 40 SMs. We conclude that choosing the same number of operating SMs for all the kernels of an application can actually hinder performance and constrain the ability to evenly distribute activity among SMs.

### 4.1.2 Aging model

Current that passes through transistors and their switching activity stress the transistor material. Two phenomena that cause aging-induced wear on transistors are Negative-Bias Temperature Instability (NBTI) [77] and Hot-Carrier Injection (HCI) [78]. NBTI affects mostly P-type Metal-Oxide-Semiconductor (PMOS) transistors and results in permanent threshold voltage ($V_{th}$) shift, while HCI affects mostly N-type Metal-Oxide-Semiconductor (NMOS) transistors.

To estimate the results of aging on the SMs of a GPU, we utilized the model presented in [34]. The aging effect is translated as relative delay incurred to a system, due to the degradation of the material of its computational blocks. We assumed that all the transistors of a SM have the same dimensions and are affected seamlessly by aging. This assumption is in line with other similar works such as [36, 37, 8]. The assumption is vital in order to simplify the underlying GPU model that we use to evaluate the developed methodology.

Equation 4.1 is used to calculate the relative delay of a block, caused by the threshold voltage ($V_{th}$) shift:

$$\Delta^{rel} d_{T_B}(t) = \Big(1 - \frac{\Delta V_{th}(t)}{V_{dd} - V_{th}(t_0)}\Big)^r - 1 \tag{4.1}$$

$V_{dd}$ stands for the supply voltage, $V_{th}(t_0)$ is the threshold voltage at the initial state, $\Delta V_{th}(t)$ is the threshold voltage shift caused by aging at time $t$ and $r$ is a technology dependent parameter. The average threshold voltage shift caused by the NBTI phenomenon is calculated by Equation 4.2:

$$\Delta^{avg} V_{th}(t) \leq \int_0^1 A_N u(V_{dd}) \frac{(v(T_B) \cdot \delta_B \cdot \delta_e \cdot t_m)^n}{w(\delta_B \cdot \delta_e, T_B, t)^{2n}} \mathrm{d}\delta_e \tag{4.2}$$

where $\delta_B$ is the duty cycle of the block, $\delta_e$ is the effective duty cycle, $T_B$ is the block temperature and $t_m$ is the period between two measurements. $A_N$ is a technology dependent parameter. As demonstrated in Equation 4.3, as duty cycle of a block we consider

$$\delta_B = \frac{t_{stress,B}}{t_{total}} = \frac{cycles_{stress,B}}{cycles_{total}} \tag{4.3}$$

We consider the effective duty cycle, $\delta_e$, to be uniformly distributed between 0 and 1 among the different transistors of a block. The functions $u$, $v$ and $w$ are shown in Equations 4.4, 4.5 and 4.6 respectively:

$$u(V_{dd}) = (V_{dd} - V_{th}) \cdot e^{(\frac{V_{dd}-V_{th}}{E_0})} \tag{4.4}$$

$$v(T) = \xi_4 \cdot e^{(\frac{-E_a}{kT})} \tag{4.5}$$

$$w(\delta, T, t) = 1 - \Big(1 - \frac{\xi_1 + \sqrt{\xi_3 \cdot v(T) \cdot (1 - \delta(t)) \cdot t_m}}{\xi_2 + \sqrt{v(T) \cdot t}}\Big)^{\frac{1}{2n}} \tag{4.6}$$

where $E_a$ is the activation energy, $k$ is the Boltzmann constant, $T$ is the temperature, $\delta(t)$ is the duty cycle at a specific moment $t$, $E_0$ and $\xi_i$ are technology dependent constants.

Regarding the effect of the HCI phenomenon, Equation 4.7 calculates the threshold voltage shift:

$$\Delta V_{th}(t) = A_H \cdot \sqrt{\alpha_{avg,B}} \cdot u(V_{dd}) \cdot v(T_B) \cdot \sqrt{\alpha_B \cdot f \cdot t} \tag{4.7}$$

$f$ is the frequency of a block, $A_H$ is a technology dependent variable, $a_B$ is the activity factor of a block and $a_{avg,B}$ represents the average activity factor of a block. Equation 4.8 expresses the relation that calculates the *activity factor* of a block $B$.

$$\alpha_B = \frac{t_{active,B}}{t_{total}} = \frac{cycles_{active,B}}{cycles_{total}} \tag{4.8}$$

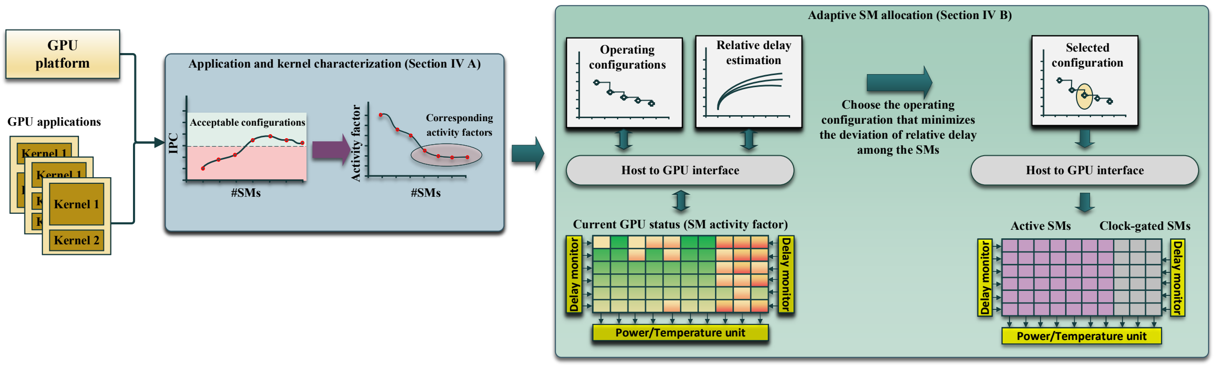

GPU platform
GPU applications
Kernel 1
Kernel 1
Kernel 1
Kernel 2
Application and kernel characterization (Section IV A)
IPC
Acceptable configurations
#SMs
Activity factor
Corresponding activity factors
#SMs
Adaptive SM allocation (Section IV B)
Operating configurations
Relative delay estimation
Choose the operating configuration that minimizes the deviation of relative delay among the SMs
Selected configuration
Host to GPU interface
Host to GPU interface
Current GPU status (SM activity factor)
Active SMs
Clock-gated SMs
Delay monitor
Delay monitor
Delay monitor
Delay monitor
Power/Temperature unit
Power/Temperature unit

Figure 4.2: An overview of the developed methodology [75]

## 4.2 METHODOLOGY

An overview of the developed methodology is presented in Figure 4.2. The goal of the presented methodology is to improve GPU throughput and minimize the divergence of relative delay among the SMs. As GPU throughput, we define $T_{GPU} = \frac{I_T}{C_T}$, where $I_T$ stands for the total number of instructions executed on the GPU during $C_T$ total cycles. The methodology consists of two phases: i) the *application and kernel characterization*, and ii) the *adaptive SM allocation*. During the first phase, applications are profiled and classified. Information such as IPC, Memory Bandwidth, and the activity factor per SM are collected for each application for various SM configurations. The characterization phase collects information both at the application and the kernel level. During the second phase, we utilize the information extracted during characterization. Based on this information, the host (main processor) decides the number of SMs assigned to each kernel. The number of active SMs during a kernel execution is crucial for the GPU throughput and the aging distribution among SMs. The host monitors and records the activity of each SM in order to estimate its aging. We consider that on-chip delay monitors [38] provide input to the host regarding the status and the history of the SMs in terms of activity factor. Similar to [37], an on-chip power-gating unit [79] is used for clock-gating unused SMs. In this way, we achieve improved performance and balanced distribution of SM activity at the same time.

### 4.2.1 Application and kernel characterization

The goal of this phase is to collect the necessary information in order to guide the adaptive resource allocation to optimally allocate SMs. As demonstrated in Section 4.1.1, there is a double incentive for limiting the available SMs to a kernel. On the one hand, certain applications and kernels demonstrate higher IPC when they are executed on a limited number of SMs, instead of all the available SMs on the GPU. On the other hand, by limiting the available SMs to a kernel, we can clock-gate SMs that are not utilized by an application. By clock-gating computational resources we reduce their activity, thus we have the opportunity to distribute activity among the cores in a more balanced way. As a result, aging-induced relative delay is contained in smaller ranges, leading to a uniformly aged GPU. The purpose of the application characterization is to extract the acceptable configurations for the kernels of an application and for the application as a whole, for a specific GPU micro-architecture. Additionally, this step is executed only *once* and the extracted configurations will serve as the operating points for the SM allocation. This means that once the operating points are extracted, they can be shared and utilized by multiple GPU systems under different hosts, as long as the GPUs exhibit similar architectural characteristics. The operating points can be included as parts of the compiler and launcher. This is common in large scale deployments (e.g., cloud and server farms), where GPUs of the same type are used for back-end calculations for seamless integration and orchestration.

For each application, providing 5 up to 60 SMs with a step of 5, we collect the necessary profiling information[1]. As *acceptable configuration* of an application/kernel, we define the number of SMs that do not result in more than 10% IPC degradation, compared to the optimal IPC ($IPC_{OPT}$) that an application/kernel can achieve on a certain GPU. In other words, the collected operating points must satisfy $IPC \geq 0.9 \cdot IPC_{OPT}$. Provided that more than one operating points might achieve

[1]The detailed experimental setup of the GPU appears in section 4.3

optimal IPC, we define as *optimal operating point* for an application/kernel the configuration that provides the highest IPC ($IPC_{OPT}$) with the fewer possible SMs. The 10% IPC margin that we allow provides us enough space to explore configurations without sacrificing performance. At the same time, the potential to clock-gate enough SMs and reduce aging divergence becomes available.

As aforementioned, this step is executed only once and its duration for our experiments did not take more than a day. Thus, the aging and power impact of the characterization phase can be considered negligible. Executing applications on a GPU for one day does not significantly impact the aging condition of the device as the first noticeable impact on performance appears after 1.5 years of usage (Section 4.3).

The information extracted during the profiling phase, for an application $A$, is represented by the following tuple, $A(\nu, IPC, AcCyc, TotCyc, MB, L_2 \rightarrow L_1, MCr, \kappa, lst)$, where:

- $\nu$ is the number of SMs of the optimal operating point,
- $IPC$ is the IPC of the optimal operating point,
- $AcCyc$ is a $\nu$ size array of the active cycles for each SM, for the optimal operating point, and
- $TotCyc$ is the number of total cycles of execution for an application, for its optimal operating point.

Additionally,

- $MB$ stands for the memory bandwidth usage for the optimal operating point,
- $L_2 \rightarrow L_1$ is the $L_2$ to $L_1$ cache bandwidth usage, and
- $MCr$ is the memory to computational instructions ratio.

The latter three numbers are used to classify applications, based on the methodology presented in [29].

- $\kappa$, stands for the number of kernels that application $A$ consists of.

Finally, $lst$ is an array of lists of tuples, containing the necessary information for all the acceptable operating points for each kernel of an application. Each list contains tuples that represent the acceptable operating points of a kernel. The structure of each tuple is $(\lambda, IPC_{krnl}, AcCyc_{krnl}, TotCyc_{krnl})$, the description and the size of each member of the tuple are:

- $\lambda$ is the number of SMs for an operating point of a kernel, an 8 *bit* unsigned integer,
- $IPC_{krnl}$ is the kernel's IPC, for this operating point, a 16 *bit* unsigned integer,
- $AcCyc_{krnl}$ is a $\lambda$ size array of the active cycles of each SM for this operating point of a kernel, an array of 64 *bit* unsigned integers, and
- $TotCyc_{krnl}$ are the total cycles for the execution of a kernel, a 64 *bit* unsigned integer.

For application $A$, $lst$ is an array of size $\kappa$ and each element of this array is a list of tuples. The tuples of each list are ordered in descending order according to the number $\lambda$. For example, $lst[5]$ would be a list of tuples, describing the characteristics of the fifth kernel of the specific application. Thus, traversing the $lst$ array for every application, the adaptive SM allocation algorithm can find the necessary values to decide the operating point for a kernel. The space requirements to hold this information, for a GPU with 60 SMs, is up to $3.3KB$ per kernel. Considering the 10% IPC margin for the acceptable operating points, it is probable that for most of the kernels of an application, the acceptable solutions are fewer than all the possible operating points. Thus, the memory needed for the tuples of a kernel can be less than $2.3KB$, for a GPU with 60 SMs. This means that the information of a kernel can fit in the cache of a regular desktop computer.

### 4.2.2 Adaptive SM allocation

The goal of the adaptive SM allocation is to improve GPU throughput and minimize the divergence of relative delay among the SMs. The tasks of this phase are:

i) monitoring the activity history and temperature of SMs;

ii) selecting the operating configuration for a kernel among the acceptable ones;

iii) SM allocation;

iv) clock-gating of unused SMs; and

v) updating the activity history of each SM after the termination of a kernel.

#### Activity and temperature monitoring

The initial step in order to launch the kernels of an application on the GPU is to monitor the activity state of the SMs and the temperature of the GPU. These parameters are essential in order to estimate the status of the platform in terms of aging. The aging condition of the SMs is detrimental for the selection of the operating point as the number of used SMs for a kernel affects both GPU throughput and aging distribution. Before the host launches a kernel, it starts the aging estimator module. In this step, the estimator reads the timing, activity and temperature information form the GPU performance counters. Data describing the current GPU condition such as frequency, temperature and SM activity is stored by the estimator module. A thread of the estimator constantly polls the on-chip delay and temperature units to acquire this information. The estimator is then able to determine the current relative delay of each SM of the GPU using the aging model presented in Section 4.1.2. This information is used during the next task, combined with kernel profiling information, to estimate the aging condition of the GPU and select an operating configuration. The polling is performed in $30us$ windows. If a kernel has a duration less than the polling time, the estimator will use the previously acquired GPU information to determine the operating point and sort the SMs for the next kernel to be executed. We believe that this polling overhead is negligible as (i) GPU temperature cannot significantly change during a kernel execution of less than $30us$; and (ii) a kernel that executes for less than $30us$ (e.g., for $700MHz$ that is equal to $21,000$ *cycles*) does not significantly affect the condition of the SMs as applications have larger kernels that dominate the SM condition.

### Configuration selection

In this task, the host chooses the appropriate operating point of SMs, by utilizing the information extracted in the application and kernel characterization phase (Section 4.2.1) and the aging condition of the SMs provided by the aging estimator.

Choosing the configuration that provides the highest IPC per application is not necessarily the best choice, as it can have unpredictable and undesired results in terms of aging. That is the reason we allow the 10% IPC drop at the profiling phase. We set this threshold experimentally as we believe that it represents a good trade-off. Even though we set it to 10%, the proposed methodology works for any threshold value the designer may choose. For each kernel of an application, the operating point configurations are explored and decided individually. We perform this step because a general operating point extracted based only on the overall application performance can restrict specific kernels, leading to suboptimal throughput (Observation 3, Section 4.1.1).

Algorithm 1 describes in details the steps and decisions for *selecting the operating configuration for a kernel*. For each kernel of an application, all the tuples of the kernel list are traversed in order to decide the operating configuration. For a GPU with 60 SMs and operating points collected with a step of 5 SMs, the maximum number of tuples for a kernel are 12 and the time needed for this search is in the range of $1us$. For instance, for kernel $krnl$ of application $A$, all the tuples of the list $lst[krnl]$ are explored before choosing a configuration. In order for the host to determine a configuration for a kernel, the first step is to estimate the projected aging condition of the GPU if a specific configuration is used. To achieve this, the aging estimation module is provided with kernel profiling information: the active cycles ($AcCyc_{krnl}$) and total cycles ($TotCyc_{krnl}$) for the configuration under consideration. Having this information, the estimator module can project the aging condition of the GPU if the specific configuration is used. The aging module returns the standard deviation of the aging-incurred relative delay and the average relative delay of all the SMs (line 9). As the focus of this work is to minimize the activity divergence of the SMs, we prioritize the configuration that minimizes the standard deviation of the relative delay. From the acceptable configurations, members of $lst[krnl]$, we choose the one that will result in a GPU with minimum standard deviation of the relative delay (line 10). Prioritizing this choice guarantees that the SMs of the GPU will age as homogeneously as possible. Balanced aging among the cores is essential to avoid performance and reliability issues. If more than one configurations minimize the standard deviation of the relative delay (*RelDel_std*), we choose the one that provides the minimum average relative delay (line 16). With this choice, the methodology aims to reduce the overall aging of the platform, since balanced aging can be achieved by more than one configurations. In case that there are multiple configurations that minimize the relative delay standard deviation, combined with minimum average relative delay, we choose the one that achieves the highest IPC with fewer SMs (line 21). This decision aims to provide higher performance, if the prerequisites for homogeneous and low aging are met. When the configuration is decided, the next step is to allocate the required SMs.

### SM allocation

When this step is reached, the host has chosen the configuration for the SMs and it is time to allocate the computational resources for the kernel to be executed. If the chosen configuration requires all the SMs of the GPU, then all the SMs are assigned to the executing kernel and execution proceeds. If the configuration asks for $\kappa$ SMs, where $\kappa$

**Algorithm 1** Operating configuration selection

```
1: procedure CONFIGURATION SELECTION(A, krnl, ProfInfo, GPU_cond)
   ▷ Application A, kernel krnl of A, ProfInfo is the profiling information for A, GPU_cond are the characteristics of the GPU
2:     $RelDel\_std = +\infty$;
3:     $RelDel\_avg = +\infty$;
4:     $config = 0$;
5:     $maxIPC = 0$;
6:     $crrnt\_RelDel\_std = 0.00$;
7:     $crrnt\_RelDel\_avg = 0.00$;
8:     for all tuples in $lst_A[krnl]$ do
9:         $(crrnt\_RelDel\_std, crrnt\_RelDel\_avg)$ = agingmodule($AcCyc_{krnl}, TotCyc_{krnl}$)
10:        if $crrnt\_RelDel\_std < RelDel\_std$ then
11:            $RelDel\_std = crrnt\_RelDel\_std$;
12:            $RelDel\_avg = crrnt\_RelDel\_avg$;
13:            $maxIPC = IPC_{krnl}$;
14:            $config = \lambda$;
15:        else if $crrnt\_RelDel\_std == RelDel\_std$ then
16:            if $crrnt\_RelDel\_avg < RelDel\_avg$ then
17:                $RelDel\_avg = crrnt\_RelDel\_avg$;
18:                $maxIPC = IPC_{krnl}$;
19:                $config = \lambda$;
20:            else if $crrnt\_RelDel\_avg == RelDel\_avg$ then
21:                if $IPC_{krnl} > maxIPC$ then
22:                    $maxIPC = IPC_{krnl}$;
23:                    $config = \lambda$;
       return $config$
```

is less than the total number of SMs, the first step is to order the SMs according to their aging. Considering the current state of the SMs provided by the aging estimator module, the host orders the SMs based on their activity factor. For a GPU with $60SMs$, the time overhead for this sorting is less than $1us$. The $\kappa$ SMs with the lower activity factor are allocated by the kernel to be executed. The rest of the SMs are left to be clock-gated in the next step. By ordering the SMs and allocating the ones with lower activity factor, we achieve better balancing of the aging. The SMs with higher activity factor will be clock-gated, thus their activity will remain the same. On the other hand, the SMs with lower activity factor, will execute more instructions, thus increasing their active cycles. As a result, with this choice the proposed methodology decreases the activity gap among the SMs of a GPU.

### SM clock-gating

In this step, the configuration of the SMs has been chosen and the allocation of SMs has taken place. If the chosen configuration does not utilize all the available SMs, the remaining SMs are clock-gated during the execution of the kernel to reduce activity and idle power. As shown in [80], clock-gating can happen with 1 cycle of wake-up latency. This latency is equal to the time needed to reactivate an SM after it has been clock-gated. Thus, the approach of clock-gating SMs can significantly improve aging distribution among SMs, and reduce power, while not affecting performance. After the SMs are clock-gated, the execution of the kernel begins.

### Activity status update

When a kernel completes its execution, the host checks whether there are more kernels of the same application remaining to be executed. The host does not initiate the execution of a new application unless all the kernels of the previous one have finished.

To guarantee the correct estimation of the relative delay for the SMs, the activity history of each SM is updated after a kernel terminates. The cycles that each SM was active are added to the active cycles so far for each SM. Additionally, the total cycles for each SM are updated according to the sum of $\textit{active cycles} + \textit{idle cycles}$ per SM, for the last executed kernel. In this way, the host maintains an updated and consistent view of the status of each SM. As aforementioned, in case a kernel has a very short duration, shorter than $30us$, the aging estimator module uses the previously stored information to choose the operating point for the next kernel.

Similar to all functioning electronic components, the host experiences aging too. However, with this methodology we do not consider the aging impact of the host. A methodology for minimizing activity divergence on CPUs would require all functional

Table 4.1: Set-up for the Fermi micro-architecture

| Fermi GPU micro-architecture | | | |
|---|---|---|---|
| # of SMs | 60 | Core frequency | $700MHz$ |
| Warps per SM | 48 | Blocks per SM | 8 |
| Shared Memory | $48kB$ | $L_1$ Data cache | $16kB$ per SM |
| $L_1$ Instr. cache | $2kB$ per SM | $L_2$ cache | $768kB$ |
| Warp scheduler | GTO [81] | | |

Table 4.2: Set-up for the Tesla micro-architecture

| Tesla GPU micro-architecture | | | |
|---|---|---|---|
| # of SMs | 60 | Core frequency | $600MHz$ |
| Warps per SM | 24 | Blocks per SM | 8 |
| Shared Memory | $16kB$ | $L_2$ cache | $196kB$ |
| $L_1$ Instr. cache | $4kB$ per SM | Warp scheduler | GTO [81] |

blocks of the CPU to age with the same rate which is out of the scope of the presented research. Additionally, based on the aforementioned overhead analysis, the tasks that the host executes, e.g., aging estimation, temperature reading, SM ordering, are far less demanding compared to the GPU workload. As a result, the host does not experience intensive workloads under the scenario of the proposed methodology.

