

**EVALUATING IMPACTS OF  
MICRO-ARCHITECTURAL METRICS ON ERROR  
RESILIENCE AND PERFORMANCE OF GENERAL  
PURPOSE GPU APPLICATIONS**

**A Thesis Submitted to  
the Graduate School of Engineering and Sciences of  
İzmir Institute of Technology  
in Partial Fulfillment of the Requirements for the Degree of**

**MASTER OF SCIENCE**

**in Computer Engineering**

**by  
Burak TOPÇU**

**July 2023  
İZMİR**

We approve the thesis of **Burak TOPÇU**

**Examining Committee Members:**

---

**Assistant Professor Dr. Erdem ALKIM**  
Department of Computer Science, Dokuz Eylül University

---

**Professor Dr. Cüneyt Fehmi BAZLAMAÇCI**  
Department of Computer Engineering, Izmir Institute of Technology

---

**Assistant Professor Dr. Işıl ÖZ**  
Department of Computer Engineering, Izmir Institute of Technology

**10 July 2023**

---

**Assistant Professor Dr. Işıl ÖZ**  
Department of Computer Engineering  
Izmir Institute of Technology

---

**Professor Dr. Cüneyt Fehmi BAZLAMAÇI**  
Head of the Department of  
Computer Engineering

---

**Professor Dr. Mehtap EANES**  
Dean of the Graduate School of  
Engineering and Sciences

## ACKNOWLEDGMENTS

Firstly, I would like to begin by expressing my gratefulness to my supervisor, Associate Professor Işıl Öz. She has illustrated my research environment by pointing out hot topics and has accompanied me by discussing my deductions and experimental results. Furthermore, I have had the opportunity to publish and present my works at academic conferences and journals in a funded way with the help of project collaborations that she leads.

I would like to thank the PARS research group due to our academic meetings in which the group member presented current works in their research domain and resources that enabled me to speed up the experiments. In addition, I would like to thank the Scientific and Technological Research Council of Turkey (TÜBİTAK) due to the project's partnerships at FTGPGPU (Hardware Fault Tolerance Analysis for General Purpose Graphics Processing Units (GPGPU) Applications, Grant No: 119E011) and RAPPROX (Resource-Aware Compiler Design for Approximate Computing Techniques in General Purpose Graphics Processing Units (GPGPU) Applications, Grant No: 122E395) in which I worked as a graduate-funded researcher.

I would like to thank my colleagues for their helpful, understanding, and supportive approaches such that we distributed the courses depending on the TAs' research focus and the workload of these courses equally.

I would like to thank my family members. Their beyond-conditions love motivates me to be lively in my daily life, which positively reflects my academic research motivation and dedication. Moreover, I would like to thank my girlfriend and close friends because they have always supported me whenever I feel overwhelmed.

Finally, I would like to thank and express my loyalty to the greatest leader, Atatürk, who is the founder leader of our country, paving the way for or forming the state systems & ministries, and educational institutions such as IZTECH.

# ABSTRACT

## EVALUATING IMPACTS OF MICRO-ARCHITECTURAL METRICS ON ERROR RESILIENCE AND PERFORMANCE OF GENERAL PURPOSE GPU APPLICATIONS

Rapidly growing data processing tasks require powerful and energy-efficient heterogeneous computing systems, and GPUs take on a significant mission for those systems in accelerating heavy workloads by executing multiple parallel tasks concurrently. Increasing architectural complexity and widening employment of GPUs bring error resiliency concerns for safety-critical applications. Furthermore, approaches that enhance performance and reduce energy dissipation handle error resiliency on GPUs through approximate computing solutions. Evaluating error resiliency in terms of either identifying error proneness of a system or investigating approximations without much disturbing the output necessitates robust knowledge about the execution of a program on a device.

In this thesis, we develop a runtime performance and power monitoring tool visualizing the execution with detailed micro-architectural metrics. By utilizing the tool, we acquire several fundamental understandings about runtime performance bottlenecks and how perturbations affect output quality. Afterward, we propose a framework predicting fault vulnerability for error-resilient GPU applications. The framework can accurately estimate error tolerance and saves from analyzing the fault occurrence probability requiring significant effort. Depending on the performance bottlenecks observed with the tool and the error propagation gained during prediction experiments, we introduce a hardware-based approximation computing approach targeting to improve the performance and power of GPU programs, especially memory-bound ones. The approximation method, which resolves memory utilization bottlenecks at runtime, enhances performance by 1.49× (up to 2.1×) and diminishes energy consumption by 28.4% (up to 52.6%) while maintaining the accuracy on the output above 98%.

## ÖZET

### MİKRO-MİMARİ METRİKLERİN GENEL AMAÇLI GPU UYGULAMA HATALARINA VE PERFORMANSINA ETKİLERİNİN DEĞERLENDİRİLMESİ

Hızla artan veri işleme görevleri güçlü ve enerji tüketimi açısından verimli heterojen hesaplama ortamları gerektirir ve GPU cihazları birçok görevi paralel şekilde çalıştırarak bu sistemlerdeki yoğun iş yüklerini hızlandırmada önemli bir misyon üstlenir. Artan mimari karmaşıklık ve GPU cihazlarının yaygın şekilde kullanılması güvenlik açısından önemli uygulamalar için hataya karşı dayanıklılığa ilişkin endişeler ortaya çıkarır. Yanı sıra, performansı artırırken enerji tüketimini azaltmayı hedefleyen yaklaşımlar ise hataya karşı dayanıklılığı yakınsamalar yapmak ve faydalanmak yönüyle konuyu ele alır. Hataya karşı dayanıklılığı, hata oluşumuna yönelimi veya çıktıyı çok bozmayacak yakınsamaları değerlendirmek bir programın cihazdaki çalışmasına yönelik kapsamlı bilgilere sahip olmayı gerekli kılar.

Bu tezde, GPU'daki gerçek zamanlı çalışmayı mikro mimari ölçümler aracılığıyla sunan ve görselleştiren bir performans ve güç izleme aracı geliştirdik. Bu araç sayesinde, çalışma esnasındaki performans darboğazları ve meydana gelen hataların çıktı kalitesini nasıl etkilediği hakkında birçok temel anlayış elde ettik. Daha sonra, GPU uygulamaları için hata güvenlik açığını tahmin eden bir yapı öneriyoruz. Bu yapı, hata toleransını doğru bir şekilde tahmin etmeyi sağlar ve önemli çaba gerektiren hata oluşma olasılığını analiz etmekten kurtarır. İzleme aracıyla gözlemlenen performans darboğazları ve tahmin deneyleri sırasında elde edilen hata yayılımı gözlemlerini temel alarak, özellikle bellek kullanımından kaynaklı GPU programlarının performansını ve gücünü iyileştirmeyi hedefleyen donanım tabanlı bir yakınsama aracı sunuyoruz. Çalışma zamanında bellek kullanımına yönelik darboğazlarını çözen yakınsama yöntemi çıktıdaki doğruluğu %98'in üzerinde tutarken, performansı  $1,49\times$  (en fazla  $2,1\times$ ) artırır ve enerji tüketimini %28,4 (%52,6'ya kadar) azaltır.

*Once upon a time, these apes were also humans ridiculing others, and we live together  
with their evolved forms; we should pay more attention!*