## 4.3 EVALUATION

To evaluate our solution we conducted extensive experiments using the open-source GPGPU-Sim [82] simulator and the Rodinia benchmarks [9]. The benchmarks were compiled with CUDA 3.2 and the simulator set-up allowed only single kernel launching per application. This means that for every application, only one kernel was executing at a time. GPGPU-Sim allows users to use different GPU architectures and configure various aspects of the GPU. To evaluate the developed method, we used the NVIDIA Fermi and Tesla micro-architecture. Specifically, in Table 4.1 we present the exact set-up of the Fermi micro-architecture, whereas in Table 4.2 we present the set-up of the Tesla micro-architecture.

To acquire the necessary temperature measurements that are required by the aging-model, GPUWattch [83] was used to extract power measurements for each simulation. Providing the power measurements and a GPU floorplan, the HotSpot [84] tool was used to estimate the temperature among the components of the GPU. Finally, we utilized the ExtraTime aging framework to estimate the relative delay [34]. We evaluated the proposed methodology under the following cases:

i) We considered that the GPU was executing a single application for a period of three years. In this step, the focus was given on evaluating the worst case relative delay for the SMs of the GPU.

ii) We created a more realistic scenario. In order to further evaluate the performance benefits of the proposed methodology as well as the impact on the aging divergence, we created seven queues of applications that we used as workload for the GPU. Four queues follow an application distribution according to the classification presented in [29], while the rest three mixes are composed of randomly picked applications from a benchmark pool.

For both cases, we compared the proposed methodology against four other approaches.

**Default:** In this approach each application was assigned all of the available SMs. This approach is an aggressive policy, meaning that the scheduler does not consider the activity factor of the SMs when assigning thread blocks to them.

**Profiling [30]:** Every application is profiled and we extract the configuration that yields the optimal IPC. Then, this number of SMs are assigned to the application. However, this method does not take into consideration the possible inequality of activity among the SMs.

**Aging-aware [38]:** In this method, the optimal configuration per application is also extracted. Then, the authors choose to apply an up to 8% performance loss margin in order to reduce the allocated SMs and always clock-gate some SMs. This approach orders the SMs based on their degradation rate and assigns the less degraded to the executing

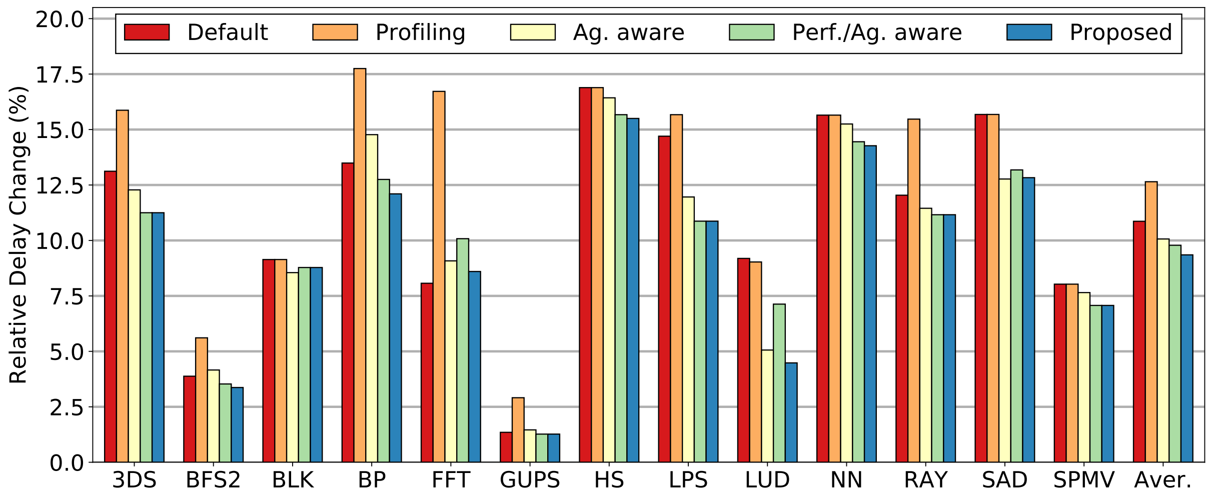


Figure 4.3: Worst (highest) relative delay change (Equation 4.1) among the 60 SMs of the GPU after three years

application, in accordance with the optimal number and the performance loss margin. However, they always select less number of SMs, compared to the optimal configuration which may not be the best solution for equal distribution of activity factor, especially for memory intensive applications.

**Performance- and Aging-aware [85]:** A methodology for SM allocation that boosts the overall performance and reduces the activity imbalances among SMs. It achieves these goals by clock-gating SMs. This methodology though, does not consider the kernel characteristics of applications. It considers applications at a coarse-grain level, overlooking the differences in performance and activity among the kernels of an application. Note that [85] presents an initial version of the presented technique.

### 4.3.1 Single application evaluation

The experiments in this section were conducted using only the Fermi micro-architecture set-up. Equation 4.1 describes the relative delay that an SM of the GPU suffers due to NBTI and HCI phenomena. Figure 4.3 depicts, for all the evaluated methods, the worst (highest) relative delay change (Equation 4.1) among the 60 SMs of

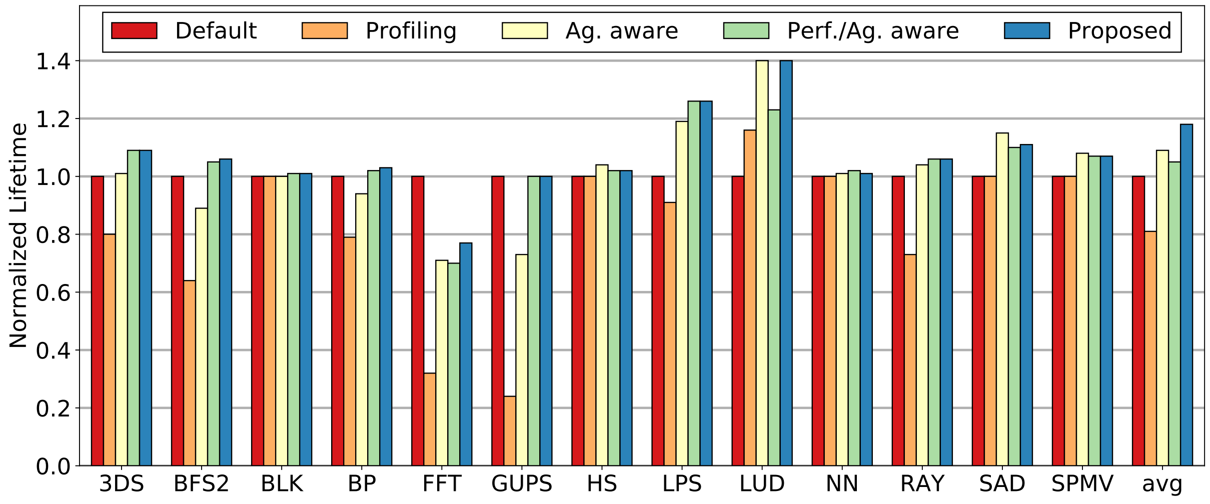


Figure 4.4: Normalized lifetime estimation per benchmark

the GPU after three years (1095 days). We plot this relative delay, normalized to the worst case relative delay of the *Default* methodology. The proposed methodology has the smallest impact, approximately 9.4% on the worst case relative delay comparing to all other approaches. Specifically, the *Profiling* method has the worst impact on the GPU with an average impact of 12.6%. This can be explained by the fact that it tries to maximize the performance without taking into consideration the activity and the history of the SMs. The *Aging-aware* approach manages to reduce the relative delay in the long run by applying a performance loss margin in order to reduce the number of allocated SMs. However, the number of SMs that this method utilizes is not the best solution for equal distribution of the activity factor, as further experiments (Section 4.3.2, *Diverse workload evaluation*) reveal. Finally, the *Performance- and Aging-aware* achieves similar worst case average relative delay comparing to the proposed methodology. However, for specific benchmarks, e.g., LUD and FFT, it results in greater worst case relative delay. This happens because, this method tries to balance performance and aging looking only at the application level (coarse-grain optimizations) rather than at the kernel-level.

Based on the worst case relative delay, Figure 4.4 depicts the normalized lifetime of the GPU. In order to calculate the lifetime, we utilized the method presented in [39].

Table 4.3: Application classification

| Class | Benchmarks | Class | Benchmarks |
|---|---|---|---|
| M | BLK, GUPS | C | BFS2, SPMV |
| MC | 3DS, BP, FFT, LPS, RAY | A | HS, LUD, NN, SAD |

Specifically, the lifetime of the GPU is defined as the time elapsed before at least one SM reaches the critical point. We set this critical point, as the worst relative delay among the SMs under the *Default* method at seven years. Overall, the proposed approach increased the lifetime of the GPU by 18% on average, while the *Aging-aware* method was the second best with an increase of 9% on average.

### 4.3.2 Diverse workload evaluation

In order to further explore the trade-offs of all the approaches and evaluate in depth the performance benefits of the proposed methodology as well as the impact on the aging divergence, we created a more realistic scenario. Specifically, we created seven queues of applications that we used as workload for the GPU. The applications are profiled and characterized before execution. To characterize the applications we used the methodology presented in [29]. Particularly, all applications belong to one of the four classes:
(1) M-oriented: memory intensive applications that access regularly the memory and demand large amounts of data to be transferred. (2) MC-oriented: memory-cache intensive applications have high memory activity, not high enough though to belong to the M category. Additionally they demonstrate high cache activity. (3) C-oriented: cache-intensive applications that utilize heavily data from the $L_2$ cache. (4) A-oriented: compute-intensive applications that demonstrate low main memory and cache utilization; applications in this category perform a high number of computational instructions.
Table 4.3 explicitly shows in which class each application belongs to.

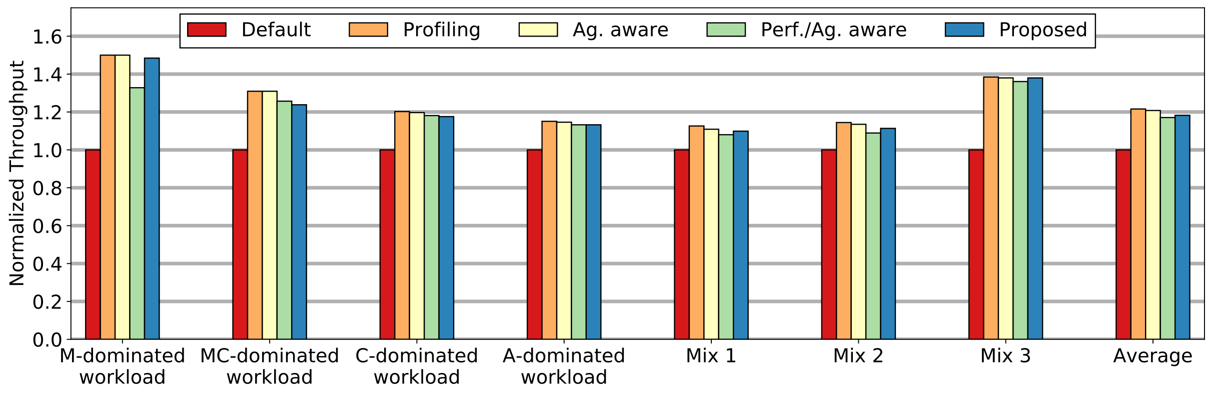


Figure 4.5: GPU throughput comparison per queue, Fermi micro-architecture

**Fermi micro-architecture**

Figure 4.5 depicts the total GPU throughput for the utilized queues during our experiments. The overall observation is that the proposed method always achieves higher throughput than the *Default* method. This is explained by the approach we follow for the SM allocation. Unlike the default approach, we profile applications at the kernel-level. After obtaining the required information, we can provide each application with the number of SMs that achieves optimal IPC, allowing a 10% IPC loss margin. The margin is necessary to keep aging balanced as much as possible. The proposed methodology achieves lower throughput than the *Profiling* and *Aging-aware* methodologies. For the former, this is explained because the *Profiling* methodology picks always the configuration that provides the highest application IPC, thus leading to the highest GPU throughput. Though, this decision does not come at no cost as the *Profiling* methodology does not consider the aging deviation among the SMs. For the latter, the *Aging-aware* methodology achieves higher GPU throughput as a result of a smaller performance trade-off. The *Aging-aware* methodology considers as acceptable, configurations that have up to 8% lower IPC than the optimal. This smaller space of acceptable configurations leads to higher GPU throughput but has a cost at balancing aging among the SMs. Comparing with the *Performance- and Aging-aware* methodology, for the *MC* and *C* workloads, the proposed methodology achieves marginally lower GPU throughput, whereas for the $A$ workload the two methods achieve the same GPU throughput. For the $M$ workload though, the proposed methodology achieves 9% higher GPU throughput. The differences between these two methods have to do with the kernel-level tuning. The *Performance- and Aging-aware* methodology picks an overall configuration for all the kernels of the applications, instead of the proposed methodology approach that picks configurations for each kernel individually. Memory-intensive applications demonstrate in general low IPC due to their data-bounded nature. Some kernels of a memory-intensive application might execute many calculations though, having a high IPC. As these applications demonstrate low IPC, the optimal configurations for the overall application consist of few SMs. This can severely impact a kernel that is compute-intensive but belongs to a memory-intensive application. The proposed methodology will not hinder the performance of such a kernel as it can choose a configuration with many SMs for the specific kernel and later restrict the available SMs for memory-intensive kernels. The same trend follows the three random mixes. Overall, the proposed methodology achieves increased GPU throughput by an average of 18% compared to the *Default* method.

Figure 4.6 depicts the relative delay caused by the SM activity, over the course of three years for all seven workloads. The coloured area corresponds to the activity divergence among the SMs. In other words, the smaller the area of a method is, the more balanced the activity is among the SMs. Generally we observe that the proposed methodology demonstrates smaller divergence comparing to all other approaches. The *Default* method presents lower values for the relative delay which is expected and explained by a low activity factor of the SMs. As Figure 4.5 shows though, this low activity factor means that throughput is kept low. The proposed method causes 89.8% lower standard deviation of the activity factor comparing to the *Aging-aware* approach together with 5.9% lower average activity factor for the MC-dominated queue. This is translated to 89.6% decrease in relative delay divergence among the SMs and 3.1%

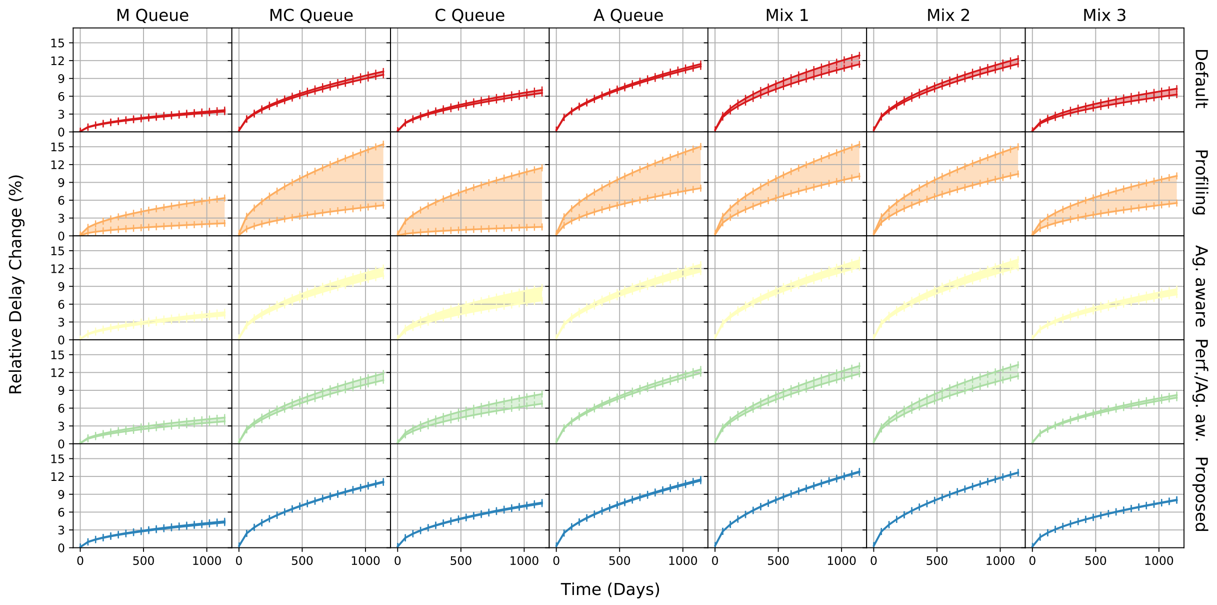


Figure 4.6: Relative delay incurred by aging per evaluated workload, Fermi micro-architecture

decrease of average relative delay for the MC-dominated queue. Additionally, the proposed method causes 87.3% lower standard deviation of the activity factor comparing to the *Performance- and Aging-aware* for the C-dominated queue, due to the fact that it takes into consideration the characteristics of the kernel employing a more fine-grain optimization.

Furthermore, Figure 4.7 presents the distribution of the activity factor among the SMs for all the evaluated approaches per queue. We can see that due to the fact that the *Profiling* method does not take into consideration the activity factor, it creates great activity inequalities among the SMs of the GPU. Also, the proposed method balances activity factor better among the SMs than the *Aging-aware* and *Performance- and Aging-aware* approaches creating a continuous spectrum of SM activity.

Last, Figure 4.8 depicts the normalized average power overheads for all the evaluated techniques for the seven workloads. On average, the proposed technique uses 1.4% more power than the *Default* method. This overhead is explained by the improved performance demonstrated by the proposed technique. Higher throughput means that the same

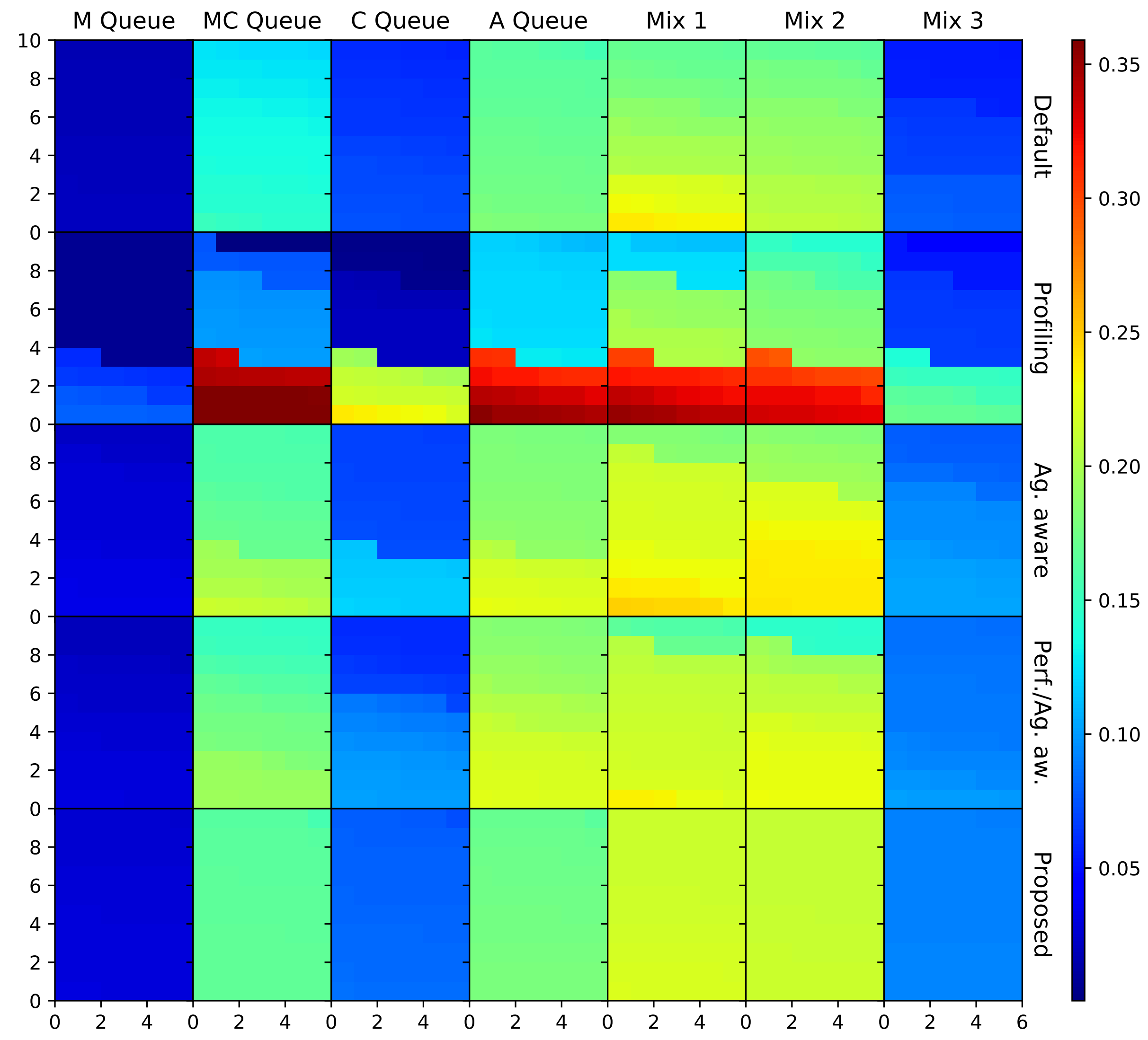


Figure 4.7: Activity factor per SM for each evaluated workload, Fermi micro-architecture [75]

amount of work is executed in shorter periods of time. As a result, SMs are active for a higher percentage of their total time while a kernel is executed. However, by clock-gating SMs, the proposed method succeeds in mitigating power overheads. Further mitigation of power overheads can be achieved by allowing a higher IPC margin during the characterization phase. Nevertheless, a higher IPC margin for acceptable operating points will result in lower throughput gains during execution. An overhead of 1.4% greater power on average, comparing to the *Default* method appears as a reasonable trade-off in