---

*Maymunland, Indigo*

# TABLE OF CONTENTS

LIST OF FIGURES .....	ix
LIST OF TABLES .....	xi
CHAPTER 1. INTRODUCTION .....	1
1.1. Thesis Organizations and Contributions .....	4
CHAPTER 2. BACKGROUND .....	7
2.1. CUDA Programming Language .....	7
2.2. GPU Architectures .....	8
2.3. Micro-architectural metrics .....	9
2.3.1. GPGPU-Sim .....	10
2.4. Soft Error Vulnerability .....	11
2.5. Approximate Computing .....	13
CHAPTER 3. GPPRMON: GPU RUNTIME PERFORMANCE AND POWER MONITORING TOOL .....	14
3.1. Related Work .....	16
3.2. Methodology .....	18
3.2.1. Micro-architectural Metric Collection .....	18
3.2.1.1. Performance Metrics .....	19
3.2.1.2. Power Metrics .....	20
3.2.2. Visualization .....	21
3.3. Case Studies .....	25
3.3.1. Performance Bottleneck Analysis and its Power Impacts for a Memory-Intensive Workload .....	25
3.3.2. Performance-Power Analysis of an Embedded Application .....	30
3.4. Summary .....	34
CHAPTER 4. SOFT ERROR VULNERABILITY PREDICTION OF GENERAL PURPOSE GPU APPLICATIONS .....	35
4.1. Related Work .....	36
4.2. Methodology .....	39

4.2.1. Fault Injection Framework .....	40
4.2.2. Metric Collection .....	41
4.2.3. Data Preprocessing .....	44
4.2.4. Feature Selection .....	45
4.2.5. Outlier Elimination .....	47
4.2.6. Prediction Model Evaluation .....	49
4.3. Experimental Study .....	51
4.3.1. Experimental Setup .....	51
4.3.2. Experimental Results .....	54
4.3.2.1. Preprocessing Methods .....	54
4.3.2.2. Feature Selection .....	55
4.3.2.3. Outlier Elimination .....	56
4.3.2.4. Regression Results .....	57
4.3.2.5. Classification Results .....	58
4.4. Summary .....	60
CHAPTER 5. APPROXTRACKER: MEMORY-DRIVEN GPU APPROXIMA- TOR ENHANCING PERFORMANCE, ENERGY EFFICIENCY, AND DATA UTILIZATION .....	62
5.1. Related Work .....	63
5.2. Methodology .....	65
5.2.1. ApproxTrackerL1D .....	67
5.2.2. ApproxTrackerL2 .....	70
5.3. Experimental Study .....	71
5.3.1. Experimental Setup .....	71
5.3.2. Experimental Results .....	73
5.3.3. ApproxTracker Performance and Driven-Power on Various GPU Applications .....	74
5.3.4. ApproxTracker Performance and Driven-Power Improvements on Various Datasets .....	80
5.4. Summary .....	83
CHAPTER 6. CONCLUSION .....	86
REFERENCES .....	88



# LIST OF FIGURES

<u>Figure</u>		<u>Page</u>
Figure 2.1	CUDA, PTX, and SASS code snippets for <i>vectorAdd</i> GPU code. ....	7
Figure 2.2	The 3D Grid and Thread Block representations of a GPU kernel workload. ....	8
Figure 2.3	A memory workload analysis overview section snapshot taken from Nsight Compute tool. ....	10
Figure 2.4	The simulator’s modern GPU modeling. ....	11
Figure 2.5	Fault detection and classification flowchart Öz and Karadaş (2022). .	12
Figure 3.1	A workflow overview of <i>GPPRMOn</i> framework. ....	19
Figure 3.2	<i>General View</i> displaying average performance and power consumption measurements at runtime. ....	22
Figure 3.3	<i>Spatial View</i> revealing memory access statistic at runtime. ....	23
Figure 3.4	<i>Temporal View</i> monitors instruction executions together with the performance and power consumption of the corresponding SM. ....	24
Figure 3.5	Average memory access statistics in the cycle range of [5000, 100000]. 27	
Figure 3.6	Instruction monitoring for the load instructions on SM0 in the cycle range of [5000, 30000]. ....	28
Figure 3.7	Memory performance overview in the cycle range of [5000, 9500]. .	29
Figure 3.8	IPC and dissipated power metrics throughout the [5000, 155000] cycle interval of Kernel 0. ....	33
Figure 3.9	Instructions preparing threads for kernel task with thread-specific data in the cycle range of [5000, 6500]. ....	33
Figure 4.1	General overview of fault prediction framework. ....	40
Figure 4.2	SDC, crash, and masked fault rates for target GPU kernels. ....	42
Figure 4.3	The upper and lower triangles show Pearson and Spearman correlation results, respectively, between the simulator/profiler metrics and the fault rates (left is with the simulator and right is with profiler features). ....	47