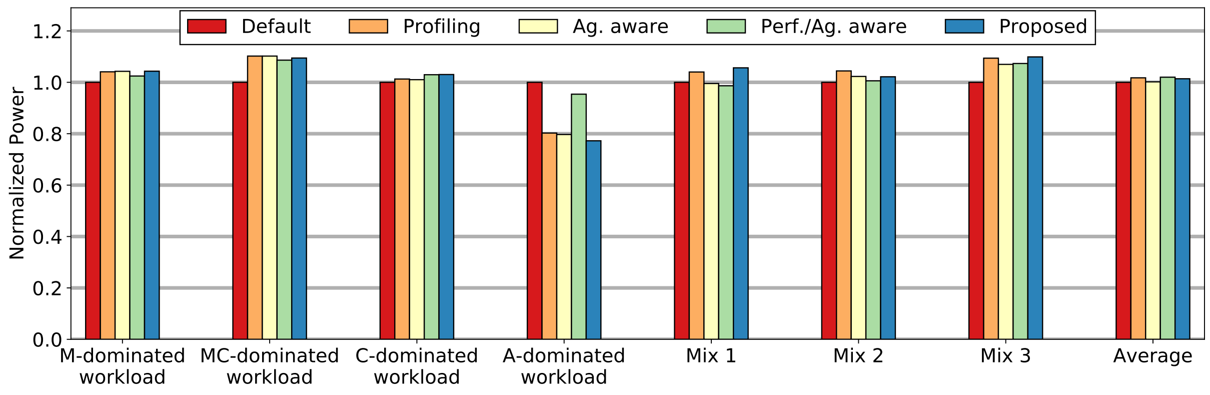


Figure 4.8: Normalized average power per evaluated workload, Fermi micro-architecture

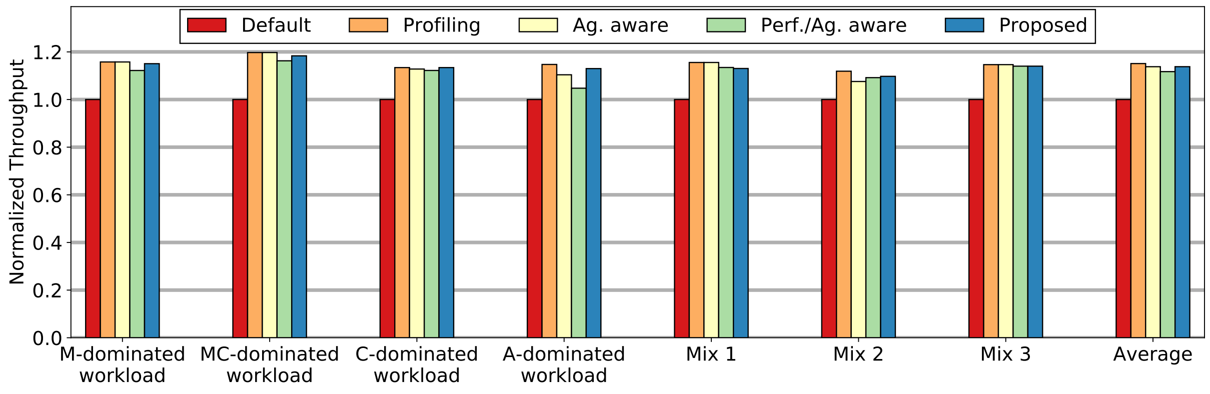


Figure 4.9: GPU throughput comparison per queue, Tesla micro-architecture

order to achieve 18% higher throughput on average, compared to the same method.

### Tesla micro-architecture

In this section, we present the experimental results while using the Tesla micro-architecture. We observe similar behavior of the proposed technique for the GPU with the NVIDIA Tesla micro-architecture as in the experiments conducted with the Fermi micro-architecture. Specifically, Figure 4.9 presents the normalized GPU throughput experimental results. On average, the proposed technique outperforms the *Default* approach by 13.8%. Additionally it outperforms the *Performance- and Aging-aware* technique except for the *Mix* 1 workload. We observe that all the methods improve their throughput against the *Default* approach by a lower percentage than for

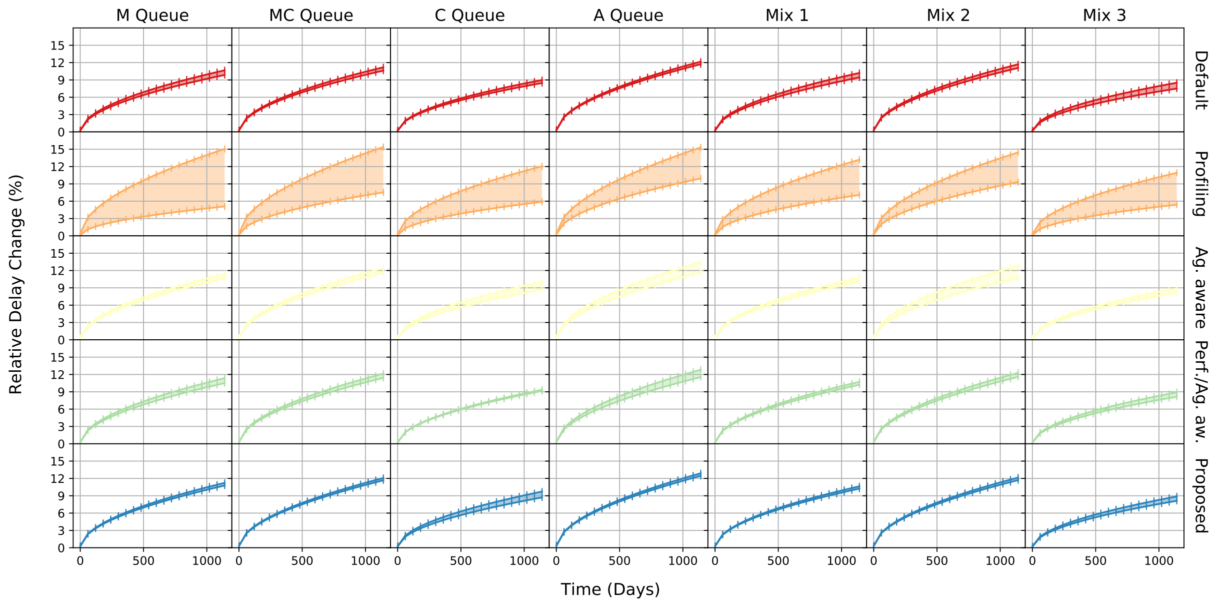


Figure 4.10: Relative delay incurred by aging per evaluated workload, Tesla micro-architecture

the Fermi micro-architecture GPU. This is a result of smaller caches, except for the $L_1$ instruction cache, smaller shared memory, and fewer warps per SM. Due to these architectural characteristics, the comparing methodologies do not achieve as high performance as achieved for the Fermi micro-architecture.

Figure 4.10 depicts the span of relative delay change among the SMs, during a period of three years. The proposed method demonstrates lower relative delay divergence compared to the rest of the methods, except for the $C$ queue, where it demonstrates 97.5% higher divergence than the *Default* method, and 3.9× higher divergence than the *Performance- and Aging-aware* method. Although, for the specific queue, the proposed method achieves higher throughput than the aforementioned methods. The higher throughput achieved by the proposed method causes a higher relative delay divergence among the SMs.

Accordingly to Figure 4.7, Figure 4.11 depicts the distribution of the activity factor among the SMs for the Tesla micro-architecture. For the majority of the queues, the proposed methodology demonstrates more balanced activity among the SMs, compared to

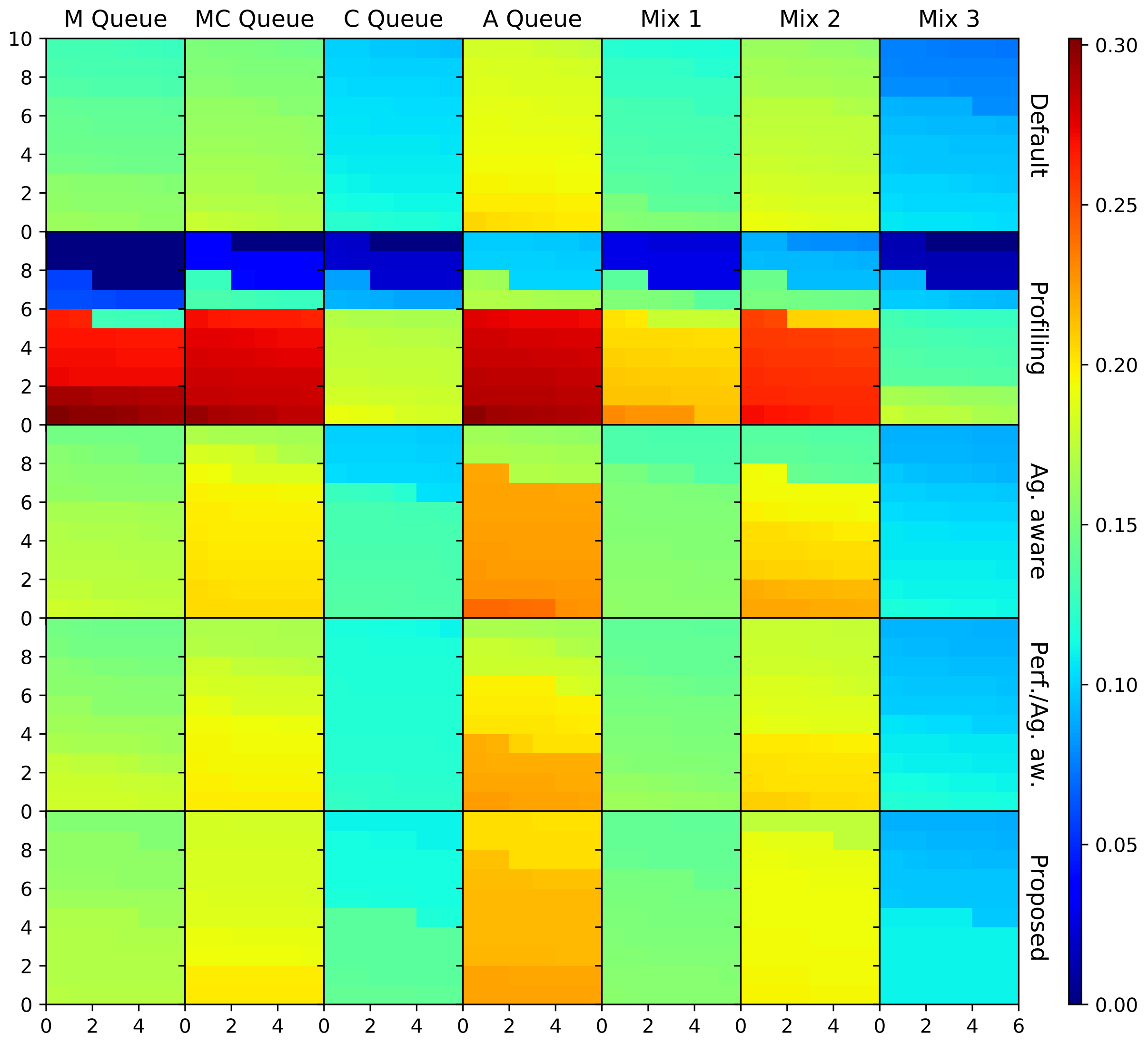


Figure 4.11: Activity factor per SM for each evaluated workload, Tesla micro-architecture [75]

the other approaches. Comparing to the experiments on the Fermi micro-architecture, we observe that the maximum value of the activity factor is lower for the Tesla micro-architecture. This is a direct result of the lower overall performance demonstrated for the Tesla micro-architecture, by all the methods.

Finally, Figure 4.12 presents the power results for the experiments on the Tesla micro-architecture. Overall, the proposed methodology demonstrates on average a 0.8% power overhead compared to the *Default* approach. However, the proposed methodology

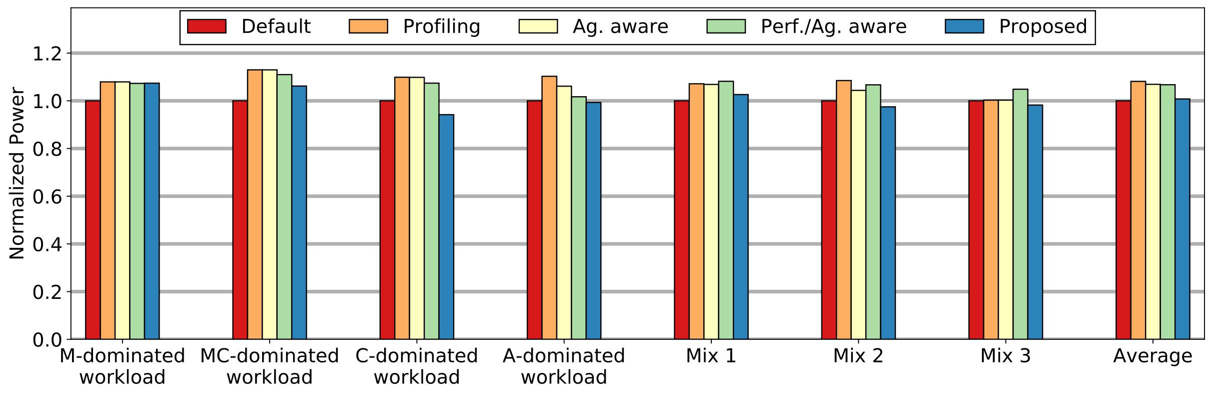


Figure 4.12: Normalized average power per evaluated workload, Tesla micro-architecture

outperforms the other three compared methods by achieving lower average power overhead per queue, for all the workloads used.

## 4.4 SUMMARIZING

In this chapter we presented an allocation methodology for SMs. The execution scenario for this methodology considers single application execution on GPUs. The developed methodology aims at improved GPU throughput and activity balancing for the SMs. It provides a fine-grain mechanism to allocate SMs, based on the kernel-level needs of each application. Profiling information per application is first gathered at an off-line stage. This information is necessary to decide the configurations per application and per kernel during execution. Experimental results for the NVIDIA Fermi and Tesla GPU micro-architectures demonstrate that the proposed methodology achieves its goals. Compared to the default approach for SM allocation, and state-of-the-art allocation policies, the proposed methodology achieves higher throughput and balances activity among the SMs. Even though on average, power consumption is higher when using the proposed methodology compared to the default allocation policy, that is an acceptable trade-off, given the higher performance that is achieved by the proposed methodology.

# CHAPTER 5

# PERFORMANCE-BASED AND ACTIVITY-AWARE RESOURCE ALLOCATION FOR CONCURRENT GPU APPLICATIONS

In this chapter we present the developed methodology for concurrent applications, executing on a GPU. The methodology focuses at improving system throughput while balancing aging among the SMs. We first present some necessary background information. Then we thoroughly present the methodology, and finally we evaluate it with extensive experiments.

## 5.1 PRELIMINARIES

### 5.1.1 Problems in concurrent application execution

Concurrent application execution faces two major challenges. First, the throughput of the GPU can vary significantly depending on the pairing of the applications and the configuration of the processing units to each application. Second, the computing activity among the processing units is highly non-uniform, leading to aging divergence in the course of time.

GPU's throughput and applications' performance can change considerably depending on the characteristics of the applications that are executed together and the SM partitioning between them. Specifically, applications vary in terms of number of commands, the nature of these commands, e.g., compute or memory intensive, the memory bandwidth they utilize, last level cache misses etc. Based on these metrics, we can classify applications and measure their inter-application interference [29]. Particularly, computationally intensive applications require a large number of SMs, as they have to execute more compute-based instructions. Consequently, the more SMs they have available the higher throughput they can achieve. On the contrary, memory intensive applications do not need a high number of SMs to achieve the highest possible throughput as memory bandwidth utilization is the most important resource for them. Thus when executing applications concurrently on the same GPU, we need to take into consideration the types of the applications when placing them together, as this will significantly affect the overall system throughput. Additionally, a very important aspect that determines the system's throughput is the partition of the number of SMs between two co-executing applications. Partitioning the SMs equally between the two applications, although it might seem fair, it cannot guarantee high throughput in application or system level. Applications with low inherent IPC harm the system's overall throughput and they do not benefit from a higher number of SMs. Nevertheless, applications that can achieve high IPC, benefit from the use of more SMs and will contribute in a higher throughput for the system.

The second problem of concurrent application execution on GPUs is that SMs demonstrate varying levels of activity during execution. Due to the different nature of each application (e.g., computationally or memory intensive) and the different number of instructions executed per application, SMs are utilized non-uniformly. In the worst case scenario, if every time we execute a pair of applications, the application with the less active cycles gets mapped on the same SMs, the system will demonstrate a bipolar image in terms of activity and as a consequence aging. The effects of aging can be even intensified by the existence of PV. PV describes the differences that can occur in chips and electronic components, even on the same wafer. These differences occur due to material impurities or imperfections during the fabrication process.

Specifically, the inability to control manufacturing parameters and processes in sub-wavelength lithography has significantly affected PV [86]. PV negatively affects the maximum frequency of transistors and, consequently, the core frequency and the leakage power. As described in [49], PV consist of two main components, Die-to-Die (D2D) and Within-Die(WID). D2D variations equally affect all the transistors of a die, and they are a result of within-wafer, wafer-to-wafer, and lot-to-lot variations during fabrication. WID variations affect transistors of a die in different ways, and are further divided into random and systematic.

In [52], the authors introduced the concept of Core-2-Core (C2C) variations, which are a result of spatially correlated WID variations. Due to the diminishing size of cores, allowed by progress in material and fabrication technologies, more SMs than ever are integrated in a GPU. This increment in the number of SMs negatively affects process variations, as it subsequently increases the probability of spatially correlated phenomena among cores.

In this work, we focus on C2C variation effects by considering SMs as the computational cores. As presented in [48], for GPUs consisting of $N_{SMs}$ identical SMs, the chip surface is represented by a $N_{chip} \times N_{chip}$ grid. The process variation value $p_{ij}$ of a grid cell ($i, j \in [1, N_{chip}]$) can be defined as a Gaussian random variable. Furthermore, the process variation values of two points on the grid demonstrate correlation. This correlation is measured by the correlation coefficient of variation which is described by Equation 5.1:

$$\rho_{ij,kl} = e^{-\alpha\sqrt{(i-k)^2+(j-l)^2}}, \forall i, j, k, l \in [1, N_{chip}] \tag{5.1}$$

where $\alpha$ represents how quickly spatial correlations die out.

The scope of the developed methodology focuses on SMs, which is a higher abstraction level than the transistor-level. As shown in [87], the frequency of a SM, which is affected by process variation, can be approximated by the worst-case delay of identical critical paths. To facilitate calculations and allow the methodology to be adaptive to run-time changes, we assume that the gates of a critical path lie within a grid cell and that critical paths are uniformly distributed within a SM. We base these assumptions on previous research approaches [48, 87], where similar PV estimation models were utilized.

Thus, the maximum frequency of each SM $i \in [1, N_{SM}]$ is:

$$f_i^{MAX} = K' \min_{k,l \in S_{CP,i}} \left( \frac{1}{p_{kl}} \right) \tag{5.2}$$

In Equation 5.2, $K'$ is a technology-dependent constant, and $S_{CP,i}$ stands for a set of grid cells that contain critical paths in SM $i$. The initial operating frequency of each SM is estimated by Equation 5.2. SM frequency directly affects threshold voltage shift (Equation 4.7) as well as $\delta_B$ (Equation 4.3) and $\alpha_B$ (Equation 4.8). Consequently, PV affects the operating frequency of the SMs and eventually the relative delay incurred by aging (Equation 4.1). Due to PV, the SMs of a GPU demonstrate different operating frequencies. Disregarding the PV effects will lead to inaccurate estimation of aging induced relative delay. Accordingly, inaccurate estimation of relative delay will lead to unpredictable allocation of SMs. Thus, the activity of the applications will not be distributed in a way that can balance the aging condition of the SMs. Specifically, compute intensive applications tend to demonstrate high activity, contributing in a more aggressive aging. Thus, if the aging condition of SMs is not accurately estimated, any effort to balance aging among SMs will fail, and the aging imbalances of the GPU components will worsen.

Ultimately, the aging imbalance can cause unexpected delays in execution, reduce reliability and even cause malfunction to a system due to some parts of it being very aged while the others have not yet reached a critical point. Thus the Mean-Time-To-Failure (MTTF) for a GPU can shorten, even though the system has areas that are underutilized.

### 5.1.2 Concurrent application execution on GPUs

Multi-application execution can greatly affect the performance of GPUs, due to contention on shared resources (e.g., $L2$ cache, memory controller), and at the same time significantly affects the aging rate of the platform in a non-uniform way. The goal of the

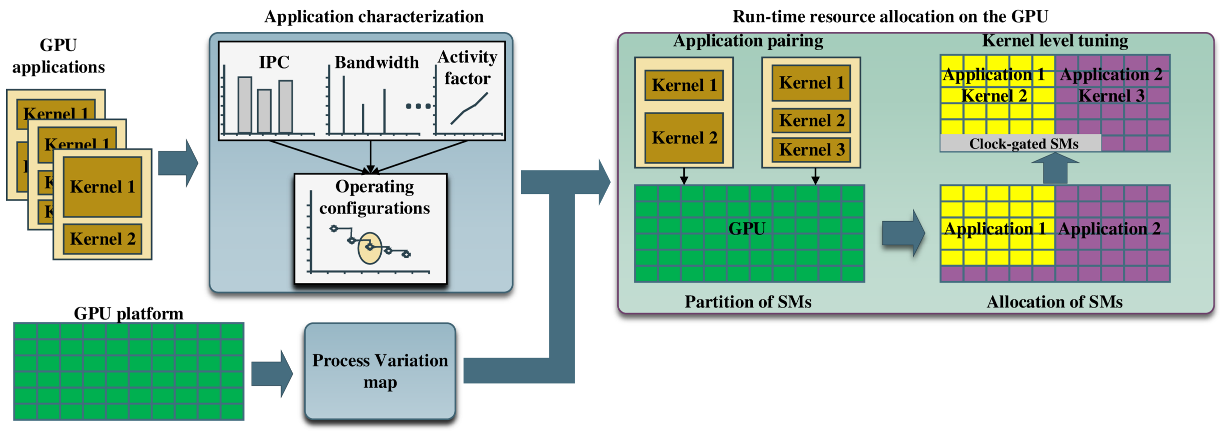


Figure 5.1: Overall flow of the proposed methodology [88]

proposed methodology is to increase the throughput of a GPU under concurrent application execution while balancing the aging on the computing components. As throughput we define $T = \frac{I_T}{C_T}$, where $I_T$ is the total number of instructions and $C_T$ is the total number of cycles.