Figure 4.4	Prediction experiments, where preprocessing methods result in the highest accuracy values by keeping feature selection, outlier eliminator, and ML classifier the same. ....	54
Figure 4.5	Spearman and Pearson correlation results between the features and the fault rates. ....	55
Figure 4.6	Prediction experiments, where feature selection methods result in the highest accuracy values by keeping preprocessing, outlier eliminator methods, and ML classifier the same. ....	56
Figure 4.7	Prediction experiments, where outlier elimination methods result in the highest accuracy values by keeping preprocessing, feature selection methods, and ML classifier the same. ....	57
Figure 5.1	Execution flow of a memory request starting from LD/ST unit. ....	66
Figure 5.2	Locating ApproxTrackerL1D within the LD/ST unit on SMs. ....	67
Figure 5.3	Average miss rates on L1D and L2 caches experimented with naive, ApproxTrackerL1D and ApproxTrackerL2 versions. ....	76
Figure 5.4	Normalized IPC and GPU Simulation Cycle experimented with naive, ApproxTrackerL1D, and ApproxTrackerL2 versions. ....	77
Figure 5.5	Normalized runtime power dissipation and energy consumption obtained through naive ApproxTrackerL1D and ApproxTrackerL2 versions. ....	79
Figure 5.6	Average miss rates on L1D and L2 caches for various data types experimented with naive, ApproxTrackerL1D and ApproxTrackerL2 versions. ....	80
Figure 5.7	The number of L2 cache accesses and misses obtained through naive, ApproxTrackerL1D, and ApproxTrackerL2 versions among various data types. ....	81
Figure 5.8	IPC and GPU simulation rates with ApproxTrackerL1D/L2 approaches for various data types. ....	82
Figure 5.9	Average driven-power and energy consumption with ApproxTrackerL1D/L2 for various data types. ....	84

# LIST OF TABLES

<u>Table</u>		<u>Page</u>
Table 3.1	GV100 Based on Volta Architecture Configuration Specifications. . .	25
Table 3.2	Page Ranking kernel performance statistics. ....	26
Table 3.3	Dissipated power measured during K1’s execution on GV100 in the cycle range of [5000,10000]. ....	31
Table 3.4	Fast Fourier kernel performance statistics. ....	32
Table 4.1	The metrics collected from the simulator. ....	43
Table 4.2	The metrics collected from the profiler. ....	44
Table 4.3	Selected features from the collected metrics. ....	48
Table 4.4	CUDA applications used in our experiments. ....	52
Table 4.5	Preprocessing, Feature Selection/Reduction, and Outlier Elimina- tion methods and their hyper-parameter configurations. ....	53
Table 4.6	Regression accuracy results for masked faults on each machine learn- ing algorithm. ....	58
Table 4.7	SDC and Crash rates classification results for 2/3-class evaluations among all preprocessing methods. ....	59
Table 5.1	CUDA applications used in our experiments. ....	72
Table 5.2	Graph specifications. ....	72
Table 5.3	RTX2060 GPU configuration based on Turing architecture. ....	73
Table 5.4	Baseline performance overview among target applications. ....	75
Table 5.5	The number of memory accesses on caches experimented with naive, ApproxTrackerL1D, and ApproxTrackerL2 versions. ....	78

# CHAPTER 1

## INTRODUCTION

Today's data processing tasks with drastically increasing data require heterogeneous computer systems with high performance and energy efficiency. Massively parallel GPU architectures play a key role in accelerating workloads that can run in parallel, such as deep learning, big data, scientific computing, and streaming applications. According to TOP500 (2023) statistics, 21 entries out of the listed 28 accelerators/co-processors are GPUs or GPU-based systems manufactured by NVIDIA, and the most preferred GPU device is the Volta architecture-based Quadro GV100 GPU (NVIDIA (2018b)). Besides NVIDIA, AMD and Intel are the other GPU manufacturing leaders in the industry and exist in the same statistics.