Figure 5.1 presents an overview of the proposed methodology. The starting point is application characterization where the goal is to gather information regarding the performance, e.g., Instructions Per Cycle (IPC), memory bandwidth utilization, Memory-to-Compute instructions ratio, and the activity factor per SM for various applications. Based on this information we extract operating configurations per application that will drive the next steps. Assuming a queue of incoming applications for execution and given the process variation map of the GPU, we:

i) Select pairs with low inter-application contention;

ii) Decide the number of SMs per kernel per application in order to further increase GPU's throughput; and

iii) Allocate the SMs based on the compute and aging history of the SMs of the platform, the activity factor of each application and the GPU's PV.

iv) Clock-gate SMs during the execution of different kernels if overall throughput is not

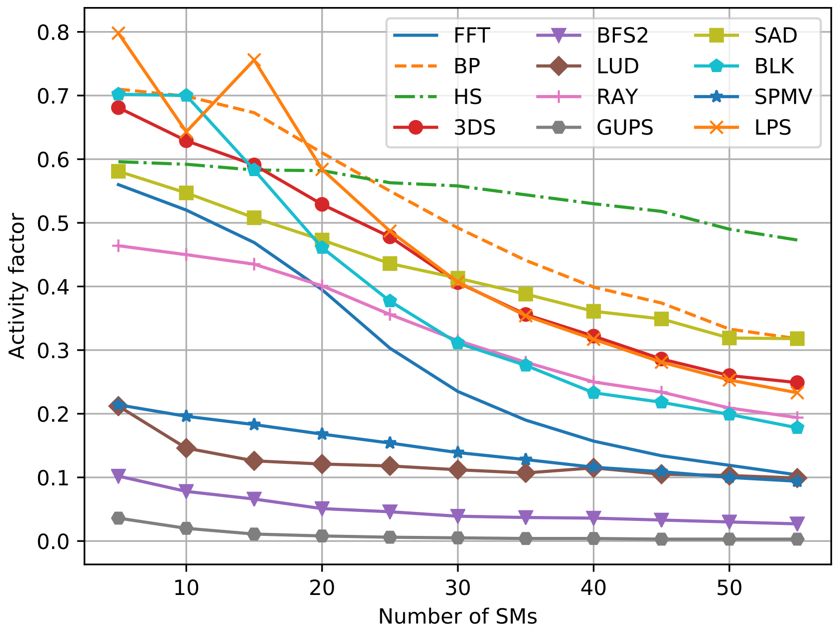


Figure 5.2: Application activity factor for different configurations [8]

significantly affected.

After the end of an application or pair of applications, the aging history of each SM is updated based on the impact of the activity factor on the utilized SMs.

## 5.2 METHODOLOGY

### 5.2.1 Application characterization

The first step of the proposed methodology consists of information extraction for GPU applications. Given a GPU with $n_{tot}$ number of SMs, we profile applications for different SM configurations. Metrics regarding application performance and characteristics are extracted in order to serve the run-time SM allocation and management.

For a given application $A$, we execute it on the GPU starting from 5 SMs up to 55 with a step of 5 SMs. For each configuration, we collect, among other information, the IPC and activity factor. At a finer level, for each application we also collect the IPC for

Table 5.1: Kernel information

| **Applicat.** | **# of kernels** | **Kernel IPC compared to application IPC** |
|---|---|---|
| 3DS | 1 | 1 |
| BFS2 | 12 | $0.40, 0.06, 0.36, 0.06, 0.28, 0.18, 1.28, 0.12, 1.53, 0.18, 1.15, 0.03$ |
| BLK | 1 | 1 |
| BP | 2 | $0.69, 1.51$ |
| FFT | 8 | $1.31, 1.12, 1.13, 0.94, 0.89, 0.86, 0.92, 0.88$ |
| GUPS | 1 | 1 |
| HS | 2 | $0.99, 1.01$ |
| LPS | 1 | 1 |
| LUD | 46 | $49.21, 1.79, 0.10, 41.28, 1.70, 0.09, 41.28, 1.83, 0.10, 41.28, 2.16, 0.12, 41.28, 2.16, 0.12, 41.28, 2.37, 0.11, 41.28, 2.63, 0.13, 41.28, 2.96, 0.13, 41.28, 3.39, 0.15, 41.28, 3.95, 0.17, 41.28, 4.74, 0.21, 41.28, 5.92, 0.25, 41.28, 7.89, 0.30, 41.28, 11.84, 0.82, 41.28, 22.21, 1.97, 41.28$ |
| RAY | 1 | 1 |
| SAD | 3 | $0.56, 17.32, 18.94$ |
| SPMV | 1 | 1 |

all its kernels. Figures 1.2 and 5.2 depict the IPC and the activity factor respectively, for each application and for all the tested configurations. Also, Table 5.1 presents how many kernels each application has, and the $\frac{application\ IPC}{kernel\ IPC}$ ratio for the configuration in which the application had the highest IPC. From Table 5.1, we notice that certain kernels demonstrate higher IPC than the actual IPC of the application ($\frac{application\ IPC}{kernel\ IPC} < 1$). This is explained by the fact that the duration of some kernels is significantly shorter than the duration of the application, e.g., 100× shorter. As a result a kernel's contribution to the application IPC can be minimal. Based on Figures 1.2, 5.2 and Table 5.1, we make the following observations.

***Observation 1***: Applications can be divided in three categories in terms of their IPC behavior: In Section 5.2.2, applications of a queue are matched into pairs. After that, in Section 5.2.2, SMs are partitioned among the applications of a pair and in Section 5.2.2, specific SMs are allocated to each one of them. At (i) applications that continuously increase their IPC when more SMs are available, e.g., HS; (ii) applications whose IPC remains constant after a certain number of SMs, e.g., GUPS; and (iii) applications that decrease their IPC after the available SMs exceed a threshold, e.g., FFT. To summarize, in order to increase application IPC, there is an incentive to provide more SMs to certain applications, whereas in other cases there is incentive to limit SMs up to a certain number.

***Observation 2***: Regarding the activity factor, applications can be divided into two categories: (i) applications whose activity factor continuously drops as the number of available SMs increases, e.g., 3DS; and (ii) applications whose activity factor remains constant or demonstrates a negligible change as more SMs are available, e.g., GUPS. Thus, increasing the available SMs for an application is very beneficial when the activity factor drop is combined with an IPC increase.

***Observation 3***: Kernel IPC compared to application IPC varies significantly, e.g., it can be up to 49× smaller.

***Observation 4***: The IPC among kernels of the same application might vary significantly, e.g., up to 417×. This difference is explained as kernels can perform distinctively different tasks. For example, one kernel may fetch data from memory, thus being memory-bounded and demonstrating low IPC. On the other hand, a different kernel of the same application may perform heavy computations thus demonstrating high IPC.

*Observations 1* and *2* can be justified by the behavior of the applications. Computationally-intensive applications will benefit by large number of SMs as they can increase throughput.In Section 5.2.2, applications of a queue are matched into pairs. After that, in Section 5.2.2, SMs are partitioned among the applications of a pair and in Section 5.2.2, specific SMs are allocated to each one of them. At At the same time, the activity factor of the SMs will be reduced, as the workload will be distributed among more SMs. On the other hand, applications whose IPC does not increase as the number of SMs increases are more memory bounded and their activity factor remains relatively unaffected.

For each application, we define as $\nu$ the minimum number of SMs that provides the maximum IPC for an application. The ideal configuration for an application $A$ is represented as $A^{\nu}(IPC, MB, L_2 \rightarrow L_1, MCR, AcF, \kappa)$, where $IPC$ is the *Instructions Per Cycle*, $MB$ is the *memory bandwidth*, $L_2 \rightarrow L_1$ is the *$L_2$ to $L_1$ cache bandwidth*, $MCR$ is the *memory to compute instructions ratio*, and $AcF$ is the *activity factor*. Last, $\kappa$ ($\kappa < \nu$) describes the number of SMs for which the IPC of $A$ does not drop more than 20% than the ideal configuration. This threshold is determined experimentally, and it is a trade-off between system throughout and application progress. A lower threshold would deprive the run-time system of the necessary flexibility during SM partition, as compute intensive applications would throttle applications with low SM requirements. A higher threshold would harm the overall system throughput as compute intensive applications would allow high margins of IPC drop, thus reducing overall system throughput. Additionally, the IPC per kernel of $A$ is collected. For every kernel $x$ of $A$, we keep the following information: $A^{i}_{x}(IPC)$, $i \in [5, \nu]$.

### 5.2.2 Run-time resource allocation on the GPU

The proposed GPU run-time resource management provides improved performance and minimizes the aging divergence of SMs. The steps that are executed during this phase are the following: i) application pairing, ii) SM partitioning, iii) allocation of SMs, and iv) kernel level tuning.

#### Application pairing

Assuming a queue of applications, the first step of the run-time methodology is to decide how to pair applications in order to increase the GPU throughput. According to [29], applications can be classified in four categories based on their behavior. We use the information obtained off-line, $IPC, MB, L_2 \rightarrow L_1$ and $MCR$, as well as a modified version of the ILP methodology presented in [29] to pair the applications of the queue, in a way that will minimize slowdown during concurrent execution. Given an initial queue of applications, the resource manager pairs the applications and selects for execution the pair with the lowest slowdown. During this stage, the composition of the queues, in terms of applications, as well as the arrival time or sequence of applications is not known a priori. More applications may arrive to the queue while a pair of applications is executing on the GPU. Once the pair finishes (non-preemptive execution), the algorithm recalculates the best matching of applications and pushes to the GPU the pair with the lowest slowdown. Minimizing slowdown serves a double purpose. First, it leads to higher overall GPU throughput, and second, it has less contribution to aging since lower slowdown is equivalent to less activity of the SMs.

**SM partitioning**

After deciding the application matching, the run-time resource allocator decides, based on the profiling information, how many SMs will be allocated by each application. The goals of this step are the following: i) to provide high GPU throughput, and ii) to minimize activity divergence among SMs in order to balance aging effects. Having to partition $n_{tot}$ SMs between two applications $A$ and $B$, we distinguish the following cases:

- if $\nu_A + \nu_B \leq n_{tot}$, we provide each application with the SMs of its ideal configuration. In case $\nu_A + \nu_B < n_{tot}$, the surplus of SMs will be clock-gated after SM allocation.
- if $\nu_A + \nu_B > n_{tot}$, we decide to favor the application with higher ideal IPC. This choice is based on *Observations 1, 2* and the fact that the application with higher IPC will contribute to higher overall throughput. In this scenario we can distinguish the following cases. Assume that $IPC_A^{\nu} > IPC_B^{\nu}$.
  - If $\nu_A + \kappa_B \leq n_{tot}$, we assign $\nu_A$ SMs to application $A$ and $n_B$ SMs to application B, where $\kappa_B \leq n_B < \nu_B$ and $n_B + \nu_A = n_{tot}$.

– If $\kappa_B + \nu_A > n_{tot}$, we check whether $\kappa_A + \kappa_B \leq n_{tot}$. If the last inequality is satisfied, we proceed with the allocation. If not, application B requests fewer SMs, down to the limit of 5. If again $5 + \kappa_A > n_{tot}$, then application $A$ requests fewer SMs, until both applications request a sum of fewer than $n_{tot}$ SMs.

By prioritizing the needs of the application with higher ideal IPC, we achieve better distribution of the activity factor among the SMs (*Observation 2*), and we avoid small regions of very high activity. Such regions would lead to fast, regional aging. We avoid system performance degradation due to concurrent execution i) by using a modified version of the pairing methodology presented in [29] that minimizes system slowdown, and ii) by favoring the needs of the application with higher ideal IPC. Even if an application of a pair is executed on 5 SMs, system throughput will remain high.

**Allocation of SMs**

Once the number of SMs per application is decided, in this step we determine exactly which SMs will be tied to each application. During this process, the $n_{tot}$ SMs are ordered according to their aging condition. We estimate the aging of a SM, based on Equation 4.1, by keeping track of their activity factor, their operating frequency as determined by PV, and their temperature. Specifically, temperature contributes in the calculation of Equations 4.5 and 4.6, which in their turn contribute in Equations 4.2 and 4.7 that calculate the threshold voltage change caused by NBTI and HCI phenomena respectively. The SMs with lower relative delay are less aged, compared to SMs with higher relative delay. We assume that application $A$ allocates $n_A$ SMs, application $B$ $n_B$ SMs, and $AcF_A^{n_A} > AcF_B^{n_B}$. The inequality states that the average activity factor, of application $A$ for $n_A$ SMs, is greater than the corresponding average activity factor for $B$, configured with $n_B$ SMs. After the SMs are ordered according to their aging condition, the most aged SMs will be given to the application that demonstrated lower activity factor during profiling. In our example, the $n_B$ most aged SMs will be given to application $B$. If $n_A + n_B$ is less than the total number of SMs, the remaining SMs are clock-gated.

The application with higher activity factor will benefit by allocating the less aged SMs both in terms of performance and aging balancing. According to the correlation observed from Figures 1.2, 5.2, applications with high activity factor demonstrate also high IPC. Less aged SMs function in higher frequencies. Thus, the application with higher activity factor will utilize them more efficiently. Furthermore, the less aged SMs will be more active and as a result their activity factor will increase, leading to aging balance among SMs.

Our approach provides higher throughput while balancing aging comparing to [53], which considers only the variation of each SM and assigns the SMs with higher frequency to the application with higher IPC. Even though SMs with higher frequency benefit the application with high IPC, the nominal frequencies suffer from fluctuations due to aging. As a result, it is more accurate to order SMs according to their aging, considering also their initial PV.

**Kernel level tuning**

In this step, we present a fine-grain kernel-based tuning. First, we decide whether to clock-gate SMs that are already assigned to an application. Second, we reorder the SMs allocated by an application, thus achieving better distribution of aging.

If a kernel does not contribute to high application IPC, we clock-gate SMs during its execution so as to reduce the activity of the SMs, without sacrificing significant performance. Before a kernel is launched, its IPC is compared to the IPC of the application. The decision regarding whether and how many SMs will be clock-gated is taken by the following statement. For a kernel $x$, of application $A$, with $n_A$ SMs allocated, if $z \cdot IPC_x^{n_A} \leq IPC_A^{n_A}$, $z \in \mathbb{N}$, clock-gate the greatest number of SMs $\mu$, satisfying $5 \leq \mu < n_A$ and $IPC_x^{\mu} \geq p \cdot IPC_x^{n_A}$, $p \in \mathbb{Q} \wedge p \in (0, 1)$. With the aforementioned statement, we clock-gate SMs only if the kernel has IPC at least $z\times$ lower than the IPC of the application. Additionally, we clock-gate as many SMs as possible, without sacrificing more than $1-p$ of the kernel's IPC achieved with $n_A$ SMs.

Furthermore, before executing each kernel, we order the application SMs based on their aging. This ordering is driven by SM activity, frequency and temperature. This allows for finer balancing of aging, given that some SMs can be clock-gated during certain kernels. Thus, we take full advantage of clock-gating by distributing aging among SMs as equally as possible.

As a closing remark, the proposed methodology reduces aging divergence on SMs regardless of the input of the application. The scope of the proposed methodology focuses on SMs as the fundamental component block. We need to mention that focusing on the SM level does not allow us to consider circuit inputs at the gate level. For that reason, we adopted the model from [34] that estimates aging at the SM level. As previous research works have shown, this aging model, which works on component blocks, achieves an adequate aging estimation for the SMs [36, 37, 38, 85]. However, the resource allocator can handle different inputs for the applications by updating the activity factor of SMs in frequent intervals and ordering SMs according to their relative delay. Consequently, even if an application changes behavior due to a different input, the proposed methodology will adapt and balance aging among the SMs.

## 5.3 EVALUATION

To validate our methodology, we performed extensive simulation experiments using a modified version of GPGPU-sim [82] that supports concurrent application execution, and Rodinia [9] benchmarks as high performance parallel applications. We used the GPUWattch [83] simulator to acquire power measurements. The power measurements together with the GPU floor-plan are given as input to HotSpot [84], which outputs the temperature of the GPU. The experimental GPU set up is described in Table 5.2.

Table 5.2: Experimental set up

| GPU Architecture | | | |
|---|---|---|---|
| # of SMs | 60 | Core frequency | 700MHz |
| Warps per SM | 48 | Blocks per SM | 8 |
| Shared Memory | 48kB | $L_1$ Data cache | 16kB per SM |
| $L_1$ Instr. cache | 2kB per SM | $L_2$ cache | 768kB |
| Warp scheduler | GTO [81] | | |

In order to evaluate the proposed methodology, we created five queues with incoming applications based on Rodinia benchmarks. The benchmarks were profiled off-line to extract the necessary characterization information. Profiling is a process that can be completed in a few hours utilizing the GPGPU-Sim simulator. Following the classification in [29], the queues are:

- *M-oriented workload*: Memory-bounded applications dominate the queue.
- *MC-oriented workload*: Memory-cache applications dominate the queue.
- *C-oriented workload*: Cache-bounded applications dominate the queue.
- *A-oriented workload*: Compute intensive applications dominate the queue.
- *Equal distribution*: The queue contains equal number of applications from the 4 classes.

For a queue to be characterized as oriented towards a specific class, at least 60% of the applications of the queue need to belong to the certain class. To incorporate PV into our experiments, we produced 50 PV maps. The results we present are an average value of the throughput, relative delay, and activity factor for the 50 PV maps.

We compare the proposed methodology to the following ones:

- a *First Come First Served (FCFS)* approach that co-executes applications based on their arrival order and distributes the SMs equally between applications.

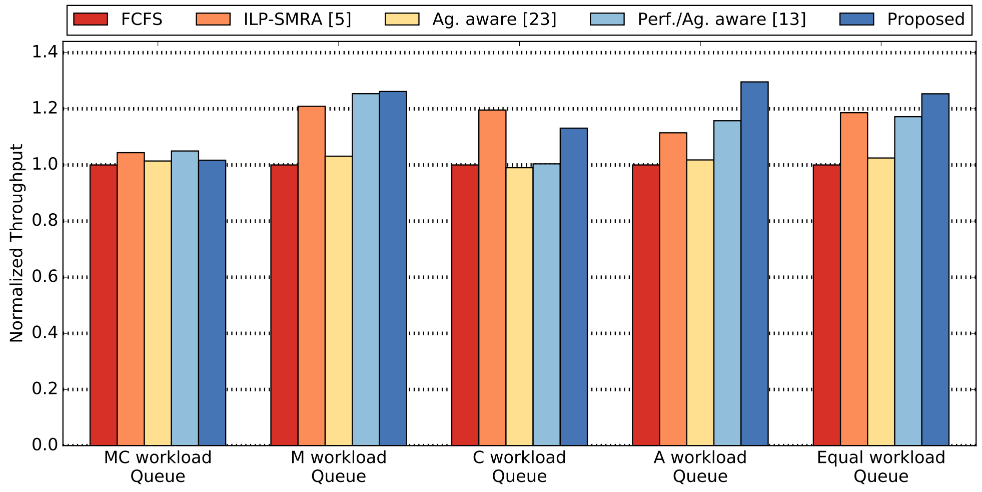


Figure 5.3: GPU throughput comparison per queue [88]

- the *ILP-SMRA* method presented in [29]. This approach follows a classification scheme for the selection of the pairs and initially divides the SMs equally among the applications. At run-time, SMs are reallocated in order to maximize the GPU throughput.

- an *Aging aware* method based on [38]. This method focuses on optimizing aging and power of the GPU. It is initially designed for one application being executed on the GPU, but to tune it for two concurrent applications, we profiled each application for up to 30 SMs. Then, we paired the applications on a FCFS way and we divided the SMs equally, 30 for each application. Each application used only the necessary number of SMs to achieve the maximum IPC possible, clock-gating the rest of the SMs.

- a *Performance/Aging aware* method presented in [8]. This methodology improves performance while it tries to keep aging divergence of SMs low. Nevertheless, this methodology does not consider PV effects and does not make SM allocation decisions based on kernel characteristics.

Figure 5.3 presents the GPU throughput comparison for the five queues. For each

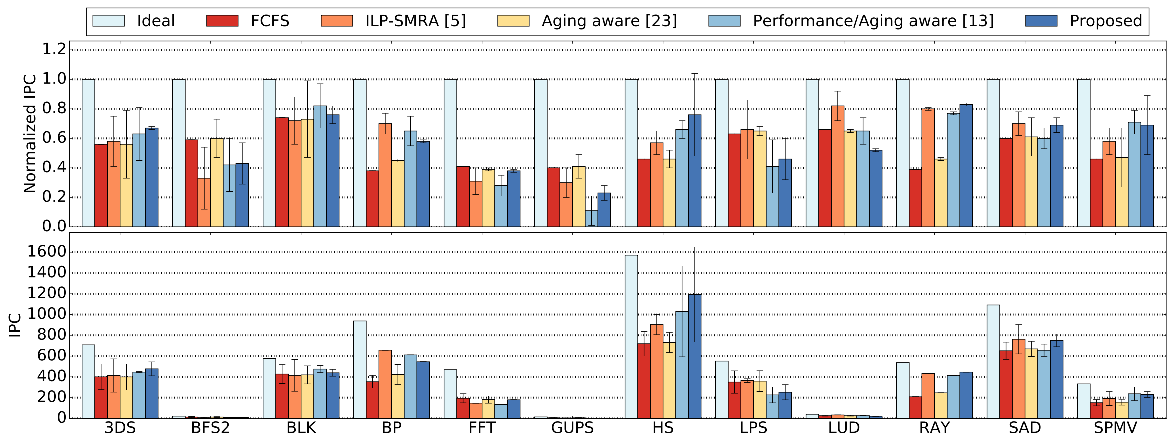


Figure 5.4: Normalized and absolute IPC per application [88]

queue, the throughput is normalized based on the *FCFS* method. An initial comment is that the proposed method always outweighs the *FCFS* and the *Aging aware* methods in terms of performance. This is expected as the latter methods do not emphasize on improving performance. The proposed method achieves up to 30% higher GPU throughput compared to the *FCFS* method, and up to 27% higher throughput compared to the *Aging aware* method. Comparing to *ILP-SMRA*, for the MC and C queues, the proposed method achieves 6% lower throughput at the worst-case. However, it achieves up to 16% higher throughput for the other three queues. The improved performance of the proposed algorithm comparing to *ILP-SMRA* can be explained by the allocation method we follow. The proposed method utilizes profiling information to partition SMs in a way that favors applications with high IPC. In contrast, *ILP-SMRA* starts by partitioning equally SMs and re-adjusts them at run-time. We argue that precious time and throughput can be lost until the re-allocation algorithm decides to transfer SMs. Additionally, to transfer a SM, all currently running threads must finish their task. This limitation delays scheduling of future threads, thus under-utilizing SMs. The proposed method outperforms the *Performance/Aging aware* method for four out of the five queues. This is expected as the kernel level optimization of the proposed method yields improved usage of SMs.

Figure 5.4 presents information about IPC per application used in the experiments. The upper sub-figure shows the normalized IPC. We first collect the IPC per application for all the pairs that the application participated. Then, we calculate the average application IPC and normalize it by the ideal IPC for that application. We notice that for certain applications, e.g., BFS2 and LUD, the proposed methodology achieves lower IPC than the ideal configuration and the compared methods. As it has already been mentioned, the presented methodology favors applications with high inherent IPC during the SM allocation. As a consequence, applications that achieve low ideal IPC are disadvantaged during allocation. Thus, these applications demonstrate lower IPC than with the compared methodologies. Absolute values of application IPC are illustrated in the lower sub-figure of Figure 5.4. We notice that, with the exception of BP and LPS, for all the applications with ideal IPC higher than 500, the proposed methodology achieves higher average IPC than the compared methodologies. From the lower sub-figure, we can also notice the effect of contention caused by concurrent application execution. There is no application for which any methodology achieves equal or greater IPC than the ideal.

Figure 5.5 depicts the relative delay caused on SMs, projected in a period of 3 years. To calculate the relative delay, we utilized the average activity factor per SM, for the 50 PV maps, after the execution of a whole queue. The bold lines of each sub-figure correspond to the relative delay caused by the average activity factor among the 60 SMs $\pm$*standard deviation*. In other words, each sub-figure demonstrates the span of relative delay among the SMs of a GPU. The smaller the colored area, the more uniformly distributed the aging is among SMs. Observing Figure 5.5 we can see that the proposed methodology demonstrates significantly more balanced aging, compared to *FCFS* and *ILP-SMRA* methods. This is expected as the latter two methods do not consider equalizing aging distribution during execution. Thus, the SMs of the GPU can demonstrate high divergence in terms of aging. On the best case, the proposed methodology achieves 36$\times$ lower standard deviation of relative delay than the *FCFS*

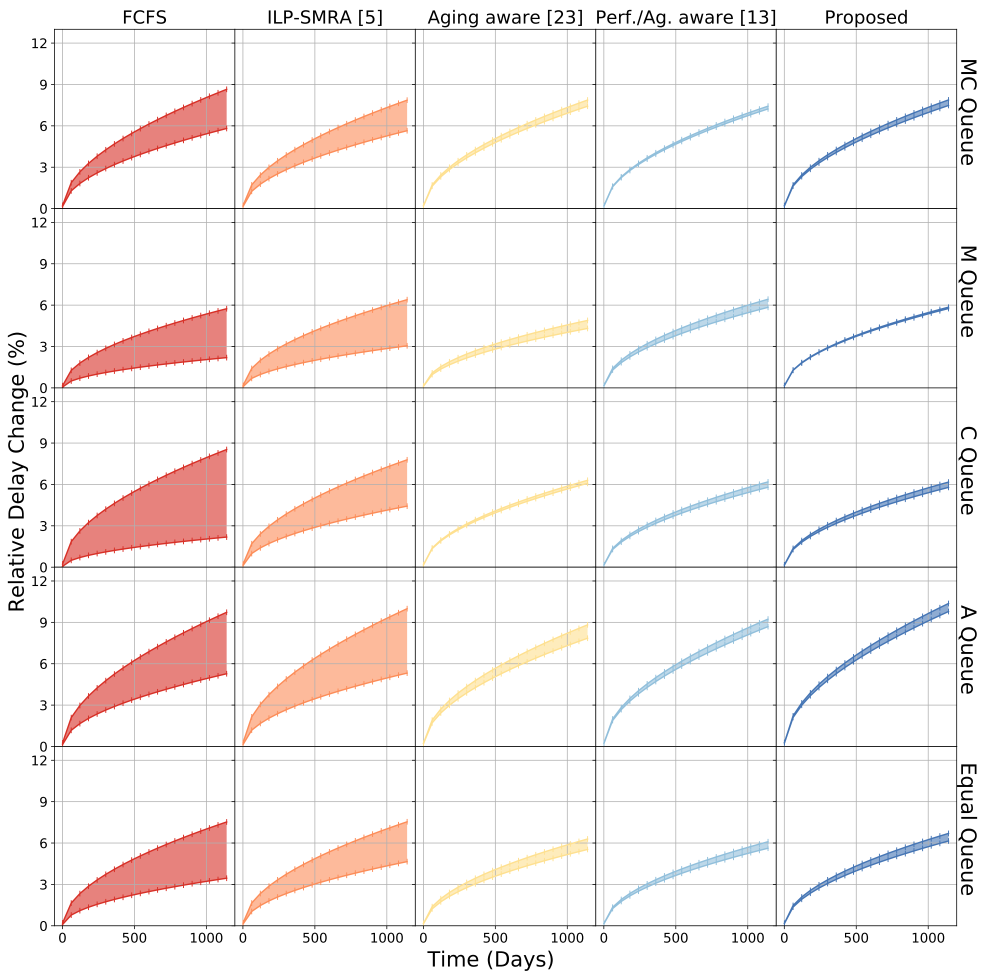


Figure 5.5: Relative delay projection for a period of three years [88]

method (M-queue) and 34× lower standard deviation of relative delay than the *ILP-SMRA* method (M-queue). Comparing to the *Aging aware* method, the proposed methodology demonstrates 1.74× higher standard deviation of relative delay for the C-queue, but for the M-queue the proposed methodology achieves 5.9× lower standard deviation. This happens as the proposed methodology reorders SMs at a finer-level and dynamically clock-gates SMs at kernel level in order to decrease aging divergence. Even though the proposed method demonstrates higher worst-case aging than the *Aging aware* method for three queues, it generally achieves lower aging divergence. Higher worst-case aging is a result of higher performance that the proposed method achieves compared to

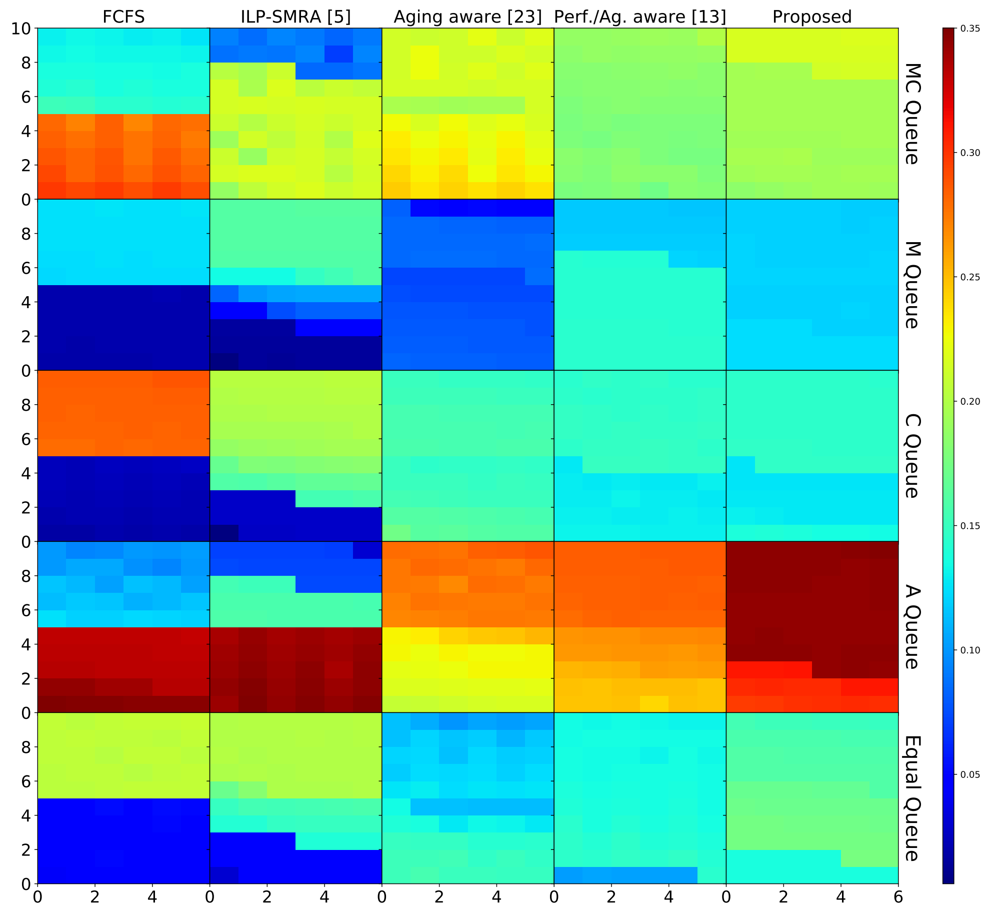


Figure 5.6: Activity factor per SM for each method and each queue [88]

the *Aging aware* method, as high activity factors are coupled with high performance. Finally, compared to the *Performance/Aging aware* method, the proposed method achieves lower relative delay divergence for three out of the five queues. For the two queues that the proposed method demonstrates higher divergence, it does not exceed 50% than the relative delay divergence of the former method.

Figure 5.6 depicts the activity factor of all the SMs on the GPU, and we can observe the activity divergence of SMs. The proposed methodology distributes activity factor in a more balanced way among SMs, comparing to the other methodologies. Balanced activity for the SMs will lead to balanced aging for the GPU, as duty cycle and activity factor are parameters of the aging model, Equations 4.2 and 4.7. The proposed methodology achieves balanced activity by ordering and allocating SMs both at the application and at the kernel level. We also observe that in certain cases, e.g., A queue, the SMs at the proposed methodology demonstrate higher average activity factor. This is a result of the high throughput that the proposed methodology achieves for these cases.

## 5.4 SUMMARIZING

In this chapter we presented an SM allocation methodology for applications that execute concurrently on GPUs. The methodology intents to improve system throughput while balancing aging among the SMs. The experiments we conducted demonstrate that the methodology achieves its goals by improving performance and keeping aging balanced among the SMs. Compared to other SM allocation policies, the developed policy achieves higher throughput and decreases aging divergence for the SMs while it also takes into consideration PV on the GPU.

# CHAPTER 6
# WEIGHT-ORIENTED APPROXIMATION FOR ENERGY-EFFICIENT NEURAL NETWORK INFERENCE ACCELERATORS

In this chapter we present a time-efficient methodology that maps NN weights to approximation levels, during inference. Provided a hardware accelerator that consists of approximate multipliers, the presented weight-oriented methodology decides which approximation level to use, in order to achieve higher energy gains, compared to an inference from exact hardware.

## 6.1 PRELIMINARIES

The error tolerant nature of NN inference presents a potential candidate for approximate computation. Specifically, the use of approximate multipliers allows energy savings as a trade-off of accuracy. Accuracy during NN inference is highly input dependent and, as NNs become deeper, the error induced by approximate multiplications has more impact on the inference accuracy. Particularly, for deep NNs static approximate multipliers fail to meet tight accuracy constraints. To this end, a methodology that efficiently utilizes approximate multipliers while keeping inference accuracy high, can significantly improve the energy efficiency of deep NNs.

## 6.2 METHODOLOGY

Figure 6.1 depicts an overview of our proposed methodology. Given a NN-oriented approximate multiplier design, we perform a weight-to-approximation mode mapping (Section 6.2.1). Specifically, we consider an inference accelerator similar to Google TPU [89], that employs a systolic MAC array and we replace the exact multipliers with the approximate multiplier. However, note that our methodology is not bound to a specific accelerator architecture. Already trained NNs are quantized to 8-bit fixed point

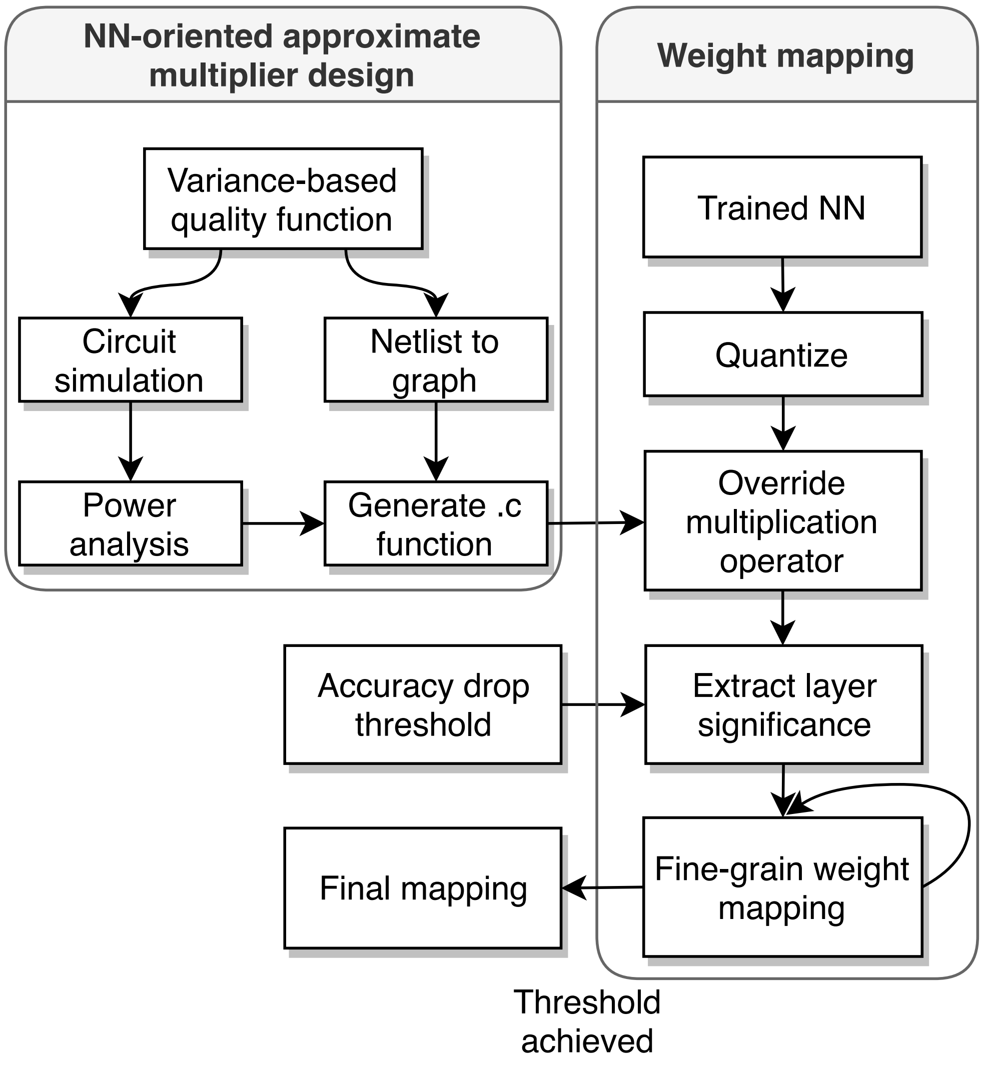

NN-oriented approximate multiplier design
Variance-based quality function
Circuit simulation
Netlist to graph
Power analysis
Generate .c function
Weight mapping
Trained NN
Quantize
Override multiplication operator
Accuracy drop threshold
Extract layer significance
Fine-grain weight mapping
Final mapping
Threshold achieved

Figure 6.1: Overview of the proposed methodology [90]

(both weights and activations) to enable their execution on the considered accelerator. Then, we override the exact multiplication with a `C`-description of the approximate multiplier and we extract the significance of each layer. Based on an accuracy drop threshold we perform a fine-grain weight mapping, rather than layer-based, in order to extract the final run-time configurations.

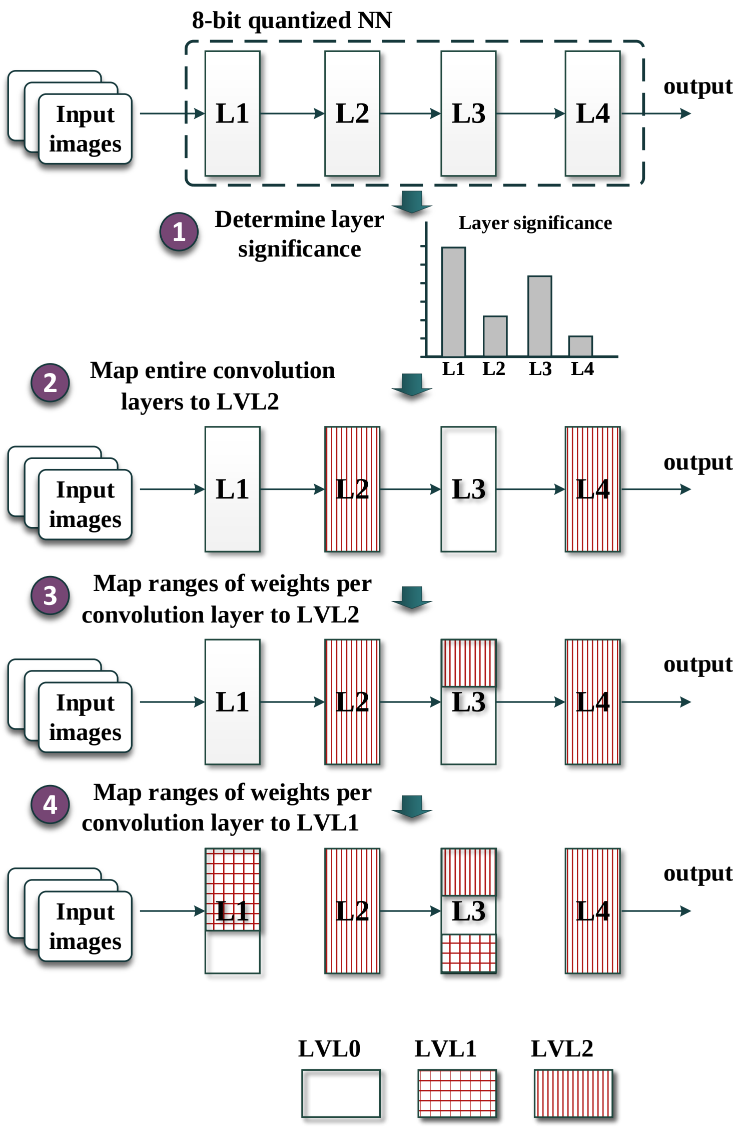
Specifically, the proposed methodology considers an approximate multiplier with three accuracy levels. The multiplier uses a 2-bit input control signal to select the desired accuracy level. By using different control signals the multiplier can achieve multiple varying accuracy levels [55].

### 6.2.1 Weight-Oriented Mapping

The proposed methodology focuses on mapping the different approximation modes based on the weight values of the NN. Specifically, given an accuracy drop threshold, the methodology decides which approximation mode will be used for each weight value, for each layer of the NN. The mapping is such that the final accuracy of the NN during inference satisfies the error threshold, and the energy consumption is minimized.

This mapping problem is very challenging due to its high complexity. Modern NNs employ tens to hundreds convolutional layers consisting of thousands to millions of different weight values. Taking also into consideration the different approximate modes of the given multiplier, an exhaustive exploration is infeasible. In an attempt to reduce the search time and space, previous approaches [13] utilize evolutionary algorithms and perform a layer-oriented mapping. However, such solutions try to solve the problem in a stochastic way being very time consuming in order to satisfy a specific accuracy threshold.

In order to find an efficient weight-to-approximate mode mapping and reduce the number of evaluated solutions, we employ a four-step methodology based on the concepts of layer significance and weight magnitude [91, 92]. The overview of the methodology can be seen in Figure 6.2. The significance of a layer is determined based on how much accuracy drop we have during inference if all the multiplications of that layer were executed with most aggressive approximate mode. The idea behind weight magnitude is that weights with small absolute value contribute little to the final result [93]. Thus, they can tolerate more error and can be mapped to the mode with the highest approximation. This concept has also been used in weight pruning where any value less than a threshold is set to zero. The four steps of the methodology are presented in detail in the following subsections. Additionally, Steps 1 and 2 are also presented in Algorithm 2 and Steps 3 and 4 in Algorithm 3. Before we present the methodology in detail we clarify that `LVL0`



8-bit quantized NN
Input images
L1
L2
L3
L4
output
1 Determine layer significance
Layer significance
L1 L2 L3 L4
2 Map entire convolution layers to LVL2
3 Map ranges of weights per convolution layer to LVL2
4 Map ranges of weights per convolution layer to LVL1
LVL0
LVL1
LVL2

Figure 6.2: Weight-oriented mapping of approximation modes

represents the exact mode of the multiplier, calculations with this mode yield exact results. `LVL1` represents an intermediate level of approximation, there are energy gains when using this mode, together with a small accuracy loss. Finally, `LVL2` represents the most aggressive level of approximation. The energy gains in this mode are the greatest for the multiplier. As a result though, there are high inaccuracies in the result, due to the high approximation level.

**Step 1 - Determine layer significance:** The focus of this step is to extract and

**Algorithm 2** Significance extraction and coarse mapping

```
1: function LAYER SIGNIFICANCE(NN, dSet, convLayers, LVL0, LVL2)
2:     set all $convLayers$ to $LVL0$
3:     $(exactAccuracy, multNumberPerLayer) \leftarrow$ EXECUTE$(NN, dSet)$
4:     $i \leftarrow 0$
5:     for $layer$ in $convLayers$ do
6:         set $layer$ to $LVL2$
7:         $accuracy \leftarrow$ EXECUTE$(NN, dSet)$
8:         $significanceList[i] \leftarrow (layer, (exactAccuracy - accuracy) \div exactAccuracy)$
9:         set $layer$ to $LVL0$
10:        $i = i + 1$
11:    sort $significanceList$
12:    return $significanceList$
13:
14: function APPROXIMATE LAYER MAPPING(NN, dSet, significanceList, LVL0, LVL2, threshold)
15:    for $layer, \_$ in $significanceList$ do
16:        set $layer$ to $LVL2$
17:        $accuracy \leftarrow$ EXECUTEWBIAS$(NN, dSet)$
18:        if $accuracy \geq threshold$ then
19:            $approximateLayer \leftarrow layer$
20:        else
21:            set $layer$ to $LVL0$
22:            break
       return $approximateLayer$
```

store the significance of each convolution layer. Initially, we map all weights of all the convolution layers to `LVL0`. The NN is executed and the accuracy is recorded along with the number of multiplications performed in each convolution layer. The accuracy of the network, using `LVL0` for all the layers, is necessary in order to calculate the layer significance. Additionally, the number of multiplications per layer is useful in cases where the significance of multiple layers is the same. Moving forward, we map the weights of each convolutional layer separately, one at a time, to `LVL2`, which is the most aggressive approximate mode and yields higher energy gains. We record the accuracy achieved while a whole layer ($L$) was approximated and we calculate the significance ($S$) of this layer, using the metric

$$S_L = \frac{ACC_{all\ layers \rightarrow LVL0} - ACC_{L \rightarrow LVL2}}{ACC_{all\ layers \rightarrow LVL0}} \tag{6.1}$$

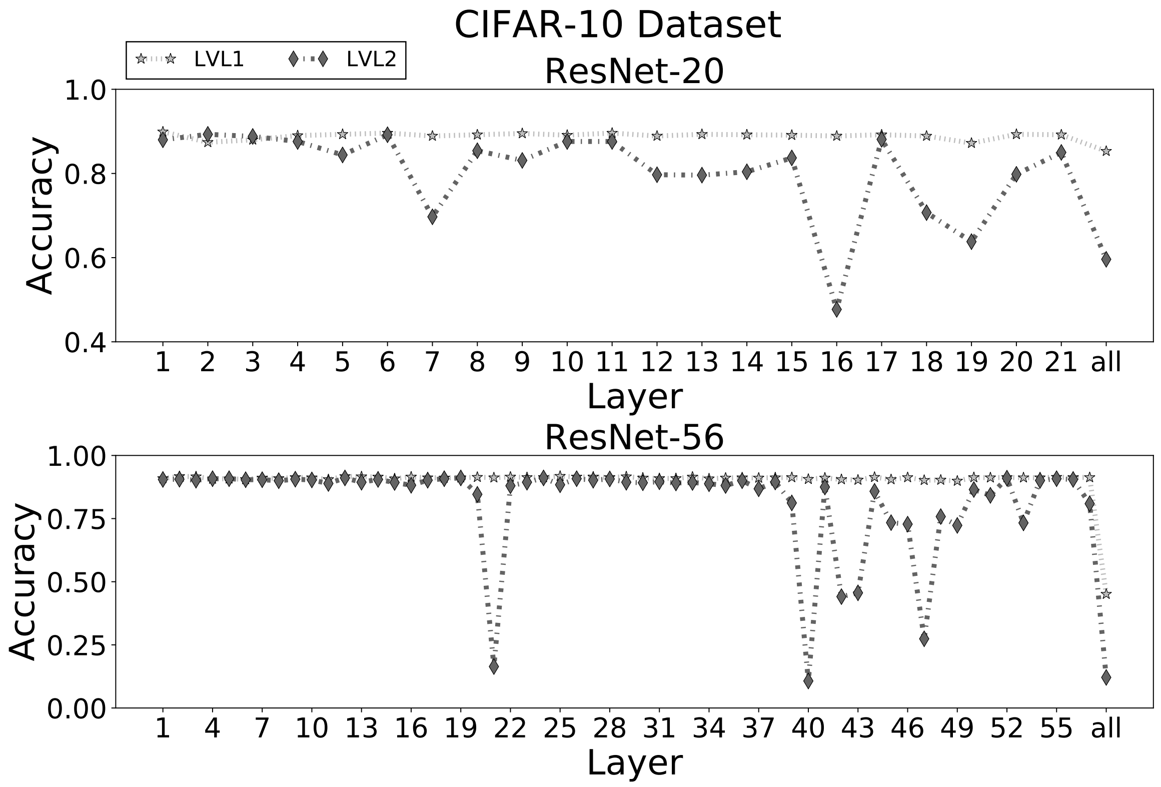


Figure 6.3: Impact of whole layer approximation on accuracy

Layers with low significance value are not considered important as they do not affect the accuracy of the NN. When the significance of all the convolution layers is extracted, we sort them based on the calculated values in an ascending order. In case multiple layers demonstrate the same significance value, the parity is broken by considering the number of multiplications in the layer. Layers with fewer multiplications are considered more significant. As an example, Figure 6.3 shows the accuracy of each separate convolutional layer for ResNest-20 [94] and ResNet-56 [94] NNs, under the dataset CIFAR-10 [95], while mapped under `LVL1` and `LVL2` approximate modes. We can see that some layers are more significant than others, e.g., layer 7 of ResNet-20 and layer 21 of ResNet-56, remarkably affecting the accuracy of the NN. The last point in x-axis corresponds to the case where all layers are approximated.

**Step 2 - Map entire convolution layers to `LVL2`:** This step aims at mapping multiple layers at the same time to `LVL2`. The reasoning behind this design choice is that layers of lower significance can potentially be configured entirely to the most approximate mode, thus yielding high energy gain without impeding high accuracy. Starting from the least significant layer, we map the next most significant one from our list to `LVL2`. It is important to mention that at this point we update the biases of the filters in order to compensate the error induced by the employed approximate multiplications. After recording the achieved accuracy during inference, we check whether the current configuration satisfies the required threshold. If the threshold is met, the current layer is saved as the last layer that can be entirely mapped to `LVL2`. In case that for a convolution layer, the achieved accuracy fails to meet the required threshold, we stop the layer search since it is expected that by adding more layers to the approximate configuration, accuracy will only be reduced. At the end of this step, we have extracted the most significant layer up to which we can configure entire layers to `LVL2`, while satisfying the required threshold.

**Step 3 - Map ranges of weights per convolution layer to `LVL2`:** The goal of this step is to determine how weights per layer will be mapped to the various modes of the approximate multiplier, for the layers that have not been entirely mapped to `LVL2` in the previous step. Specifically, we determine which ranges of weights will be mapped to the `LVL2` and which will mapped to `LVL0`. The range approach is an important aspect of the proposed methodology. The intuition behind mapping specific ranges of weights per layer to be multiplied approximately derives from the weight pruning based on magnitude [91, 92]. Weight pruning based on magnitude relies on the concept of removing neurons with weights of small magnitude, close to the value zero. The pruned NN results in more compact representation without sacrificing significant levels of accuracy. The developed approach takes advantage of the fact that certain weights do not affect the overall accuracy, even if they are removed. Thus, we approximate the multiplication of weights with small magnitude, close to zero, in an attempt to achieve energy gains. In that way, even though approximate multiplications will insert error to the calculations,

**Algorithm 3** Approximate Weight Mapping

```
1: function INITIAL WEIGHT MAPPING(NN, dSet, approximateLayer, significanceList,
   LVL0, threshold)
2:     $totLayers = len(significanceList)$
3:     $i \leftarrow 0$
4:     $layer = significanceList[i][0]$
5:     while $layer \neq approximateLayer$ do
6:         $i = i + 1$
7:         $layer = significanceList[i][0]$
8:     $j = i + 1$
9:     while $j \leq totLayers$ do
10:        $configs[j] =$ DETERMINE CONFIG($NN, dSet,$
                              $significanceList, LVL0, threshold, j,$
                              $range1, range2, range3$)
11:        if $configs[j]$ is $NULL$ then
12:            break
13:        $j = j + 1$
       return $configs$
14:
15: function FINE WEIGHT MAPPING(NN, dSet, approximateLayer, significanceList,
    LVL0, threshold, weightConfigs)
16:    $totLayers = len(significanceList)$
17:    $i \leftarrow 0$
18:    $layer = significanceList[i][0]$
19:    while $layer \neq approximateLayer$ do
20:        $i = i + 1$
21:        $layer = significanceList[i][0]$
22:    $j = i + 1$
23:    while $weightConfigs[significanceList[j][0]]$ not NULL do
24:        $j = j + 1$
25:    $k = j$
26:    while $k \leq totLayers$ do
27:        $configs[k] =$ DETERMINE CONFIG($NN, dSet,$
                              $significanceList, LVL0, threshold, k,$
                              $range4, range5, range6$)
28:        if $configs[k]$ is $NULL$ then
29:            break
30:        $k = k + 1$
       return $configs$
```

the overall accuracy will not be significantly impacted. Additionally, if we map a weight to an approximate mode, the more it appears in a layer, the higher the probability of deteriorating overall accuracy of the NN. The ranges we use depend on the introduced

```
31:
32: function DETERMINE CONFIG(NN, dSet, significanceList, LVL0, thr, lr, rangeA,
    rangeB, rangeC)
33:     set $layer = significanceList[lr][0]$ to $rangeA$
34:     $accuracy \leftarrow$ EXECUTEWBIAS$(NN, dSet)$
35:     if $accuracy < thr$ then
36:         set $layer = significanceList[lr][0]$ to $rangeB$
37:         $accuracy \leftarrow$ EXECUTEWBIAS$(NN, dSet)$
38:         if $accuracy < thr$ then
39:             set $layer = significanceList[lr][0]$ to $rangeC$
40:             $accuracy \leftarrow$ EXECUTEWBIAS$(NN, dSet)$
41:             if $accuracy < thr$ then
42:                 set $layer = significanceList[lr][0]$                to $LVL0$
                 return $NULL$
             return $rangeC$
         return $rangeB$
     return $rangeA$
```

error of `LVL1` and `LVL2`. For the `LVL2`, the range of weights is more conservative, compared to `LVL1`, due to the higher error. After the 8-bit quantization and based on the maximum accuracy drop threshold that we set for our experiments/inference, we experimentally derived the weight value ranges to map to `LVL2` as $range3 = \mathbf{0} \pm 10$[1], $range2 = \mathbf{0} \pm 5$ and $range1 = \mathbf{0}$. We start exploring configurations using a wider weight range, $range3$, and we gradually move to more narrow ranges, $range1$, until the accuracy threshold is met. Using `LVL2` on weights outside $range3$ had a strong effect on the accuracy and for that reason they were omitted. Once the configurations that satisfy the accuracy threshold have been found, the weight mapping is performed and the biases are updated for the respective mapping. The bias correction has to be performed every time we update the weight mapping in a particular filter.

**Step 4 - Map ranges of weights per convolution layer to `LVL1`:** The goal of this final step is to find which of the remaining weights, that are still assigned to `LVL0`, can be mapped to `LVL1`. Similarly to the previous step, we create ranges of weight values.

[1]Initially, the weights had float values in the range of $[-1, 1]$. Thus, the value of $\mathbf{0}$ depends on the applied quantization. For example, for 8-bit quantization in $[-128, 127]$, $\mathbf{0} = 0_{10}$, while for quantization in $[0, 255]$, $\mathbf{0} = 128_{10}$.

Since `LVL1` introduces smaller error than `LVL2`, the NN can tolerate more weights to be approximated per layer. Thus, the range of the weight values that we search in this step is greater. Specifically, given the maximum accuracy drop threshold set, we have two additional ranges for `LVL1` $range5 = \mathbf{0} \pm 30$ and $range4 = \mathbf{0} \pm 20$. We start exploring configurations using a wide range ($range5$), and we gradually move to more narrow ranges until the accuracy threshold is met. Although the ranges in Steps 3 and 4 are overlapping, if a weight is mapped in Step 3, then it is not considered in Step 4. Again, each time a weight mapping is performed we update the biases.

Our experimental analysis showed that, for all the examined NNs, the weights' values are distributed around **0**. Hence, our range-based approach enables identifying a large number of weights to be assigned to an approximate mode and thus, boosts the energy savings. Nevertheless, by just increasing the size of the examined ranges we can also cover cases that the weights are not concentrated around **0**. Finally, note that our proposed framework (steps 1-4) needs to be executed only once at design time. After Step 4, for each weight at each layer the corresponding accuracy level of the approximate reconfigurable multiplier (e.g., `LVL0`, `LVL1`, or `LVL2`) is extracted and the biases are updated. Then, during run-time inference, the extracted accuracy level is selected for each approximate multiplier of the NN accelerator.

## 6.3 EVALUATION

In order to test the effectiveness of the proposed fine-grain weight mapping, we evaluated our framework on the ResNet-20, ResNet-32, ResNet-44, ResNet-56 [94], and MobilNet-v2 [96] neural networks. For the evaluation, we utilized four datasets CIFAR-10 [95], CIFAR-100 [95], GTSRB [97], and LISA [98]. In total, our framework is evaluated against 20 different models. Initially, all NNs were trained on the aforementioned datasets using the Tensorflow machine learning library. Then, the NNs were frozen and quantized to 8-bit. In order to perform the weight mapping, we overrode the convolution layers of the quantized networks and replaced the exact multiplication function with software descriptions to emulate the results of an approximate multiplier. Figures 6.4-6.8 depict the results of our experiments in terms of energy savings for the multiplication operations and achieved accuracy during inference. The methods used for the quantitative evaluation of the presented framework are:

1) Accurate: This method is the baseline of our experiments, utilizing exact multipliers in all layers;

2) ALWANN [13]: This method utilizes the fixed approximate multipliers of the state-of-the-art library EvoApproxLib [54] and applies weight-tuning to minimize the Mean Absolute Error incurred by using approximate multipliers;

3) RETSINA [55]: This method follows a layer-oriented weight mapping (i.e., the multiplications of each convolution layer are performed at the same accuracy level but different layers might use different accuracy level) utilizing MRE-based approximate reconfigurable multipliers with three modes: exact, 0.5% MRE, and 1.5% MRE;

4) Proposed layer-wise w/ Bias: This method utilizes an approximate multiplier, namely LVRM, with three approximation modes, presented in [90], and applies error correction through bias update. However it follows a layer-oriented weight mapping;

5) Proposed fine weight mapping w/o bias: This method utilizes the LVRM multiplier and the proposed fine-grain weight mapping (Section 6.2.1), without bias update;

6) Proposed fine weight mapping w/ bias: This is our proposed method that utilizes the LVRM multiplier, fine-grain weight mapping, and bias update.

At this point it is important to mention that, in our framework, we selected four values for the accuracy drop threshold, 0.5%, 0.7%, 1.0%, and 2.0%. Additionally, in Figures 6.4-6.8 all configurations of (2)-(4) that resulted in accuracy loss up to 3% are also depicted. Accuracy loss of more than 3% is not acceptable based on our threshold values. Note that, since different layers feature different significance, in ALWANN [13] a heterogeneous architecture is proposed that comprises several fixed approximate multiplier types of [54]. In order, to achieve reconfiguration at run-time, each convolution layer uses a specific multiplier type and the rest are power-gated. However, this approach leads to highly increased area and to high loss in throughput due to the underutilized, power-gated hardware. Moreover, to achieve high accuracy a different architecture is generated for each considered NN, i.e., approximate multiplier of multiple types are selected. On the other hand, in this work, we target a generic homogeneous inference accelerator, similar to [89] and [5]. This enables us to achieve high throughput and to be NN independent, i.e., any convolutional NN can be executed on the proposed approximate reconfigurable accelerator. Therefore, for the fairness of the evaluation, we consider a homogeneous architecture for ALWANN [13] and thus, the same approximate multiplier type is used in all the convolution layers. Nevertheless, we consider all the 32 Pareto optimal approximate multipliers of [54]. In other words every design point in Figures 6.4-6.8 regarding ALWANN uses a different approximate multiplier from [54]. All the examined approximate multipliers are synthesized at the critical path delay of the exact multiplier [54] and thus, all the reported energy gains originate from the achieved power reduction.

Figure 6.4 compares different configurations of the ResNet-20 NN for the four selected datasets. As an overall observation, we see that the proposed methodology always produces configurations with the highest energy gain, within the examined accuracy loss margin. Specifically, for the CIFAR-10 dataset, the accurate configuration achieves 89.1% accuracy. The proposed framework achieves up to 17.5% energy gain for an accuracy loss of 1.7%. The configuration with closest energy gain, for the 2.0% accuracy loss margin, comes from the "Proposed layer-wise w/ Bias" method with 14.7% energy gain for an accuracy loss of 1.3%. Regarding the CIFAR-100 dataset, the proposed framework

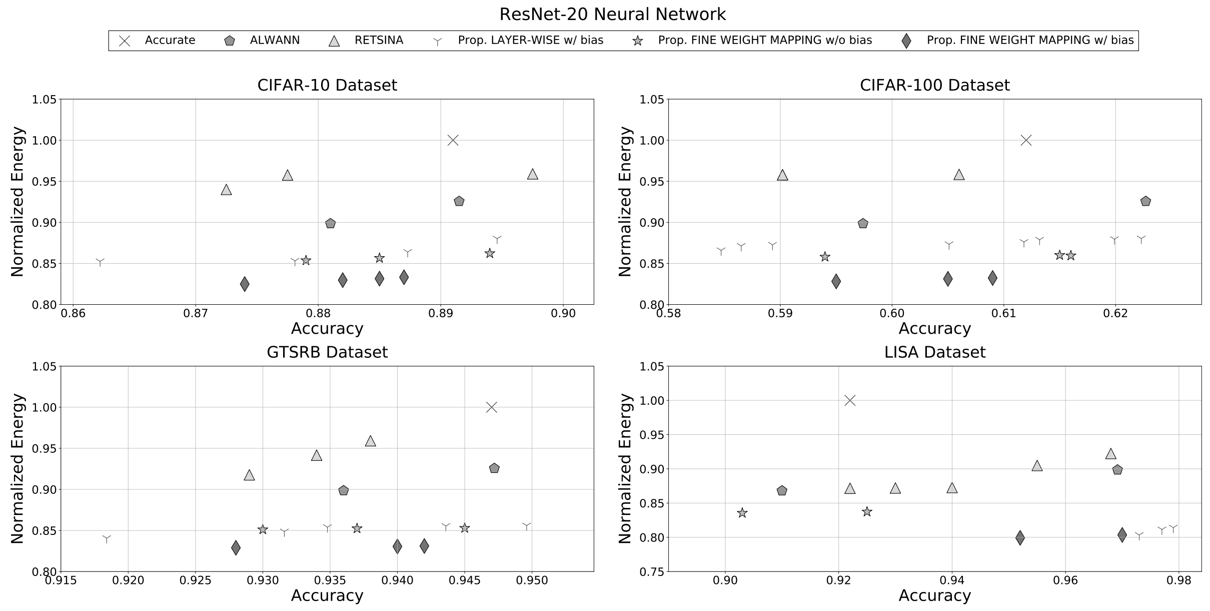


Figure 6.4: Accuracy and energy savings of ResNet-20 under different methods that utilize approximate multipliers

produces configurations with up to 17.2% energy gain with an accuracy loss of 1.7%. Comparing methodologies ALWANN and RETSINA, they do not produce configurations with greater energy gains than 10.2% and 4.2% respectively. For the GTSRB dataset, the proposed framework produces configurations with 16.9% energy gain for an accuracy loss of 0.5%, while in the same region of accuracy loss. The next best configurations considering energy gains, for the GTSRB dataset, are produced by the “Proposed fine weight mapping w/o bias” and the “Proposed layer-wise w/ Bias” methods, with energy reduction 14.7% and 14.5% respectively. Again we observe that the proposed methodology produces configurations with the highest energy gain, respecting the accuracy thresholds that have been set. For the LISA dataset, the proposed methodology achieves up to 20.1% energy gains. ALWANN, RETSINA, and the “Proposed fine weight mapping w/o bias” cannot produce configurations that achieve more than 16.5% energy gain, even by dropping accuracy lower than the proposed methodology. The “Proposed layer-wise w/ Bias” performs very close to the proposed methodology, however achieving 1.1% *lower* energy gain. In addition, note that sometimes using approximate multipliers

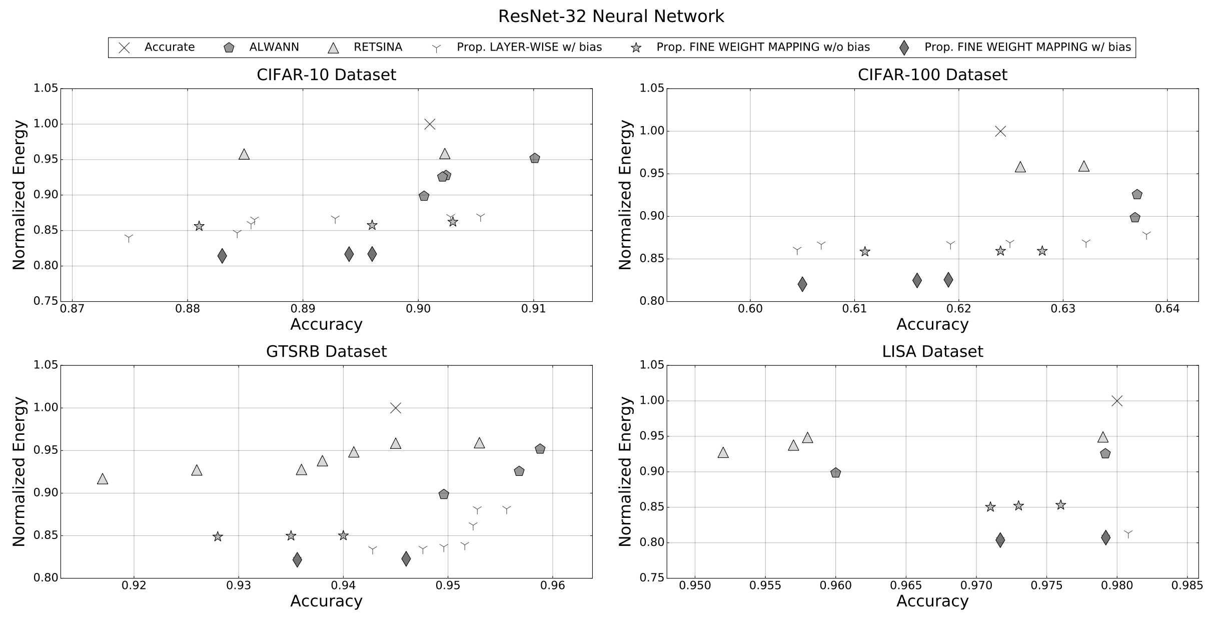


Figure 6.5: Accuracy and energy savings of ResNet-32 under different methods that utilize approximate multipliers

yields higher accuracy than the accurate method [99, 13].

Figure 6.5 depicts the comparison of the different methods for the ResNet-32 NN. For this network as well, we can see that the proposed approach finds solutions that achieve the highest energy gains, while respecting the up to 2.0% accuracy drop thresholds. Particularly, for the CIFAR-10 dataset, the proposed framework achieves up to 18.6% energy gain for an accuracy loss of 1.8%. The configuration with the closest energy gain is again the "Proposed layer-wise w/ Bias" with 15.3% energy gain for an accuracy loss of 1.7%. Regarding the CIFAR-100 dataset, the proposed method produces configurations that reduce energy consumption by 18.0% with an accuracy loss of 1.9%. ALWANN and RETSINA do not produce configurations with greater energy gains than 10.2% and 4.2% respectively. For the GTSRB and LISA datasets, the behavior is similar as before, the proposed methodology achieves the best trade-offs between energy consumption and accuracy.

Figures 6.6-6.7 depict the comparison of the different methods for the remaining two ResNets, ResNet-44 and ResNet-56 respectively. Similarly, the proposed approach finds

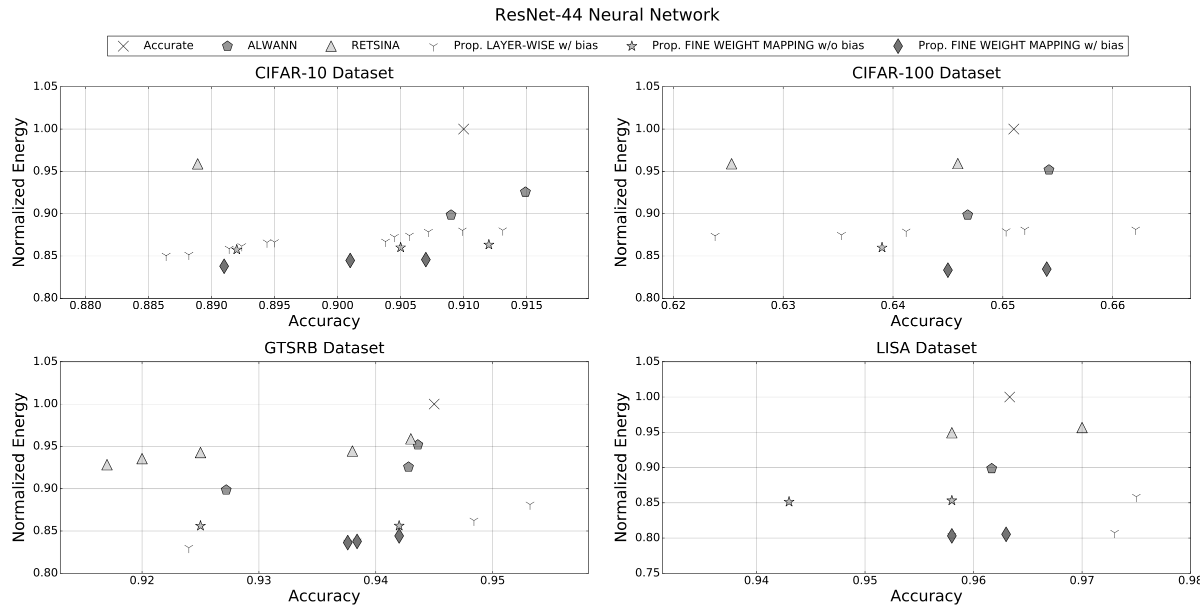


Figure 6.6: Accuracy and energy savings of ResNet-44 under different methods that utilize approximate multipliers

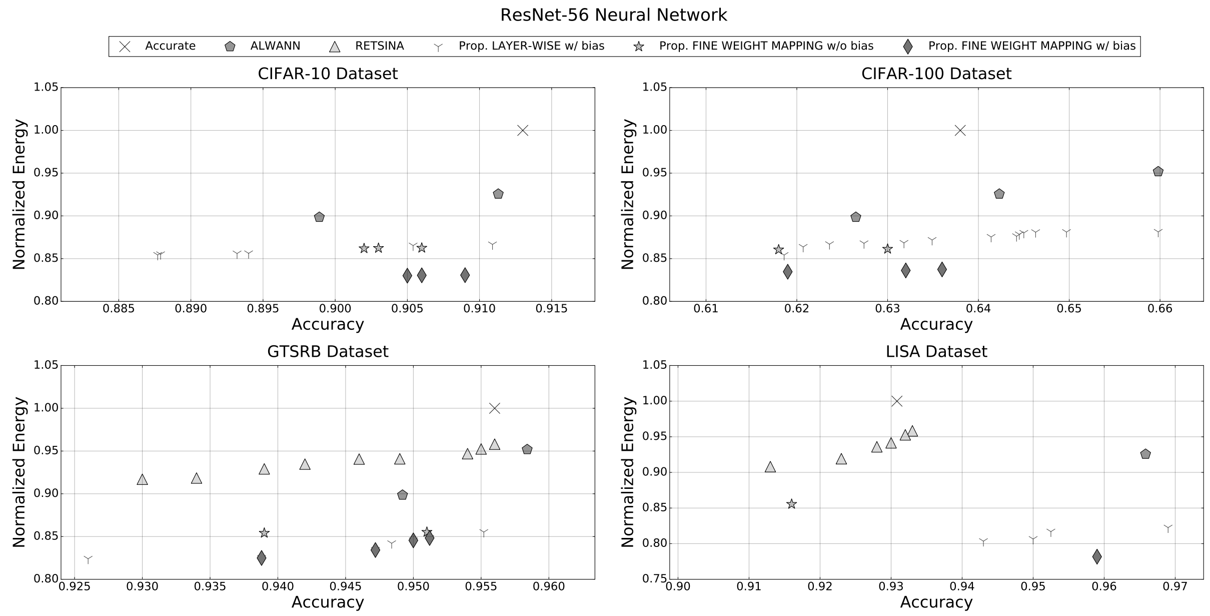


Figure 6.7: Accuracy and energy savings of ResNet-56 under different methods that utilize approximate multipliers

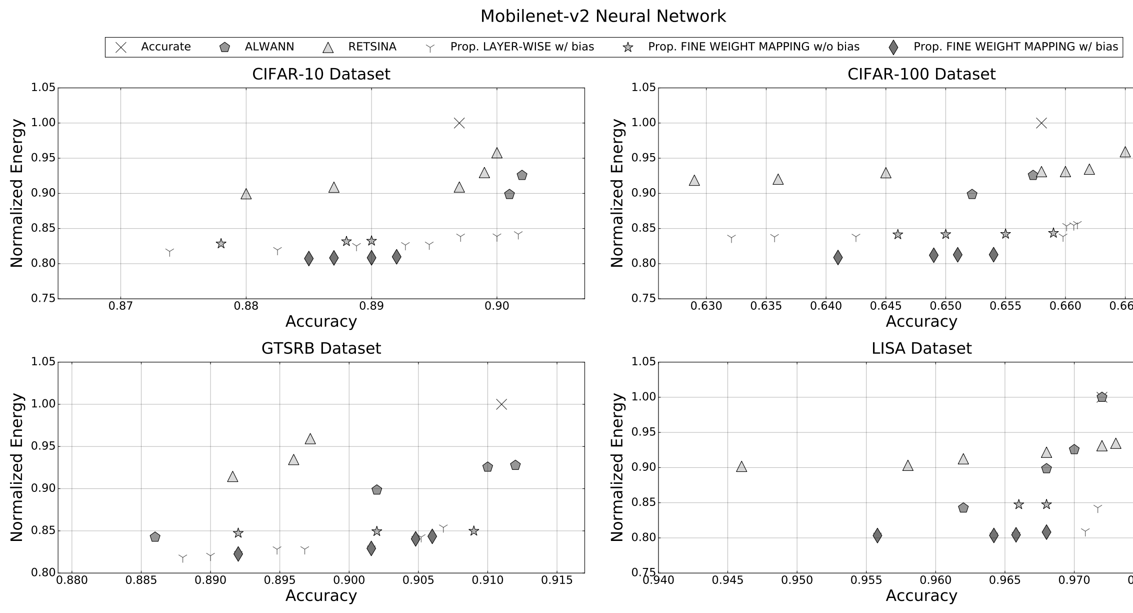


Figure 6.8: Accuracy and energy savings of MobileNet-v2 under different methods that utilize approximate multipliers

solutions that achieve the highest energy gains, while respecting the accuracy drop thresholds. Particularly, for the ResNet-44 NN the proposed framework achieves up to 16.2%, 16.7%, 16.4%, and 19.7% energy gains for the four selected datasets, while respecting the accuracy loss margin of 2.0%. For the ResNet-56, the respective gains are 17.0%, 16.5%, 17.5%, and 21.8%.

Finally, Figure 6.8 depicts the results for the MobileNet-v2 NN, which is designed for next generation mobile devices and for that reason reducing energy consumption is very important. For the CIFAR-10 dataset, the proposed framework achieves up to 19.3% energy gain for an accuracy loss of 1.2%. It is noteworthy that, for this dataset, all methods that utilize the LVRM approximate multiplier have increased energy savings. For the CIFAR-100 dataset, the proposed framework produces configurations with up to 19.1% energy gain with an accuracy loss of 1.7%. Regarding the GTSRB dataset, we can see that the "Proposed layer-wise w/ Bias" method has the best energy consumption. However, that comes with the cost of dropping accuracy more than 2.0%, i.e., the threshold we set to our framework. Finally, ALWANN, RETSINA, and the "Proposed

Table 6.1: Average energy gain comparison

| Method | Average energy gain | |
|---|---|---|
| | 0.5% thr. | 2.0% thr. |
| ALWANN [13] | 8.5% | 10.4% |
| RETSINA [55] | 4.0% | 5.9% |
| Proposed Layer-Wise w/ Bias | 15.4% | 16.5% |
| Proposed Fine Weight Mapping w/o Bias | 10.3% | 14.8% |
| Proposed Fine Weight Mapping w/ Bias | 17.6% | 18.2% |

fine weight mapping w/o bias" cannot produce configurations that achieve more than 15.7% energy gain, even by dropping the accuracy lower than the proposed methodology. The layer-wise exploration can produce slightly more accurate configurations for the LISA dataset, achieving 0.3% higher accuracy but consuming 0.1% more energy than the most accurate configuration of the proposed methodology.

As an overall comparison of the five methods, we provide Table 6.1. This table depicts the average energy gain for each method, across all the examined NNs on all the datasets. We provide the average energy gain for the configurations that achieved the 0.5% accuracy error threshold, as well as the average energy gain per methodology for the configurations that achieved the 2.0% accuracy error threshold. Overall, based on the performed evaluations we reach the following conclusions:

1) Even though both RETSINA and "Proposed layer-wise w/ Bias" are layer-oriented methods, the latter delivers better solutions due to the utilization of the LVRM multiplier and the error correction through the bias update;

2) the proposed method ("Proposed fine weight mapping w/ Bias") is better than "Proposed layer-wise w/ Bias" as the layer-oriented approaches are very coarse grain and can miss optimal solutions;

3) the proposed method is better than "Proposed fine weight mapping w/o Bias" as it was able to identify different configurations that satisfy the accuracy thresholds while consuming lower energy, by taking advantage of the bias correction method, thus

allowing to perform more approximate multiplications; and

4) even for deep NNs, where the effect of approximate multiplications is difficult to quantify, the proposed method achieved considerable energy gains, for example 17.2% on average for the ResNet-56 NN. The proposed fine-grained weight-oriented approach is not affected by the NN size and efficiently identifies the proper approximations by mapping weights to approximate modes. The latter is also verified by the fact that for the tight accuracy loss thresholds examined, compared to the other methods, the proposed framework delivers more consistent results as it features the highest energy gains along with the lowest energy reduction variance.

## 6.4 SUMMARIZING

In this chapter we presented a methodology that maps approximation modes of a multiplier to NN weights. During inference, approximate calculations can be used to reduce energy consumption, with a trade-off on accuracy. The presented methodology achieves higher energy gains compared to state-of-the-art methodologies for inference on approximate hardware accelerators, while it satisfies user defined accuracy loss thresholds. To satisfy tight accuracy requirements, the proposed methodology explores mappings based on weights per layer which is a finer level approach, compared to existing approaches. By focusing the space exploration on specific weight ranges, the developed framework delivers mapping configurations that achieve high energy gains without the need of excessive run-time. As a result, the presented methodology is efficient and can yield significant improvements for inferences on approximate multipliers.

# CHAPTER 7
# CONCLUSION

This dissertation is focused on improving performance while keeping track of various trade-offs, for hardware-accelerated systems. In this chapter we will highlight the characteristics of the developed methodologies, the results that we gathered from our experiments, what can be achieved by using these methodologies, and future extensions that we plan on researching.

## 7.1 MESSAGE-PASSING SYNCHRONIZATION FOR DISTRIBUTED SHARED MEMORY ARCHITECTURES

In Chapter 3 we presented a synchronization mechanism for multi-core systems that are connected via a mesh grid. Additionally, each core has a hardware accelerator attached to it. The hardware accelerator provides DSM and can be programmed using microcode. The DSM allows for data to be kept distributed thus, reducing the traffic in the mesh. The microcoding capabilities allow the programmer to implement functions that work on a lower level, bypassing the need for compilation by the core. As a result, the accelerators can be programmed to implement utilities that remove execution burden from the cores.

Synchronization is essential for multi-core systems as it guarantees data integrity and provides deterministic behavior when needed. Nevertheless, on many-core systems, an efficient synchronization mechanism is a non trivial task. As the number of cores on a chip scales up, synchronization starts posing an important bottleneck on performance. Traditional synchronization techniques do not scale well and limit performance gains on many-core systems.

With our research, we contribute a synchronization mechanism that is based on a client-server model. It leverages DSM and microcoding functionality on the accelerators to provide an efficient synchronization scheme. One of the cores plays the role of the server and the rest of the cores are the clients. The developed mechanism works for traditional data structures that demonstrate low levels of concurrency. Only the server has access to the data structure, while the clients interact with the server via message requests.

The developed mechanism was evaluated by synchronizing accesses to a stack, queue, deque and binary heap. As the experimental results demonstrate, it reduces the total execution cycles even when 22 cores are used. Thus it provides higher performance than the other synchronization models used. Additionally, the developed mechanism provides fairness, as all the requests are processed by the server in an almost serial way. Finally, the developed mechanism demonstrates valuable power gains. By minimizing the time wasted on network transfers between the cores, as well as the idle time lost by cores, and the lost time for spin-lock acquiring competition, the proposed mechanism achieves power gains combined with higher performance.

To sum up, the presented synchronization mechanism is an efficient choice for multi-core systems. It requires accelerators with DSM and microcoding capabilities attached to the cores of a system. If the developed mechanism is used, it will provide higher throughput for systems, fairness in execution and power gains. As a consequence, the developed mechanism can be adopted by embedded systems too, as the power constraint is of high importance for such systems.

## 7.2 KERNEL-BASED RESOURCE ALLOCATION FOR IMPROVING GPU THROUGHPUT WHILE MINIMIZING THE ACTIVITY DIVERGENCE OF SMS

In Chapter 4 we presented an SM allocation policy that considers the scenario of single applications executing on the GPU. The allocation policy aims at improving performance, and balancing aging among SMs. It provides a fine-grain approach by taking actions based on application metrics, as well as kernel level metrics.

The main idea behind the developed allocation methodology lies in the fact that application performance can actually improve when less SMs are available to certain applications. Exploratory experiments show that specific applications achieve higher performance when a restricted number of SMs is available to them. The latter observation allows higher performance while giving the potential to clock-gate unused SMs. As a result of clock-gating, the option of distributing activity among SMs is available. Equally distributed activity leads to balanced aging among SMs. High aging divergence is an undesired condition since aging affects the life-span of a GPU as well as its performance, heavily aged SMs function in lower frequencies.

The developed SM allocation methodology is based on application information collected off-line. Before an application can be executed, we collect information about its performance on the application level as well as at the kernel level. When the information on all the applications is collected, execution can start. The initial step of the methodology is to decide the exact SM configuration for each kernel of an application. When the number of SMs per kernel is decided, specific SMs are allocated. The latter is determined by the activity and aging condition of each SM. For kernels that do not utilize all the available SMs, the methodology undertakes the task of clock-gating the idle SMs. Finally, once the execution of a kernel is completed, the methodology is responsible for updating the activity history of the SMs. Additionally, the allocation methodology is responsible for monitoring the temperature of SMs, in order to derive accurate estimations of aging from the aging estimator module.

Experimental results on NVIDIA Fermi and Tesla micro-architectures demonstrate that the developed methodology achieves its goals. Specifically, on average it achieves higher GPU throughput than the default approach of allocating all the available SMs to each application. Furthermore, it achieves the best results of reducing aging and activity divergence among the SMs, compared with other state-of-the-art SM allocation policies.

Finally, the average power overhead is negligible, if we consider the important performance gains that the methodology achieves.

To conclude, the developed SM allocation methodology provides a promising solution. It can be used to improve GPU performance while introducing a small power overhead. Simultaneously, it contains aging divergence in small ranges, leading to reduced probability of hardware failure due to aging. The presented methodology can be used by modern GPUs in order to enhance their performance. As a results, GPUs can become a more prevalent option for execution acceleration.

## 7.3 PERFORMANCE- AND ACTIVITY-AWARE ALLOCATION FOR CONCURRENT GPU APPLICATIONS

In Chapter 5 we presented an SM allocation policy for applications executing concurrently on a GPU. The allocation policy consists of multiple steps and aims at improving system performance. In addition to performance, the allocation mechanism takes into consideration PV and the aging effects that occur during the life time of a GPU.

Until recently, GPU schedulers did not consider spatial multitasking. Even if multiple applications were scheduled to SMs, at a given moment only a single application was executing on the SMs. The rest of the scheduled applications were idle, waiting for the executing application to finish or halt. That scheduling policy leaves resources underutilized thus, limiting the improved performance that GPUs can offer.

The methodology we developed utilizes information collected about each application, as well as PV information about the GPU. This information is collected only once, at a stage before execution. During execution, the methodology decides which applications to pair together for execution, how many SMs to allocate per application and which exact SMs to allocate for each application, depending on the aging condition of the SMs.

Experimental results demonstrate that the developed methodology for concurrent applications achieves higher GPU throughput for different application queues with multiple characteristics, compared to state-of-the-art SM allocation policies. Additionally, it reduces the aging divergence of SMs, compared with other SM allocation policies.

All in all, the presented methodology can be used by GPUs that do not harness spatial multitasking. It can provide high throughput, and at the same time minimize aging divergence among SMs. This will lead to faster execution of applications that are offloaded to GPUs, while at the same time it will prolong the usage life of the GPU. By balancing aging among SMs, the probability of hardware failure is reduced and the expected lifetime of GPUs is increased.

As a future extension, we would like to investigate ways to add a fine-grain approach to the methodology. For example, after the pairs of applications are formed, SM allocation can be tuned according to the kernel needs of each application. For kernels that demonstrate low performance, application allocated SMs can be clock-gated, while for high performing kernels, all the allocated SMs can be used, to boost performance.

## 7.4 WEIGHT-ORIENTED APPROXIMATION FOR ENERGY-EFFICIENT NEURAL NETWORK INFERENCE ACCELERATORS

In Chapter 6 we presented a framework that maps NN weights to approximate modes on multipliers. The presented framework is time-efficient and platform independent, meaning that it does not depend on a specific accelerator. Additionally, it explores the use of multiple accuracy levels on the hardware accelerator. The produced mappings improve energy efficiency during NN inference while achieving accuracy comparable to inferences on exact hardware.

The increasing need for NN inference on edge devices and other resource restricted hardware creates a subsequent need for energy efficient computations. Hardware accelerators can leverage the principles of approximate computing to deliver computation at a lower energy cost. Albeit approximate computing can lower energy demands, inference accuracy can not be sacrificed. Furthermore, the complexity of modern NNs demands solutions that can scale and serve deep NN.

The proposed framework achieves the goals of energy efficiency while the user can select how much accuracy can be sacrificed, compared to inferences calculated with exact hardware. The framework applies a fine grain methodology and explores weight mappings per NN layer. By exploring ranges of the most frequently used weights, it achieves higher energy gains than other state-of-the-art methodologies, with an acceptable run time. An additional advantage of the developed framework is that the NN does not require retraining. The proposed framework was evaluated using multiple NNs, combined with multiple datasets.

The presented methodology can be used by system designers of NPUs or other hardware accelerators, targeting NN inferences. Using this methodology will yield energy gains that can allow inferences to be computed on resource constrained devices, as well as can enable the use of hardware accelerators on more domains that benefit from NN usage.

# REFERENCES

[1] T. G. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, *et al.*, "The 48-core scc processor: The programmer's view," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, IEEE Computer Society, 2010.

[2] J. Jeffers, J. Reinders, and A. Sodani, *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition.* Morgan Kaufmann, 2016.

[3] T. Baji, "Evolution of the gpu device widely used in ai and massive parallel processing," in *2018 IEEE 2nd Electron Devices Technology and Manufacturing Conference (EDTM)*, pp. 7–9, IEEE, 2018.

[4] B. van Werkhoven, "Kernel tuner: A search-optimizing gpu code auto-tuner," *Future Generation Computer Systems*, vol. 90, pp. 347–358, 2019.

[5] J. Song, Y. Cho, J.-S. Park, J.-W. Jang, S. Lee, J.-H. Song, J.-G. Lee, and I. Kang, "7.1 an 11.5 tops/w 1024-mac butterfly structure dual-core sparsity-aware neural processing unit in 8nm flagship mobile soc," in *IEEE International Solid-State Circuits Conference*, pp. 130–132, 2019.

[6] S. Cass, "Taking ai to the edge: Google's tpu now comes in a maker-friendly package," *IEEE Spectrum*, vol. 56, no. 5, pp. 16–17, 2019.

[7] Y. Cui, Y. Wang, Y. Chen, and Y. Shi, "Experience on comparison of operating systems scalability on the multi-core architecture," in *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pp. 205–215, IEEE, 2011.

[8] Z.-G. Tasoulas, R. Guss, and I. Anagnostopoulos, "Performance-based and aging-aware resource allocation for concurrent gpu applications," in *2018 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pp. 1–6, IEEE, 2018.

[9] 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.

[10] S. S. Sarwar, S. Venkataramani, A. Ankit, A. Raghunathan, and K. Roy, "Energy-efficient neural computing with approximate multipliers," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 14, no. 2, pp. 1–23, 2018.

[11] V. Mrazek, S. S. Sarwar, L. Sekanina, Z. Vasicek, and K. Roy, "Design of power-efficient approximate multipliers for approximate artificial neural networks," in *Int. Conf. on Computer-Aided Design*, pp. 1–7, 2016.

[12] M. A. Hanif, R. Hafiz, and M. Shafique, "Error resilience analysis for systematically employing approximate computing in convolutional neural networks," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 913–916, 2018.

[13] V. Mrazek, Z. Vasicek, L. Sekanina, M. A. Hanif, and M. Shafique, "Alwann: Automatic layer-wise approximation of deep neural network accelerators without retraining," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8, Nov 2019.

[14] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "Gpus and the future of parallel computing," *IEEE Micro*, vol. 31, no. 5, pp. 7–17, 2011.

[15] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich, "Non-scalable locks are dangerous," in *Proceedings of the Linux Symposium*, pp. 119–130, 2012.

[16] A. Morrison, "Scaling synchronization in multicore programs," *Queue*, vol. 14, no. 4, pp. 56–79, 2016.

[17] P. Fatourou and N. D. Kallimanis, "Revisiting the combining synchronization technique," in *ACM SIGPLAN Notices*, vol. 47, pp. 257–266, ACM, 2012.

[18] L. Papadopoulos, I. Walulya, P. Tsigas, D. Soudris, and B. Barry, "Evaluation of message passing synchronization algorithms in embedded systems," in *Embedded*

*Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV), 2014 International Conference on*, pp. 282–289, IEEE, 2014.

[19] I. Anagnostopoulos, S. Xydis, A. Bartzas, Z. Lu, D. Soudris, and A. Jantsch, "Custom microcoded dynamic memory management for distributed on-chip memory organizations," *IEEE Embedded Systems Letters*, vol. 3, no. 2, pp. 66–69, 2011.

[20] I. Koutras, I. Anagnostopoulos, A. Bartzas, and D. Soudris, "Improving dynamic memory allocation on many-core embedded systems with distributed shared memory," *IEEE Embedded Systems Letters*, vol. 8, no. 3, pp. 57–60, 2016.

[21] C. Min and Y. I. Eom, "Integrating lock-free and combining techniques for a practical and scalable fifo queue," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 7, pp. 1910–1922, 2015.

[22] S. Wu, J. Zhang, Y. Peng, H. Jin, and W. Jiang, "Optimization strategies for inter-thread synchronization overhead on numa machine," in *Computing and Communications Conference (IPCCC), 2015 IEEE 34th International Performance*, pp. 1–8, IEEE, 2015.

[23] T. Gangwani, A. Morrison, and J. Torrellas, "Caspar: breaking serialization in lock-free multicore synchronization," *ACM SIGOPS Operating Systems Review*, vol. 50, no. 2, pp. 789–804, 2016.

[24] Y. Oh, M. K. Yoon, W. J. Song, and W. W. Ro, "Finereg: Fine-grained register file management for augmenting gpu throughput," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 364–376, IEEE, 2018.

[25] J. Kloosterman, J. Beaumont, D. A. Jamshidi, J. Bailey, T. Mudge, and S. Mahlke, "Regless: Just-in-time operand staging for gpus," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 151–164, ACM, 2017.

[26] F. Khorasani, H. A. Esfeden, A. Farmahini-Farahani, N. Jayasena, and V. Sarkar, "Regmutex: Inter-warp gpu register time-sharing," in *Proceedings of the 45th Annual*

*International Symposium on Computer Architecture*, pp. 816–828, IEEE Press, 2018.

[27] T. D. Han and T. S. Abdelrahman, “Reducing branch divergence in gpu programs,” in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, p. 3, ACM, 2011.

[28] J. Meng, D. Tarjan, and K. Skadron, “Dynamic warp subdivision for integrated branch and memory divergence tolerance,” in *ACM SIGARCH Computer Architecture News*, vol. 38, pp. 235–246, ACM, 2010.

[29] S. R. Punyala, T. Marinakis, A. Komaee, and I. Anagnostopoulos, “Throughput optimization and resource allocation on gpus under multi-application execution,” in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 73–78, IEEE, 2018.

[30] 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.

[31] Y. Liang, X. Xie, G. Sun, and D. Chen, “An efficient compiler framework for cache bypassing on gpus,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 10, pp. 1677–1690, 2015.

[32] M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu, “Improving gpgpu resource utilization through alternative thread block scheduling,” in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 260–271, IEEE, 2014.

[33] L. Huang and Q. Xu, “Agesim: A simulation framework for evaluating the lifetime reliability of processor-based socs,” in *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, pp. 51–56, IEEE, 2010.

[34] F. Oboril and M. B. Tahoori, “Extratime: Modeling and analysis of wearout due to transistor aging at microarchitecture-level,” in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pp. 1–12, IEEE, 2012.

[35] A. Valero, F. Candel, D. Suárez-Gracia, S. Petit, and J. Sahuquillo, "An aging-aware gpu register file design based on data redundancy," *IEEE Transactions on Computers*, vol. 68, no. 1, pp. 4–20, 2019.

[36] H. Lee, M. Shafique, and M. A. Al Faruque, "Low-overhead aging-aware resource management on embedded gpus," in *Proceedings of the 54th Annual Design Automation Conference 2017*, pp. 1–6, 2017.

[37] H. Lee, M. Shafique, and M. A. Al Faruque, "Aging-aware workload management on embedded gpu under process variation," *IEEE Transactions on Computers*, vol. 67, no. 7, pp. 920–933, 2018.

[38] X. Chen, Y. Wang, Y. Liang, Y. Xie, and H. Yang, "Run-time technique for simultaneous aging and power optimization in gpgpus," in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2014.

[39] J. Tan and K. Yan, "Hvsm: Hardware-variability aware streaming processors' management policy in gpus," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 67–72, IEEE, 2018.

[40] A. Lotfi, A. Rahimi, L. Benini, and R. K. Gupta, "Aging-aware compilation for gp-gpus," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 2, p. 24, 2015.

[41] M. K. Jeong, M. Erez, C. Sudanthi, and N. Paver, "A qos-aware memory controller for dynamically balancing gpu and cpu bandwidth use in an mpsoc," in *Proceedings of the 49th Annual Design Automation Conference*, pp. 850–855, ACM, 2012.

[42] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo, "Simultaneous multikernel: Fine-grained sharing of gpus," *IEEE Computer Architecture Letters*, vol. 15, no. 2, pp. 113–116, 2016.

[43] S. Bhardwaj, W. Wang, R. Vattikonda, Y. Cao, and S. Vrudhula, "Predictive modeling of the nbti effect for reliable design," in *Custom Integrated Circuits Conference, 2006. CICC'06. IEEE*, pp. 189–192, IEEE, 2006.

[44] D. Rossi, V. Tenentes, S. Yang, S. Khursheed, and B. M. Al-Hashimi, "Aging benefits in nanometer cmos designs," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 64, no. 3, pp. 324–328, 2017.

[45] H. Chahal, V. Tenentes, D. Rossi, and B. M. Al-Hashimi, "Bti aware thermal management for reliable dvfs designs," in *2016 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pp. 1–6, IEEE, 2016.

[46] T. R. Mück, Z. Ghaderi, N. D. Dutt, and E. Bozorgzadeh, "Exploiting heterogeneity for aging-aware load balancing in mobile platforms," *IEEE Transactions on Multi-Scale Computing Systems*, vol. 3, no. 1, pp. 25–35, 2017.

[47] J. Abella, X. Vera, and A. Gonzalez, "Penelope: The nbti-aware processor," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 85–96, IEEE Computer Society, 2007.

[48] B. Raghunathan, Y. Turakhia, S. Garg, and D. Marculescu, "Cherry-picking: exploiting process variations in dark-silicon homogeneous chip multi-processors," in *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 39–44, IEEE, 2013.

[49] S. R. Sarangi, B. Greskamp, R. Teodorescu, J. Nakano, A. Tiwari, and J. Torrellas, "Varius: A model of process variation and resulting timing errors for microarchitects," *IEEE Transactions on Semiconductor Manufacturing*, vol. 21, no. 1, pp. 3–13, 2008.

[50] J. Xiong, V. Zolotov, and L. He, "Robust extraction of spatial correlation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 4, pp. 619–631, 2007.

[51] K. A. Bowman, A. R. Alameldeen, S. T. Srinivasan, and C. B. Wilkerson, "Impact of die-to-die and within-die parameter variations on the clock frequency and throughput of multi-core processors," *IEEE Transactions on Very Large Scale*

*Integration (VLSI) Systems*, no. 12, pp. 1679–1690, 2009.

[52] E. Humenay, D. Tarjan, and K. Skadron, "Impact of process variations on multicore performance symmetry," in *2007 Design, Automation & Test in Europe Conference & Exhibition*, pp. 1–6, IEEE, 2007.

[53] P. Aguilera, J. Lee, A. Farmahini-Farahani, K. Morrow, M. Schulte, and N. S. Kim, "Process variation-aware workload partitioning algorithms for gpus supporting spatial-multitasking," in *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1–6, IEEE, 2014.

[54] V. Mrazek, R. Hrbacek, Z. Vasicek, and L. Sekanina, "Evoapproxsb: Library of approximate adders and multipliers for circuit design and benchmarking of approximation methods," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 258–261, 2017.

[55] G. Zervakis, H. Amrouch, and J. Henkel, "Design automation of approximate circuits with runtime reconfigurable accuracy," *IEEE Access*, vol. 8, pp. 53522–53538, 2020.

[56] J. Schlachter, V. Camus, K. V. Palem, and C. Enz, "Design and applications of approximate circuits by gate-level pruning," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 5, pp. 1694–1702, 2017.

[57] R. Ye, T. Wang, F. Yuan, R. Kumar, and Q. Xu, "On reconfiguration-oriented approximate adder design and its application," in *International Conference on Computer-Aided Design (ICCAD)*, pp. 48–54, 2013.

[58] S. Jain, S. Venkataramani, and A. Raghunathan, "Approximation through logic isolation for the design of quality configurable circuits," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 612–617, March 2016.

[59] X. Jiao, V. Akhlaghi, Y. Jiang, and R. K. Gupta, "Energy-efficient neural networks using approximate computation reuse," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1223–1228, 2018.

[60] A. Raha, H. Jayakumar, and V. Raghunathan, "Input-based dynamic reconfiguration

of approximate arithmetic units for video encoding," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 3, pp. 846–857, 2015.

[61] O. Akbari, M. Kamal, A. Afzali-Kusha, and M. Pedram, "Rap-cla: A reconfigurable approximate carry look-ahead adder," *IEEE Trans. Circuits Syst., II, Exp. Briefs*, vol. 65, no. 8, pp. 1089–1093, 2016.

[62] O. Akbari, M. Kamal, A. Afzali-Kusha, and M. Pedram, "Dual-quality 4:2 compressors for utilizing in dynamic accuracy configurable multipliers," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, pp. 1352–1361, April 2017.

[63] V. Mrazek, Z. Vasicek, and L. Sekanina, "Design of quality-configurable approximate multipliers suitable for dynamic environment," in *NASA/ESA Conference on Adaptive Hardware and Systems*, pp. 264–271, 2018.

[64] B. Boroujerdian, H. Amrouch, J. Henkel, and A. Gerstlauer, "Trading off temperature guardbands via adaptive approximations," in *International Conference on Computer Design (ICCD)*, pp. 202–209, 2018.

[65] M. A. Hanif, F. Khalid, and M. Shafique, "Cann: Curable approximations for high-performance deep neural network accelerators," in *Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2019.

[66] A. Marchisio, V. Mrazek, M. Hanif, and M. Shafique, "Red-cane: A systematic methodology for resilience analysis and design of capsule networks under approximations," in *2020 Design, Automation and Test in Europe Conference (DATE)*, p. 4, 2020.

[67] F. Vaverka, V. Mrazek, Z. Vasicek, L. Sekanina, M. A. Hanif, and M. Shafique, "Tfapprox: Towards a fast emulation of dnn approximate hardware accelerators on gpu," in *2020 Design, Automation and Test in Europe Conference (DATE)*, p. 4, 2020.

[68] Z.-G. Tasoulas, I. Anagnostopoulos, L. Papadopoulos, and D. Soudris, "A

message-passing microcoded synchronization for distributed shared memory architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 5, pp. 975–979, 2018.

[69] X. Chen, Z. Lu, A. Jantsch, and S. Chen, "Supporting distributed shared memory on multi-core network-on-chips using a dual microcoded controller," in *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 39–44, European Design and Automation Association, 2010.

[70] D. Moloney, "1tops/w software programmable media processor," in *Hot Chips 23 Symposium (HCS), 2011 IEEE*, pp. 1–24, IEEE, 2011.

[71] L. Benini, E. Flamand, D. Fuin, and D. Melpignano, "P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012*, pp. 983–987, IEEE, 2012.

[72] M. Millberg, E. Nilsson, R. Thid, S. Kumar, and A. Jantsch, "The nostrum backbone-a communication protocol stack for networks on chip," in *17th International Conference on VLSI Design. Proceedings.*, pp. 693–696, IEEE, 2004.

[73] D. Dice, V. J. Marathe, and N. Shavit, "Flat-combining numa locks," in *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pp. 65–74, 2011.

[74] H. Sundell and P. Tsigas, "Lock-free and practical deques using single-word compare-and-swap, computing science, chalmers university of technology," tech. rep., Tech. Rep. 2004-02, Mar, 2004.

[75] Z.-G. Tasoulas and I. Anagnostopoulos, "Kernel-based resource allocation for improving gpu throughput while minimizing the activity divergence of sms," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 2, pp. 428–440, 2019.

[76] L. Huang, F. Yuan, and Q. Xu, "Lifetime reliability-aware task allocation and

scheduling for mpsoc platforms," in *2009 Design, Automation & Test in Europe Conference & Exhibition*, pp. 51–56, IEEE, 2009.

[77] W. Wang, S. Yang, S. Bhardwaj, S. Vrudhula, F. Liu, and Y. Cao, "The impact of nbti effect on combinational circuit: modeling, simulation, and analysis," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 18, no. 2, pp. 173–183, 2009.

[78] E. Takeda, Y. Nakagome, H. Kume, and S. Asai, "New hot-carrier injection and device degradation in submicron mosfets," *IEE Proceedings I (Solid-State and Electron Devices)*, vol. 130, no. 3, pp. 144–150, 1983.

[79] Q. Xu and M. Annavaram, "Pats: Pattern aware scheduling and power gating for gpgpus," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pp. 225–236, 2014.

[80] H. Kaul, M. A. Anders, S. K. Mathew, S. K. Hsu, A. Agarwal, R. K. Krishnamurthy, and S. Borkar, "A 300 mv 494gops/w reconfigurable dual-supply 4-way simd vector processing accelerator in 45 nm cmos," *IEEE Journal of Solid-State Circuits*, vol. 45, no. 1, pp. 95–102, 2009.

[81] T. G. Rogers, M. OConnor, and T. M. Aamodt, "Cache-conscious wavefront scheduling," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 72–83, IEEE, 2012.

[82] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 163–174, IEEE, 2009.

[83] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "Gpuwattch: enabling energy optimizations in gpgpus," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, pp. 487–498, 2013.

[84] W. Huang, S. Ghosh, S. Velusamy, K. Sankaranarayanan, K. Skadron, and M. R.

Stan, "Hotspot: A compact thermal modeling methodology for early-stage vlsi design," *IEEE Transactions on very large scale integration (VLSI) systems*, vol. 14, no. 5, pp. 501–513, 2006.

[85] Z.-G. Tasoulas and I. Anagnostopoulos, "Optimizing performance of gpu applications with sm activity divergence minimization," in *2018 25th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pp. 621–624, IEEE, 2018.

[86] S. Borkar, T. Karnik, and V. De, "Design and reliability challenges in nanometer technologies," in *Proceedings of the 41st annual Design Automation Conference*, pp. 75–75, 2004.

[87] K. A. Bowman, S. G. Duvall, and J. D. Meindl, "Impact of die-to-die and within-die parameter fluctuations on the maximum clock frequency distribution for gigascale integration," *IEEE Journal of solid-state circuits*, vol. 37, no. 2, pp. 183–190, 2002.

[88] Z.-G. Tasoulas and I. Anagnostopoulos, "Performance and aging aware resource allocation for concurrent gpu applications under process variation," *IEEE Transactions on Nanotechnology*, vol. 18, pp. 717–727, 2019.

[89] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Annual International Symposium on Computer Architecture*, pp. 1–12, 2017.

[90] Z.-G. Tasoulas, G. Zervakis, I. Anagnostopoulos, H. Amrouch, and J. Henkel, "Weight-oriented approximation for energy-efficient neural network inference accelerators," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 12, pp. 4670–4683, 2020.

[91] A. Renda, J. Frankle, and M. Carbin, "Comparing rewinding and fine-tuning in neural network pruning," *arXiv:2003.02389*, 2020.

[92] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Advances in neural information processing systems*,

pp. 1135–1143, 2015.

[93] M. Zhu and S. Gupta, "To prune, or not to prune: exploring the efficacy of pruning for model compression," *arXiv:1710.01878*, 2017.

[94] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[95] A. Krizhevsky, G. Hinton, *et al.*, "Learning multiple layers of features from tiny images," 2009.

[96] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *IEEE conference on computer vision and pattern recognition*, pp. 4510–4520, 2018.

[97] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel, "Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition," *Neural networks*, vol. 32, pp. 323–332, 2012.

[98] A. Mogelmose *et al.*, "Vision-based traffic sign detection and analysis for intelligent driver assistance systems: Perspectives and survey," *IEEE Trans. Intell. Transp. Syst.*, vol. 13, no. 4, pp. 1484–1497, 2012.

[99] Y. Choi, M. El-Khamy, and J. Lee, "Learning low precision deep neural networks through regularization," *arXiv:1809.00095*, 2018.

## VITA

Graduate School
Southern Illinois University

Zois Gerasimos Tasoulas

National Technical University of Athens, Athens, Greece
Diploma, Electrical and Computer Engineering, April 2016

Special Honors and Awards:

Recipient of the Graduate Doctoral Fellowship for the academic year 2019 – 2020 from Southern Illinois University Carbondale.

Dissertation Paper Title:

Resource Management and Application Customization for Hardware Accelerated Systems

Major Professor: Dr. Iraklis Anagnostopoulos

Publications:

Tasoulas, Z. G., Zervakis, G., Anagnostopoulos, I., Amrouch, H., & Henkel, J. (2020). Weight-oriented approximation for energy-efficient neural network inference accelerators. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 67(12), 4670 – 4683.

Dickerson, J., Galanis, I., Tasoulas, Z. G., Kinley, L., & Anagnostopoulos, I. (2020, June). Adaptive Approximate Computing on Hardware Accelerators Targeting Internet-of-Things. In *2020 IEEE 6th World Forum on Internet of Things (WF-IoT)* (pp. 1-6). IEEE.

Tasoulas, Z. G., & Anagnostopoulos, I. (2019). Improving GPU Performance with aPower-Aware Streaming Multiprocessor Allocation Methodology. *Electronics*, 8(12), 1451.

Tasoulas, Z. G., & Anagnostopoulos, I. (2019). Kernel-Based Resource Allocation for Improving GPU Throughput While Minimizing the Activity Divergence of SMs. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 67(2), 428 – 440.

Tasoulas, Z. G., & Anagnostopoulos, I. (2019). Performance and aging aware resource allocation for concurrent GPU applications under process variation. *IEEE Transactions on Nanotechnology*, 18, 717 – 727.

Tasoulas, Z. G., & Anagnostopoulos, I. (2018, December). Optimizing Performance of GPU Applications with SM Activity Divergence Minimization. In *2018 25th IEEE International Conference on Electronics, Circuits and Systems (ICECS)* (pp. 621-624). IEEE.

Tasoulas, Z. G., Guss, R., & Anagnostopoulos, I. (2018, October). Performance-Based and Aging-Aware Resource Allocation for Concurrent GPU Applications. In *2018 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)* (pp. 16). IEEE.

Tasoulas, Z. G., Anagnostopoulos, I., Papadopoulos, L., & Soudris, D. (2018). A Message-Passing Microcoded Synchronization for Distributed Shared Memory Architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(5), $975 - 979$.