GPU architectures mainly comprise many streaming multiprocessors (SMs) consisting of several multi-functional computational resources and memory partitions containing various storage units (Aamodt et al. (2018)). The SMs offer much more execution resources than multi-cores and special functional units developed for faster execution of computationally complex operations. The GPU memory hierarchy generically involves an on-chip cache (usually L1 on SMs), an interconnection network, and memory partitions. While the interconnection network provides the routing for the memory accesses between SMs and the memory partitions, the L2 cache and DRAM banks reside in the memory partitions. The swift evolution of high-compute tasks has led to the transformation of GPUs into versatile computational devices with much more complex architectures. While GPUs traditionally accelerate data-independent parallel workloads like natural language processing (NLP) or computer vision of artificial intelligence (AI) applications, modern GPUs further boost the performance by utilizing either domain-specific solutions such as tensor operators (Appleyard and Yokim (2017)) and Deep Learning Super Sampling (DLSS, Stine et al. (2021)) or more sophisticated execution pipeline designs like Shader Execution Reordering (SER, Rusch and Hart (2022)).

On the other side, some tasks do not effectively utilize the computational power of GPUs, which breaks down the generic applicability. To illustrate, sorting algorithms arranging elements in a specific execution flow based on comparison operations generally cannot reach devices' maximum TFLOP rates because the control flow and particular ordering necessities can cause computing resources to be idle due to waiting or control

flows. Furthermore, algorithms exposing heavy memory workloads or irregular data accesses incur memory bounds since thousands of parallel threads create pollution in both on-chip and off-chip caches whenever the data locality is low or totally lacking (Hong et al. (2022); Jain et al. (2019)). Besides the insufficiency of generic execution support, energy consumption concerns are addressed more in the literature, as GPUs are widely utilized throughout diverse applications and fields. While both performance and energy improvements contribute efficient execution of GPU programs, the design decisions become critical and get complicated, requiring the evaluation of the trade-offs between both factors (Guerreiro et al. (2019); Krzywaniak et al. (2022); Sun et al. (2018)).

The cumulative developments in architecture complexity and the scaling up of heterogeneous computer systems, including many sophisticated GPUs, have led to error resiliency trends that generate incorrect output. First, the rising reliability concerns on error vulnerability (Borkar (2005)) and high costs of conventional redundancy-based resiliency solutions bring significant research to evaluate error tolerance and seek less-than-perfect other precautions. Especially, software error-resilient solutions propose inexpensive hardware or software detectors to identify abnormal software behavior (Dimitrov and Zhou (2007); Golubeva et al. (2003); Li et al. (2008)). Even if these solutions are promising because of their ultra-low cost, they occasionally let some errors as silent data corruptions (SDCs). Second, error tolerance on GPUs presents the emerging field of approximate computing, which focuses on intentionally and carefully compromising accuracy to achieve improved performance or energy efficiency. Approximate computing proposes various techniques to reduce or completely eliminate the performance-limiting factors with tolerable error rates. For example, the proposed methods include executing more workload on the same device by utilizing operand similarity or smaller operands from 64-bit to 32-bit/32-bit to 16-bit (Garcia et al. (2021); Peroni et al. (2020); Wong et al. (2016)). Additionally, approximation methods based on hardware, software, or compiler can save from memory bottlenecks by removing long-latency memory operations, early cache evictions, and memory pipeline stalls, especially for memory-bounded applications (Aktılav and Öz (2022); Gao et al. (2022); Hoshino et al. (2018); Maier et al. (2019); Zhao et al. (2020)).

To facilitate the widespread adoption of both software-anomaly-based error detection and approximate computing, it is crucial to comprehend how perturbations, whether intentional approximations or unintended errors, affect the outcome and quality of a computation. This challenge is known as the perturbation-outcome problem (Venkatagiri et al. (2016)). Ideally, a solution to this problem would possess the following characteris-

tics: (1) automatic determination without imposing excessive burden on programmers, (2) applicability to general-purpose applications, (3) compatibility with various perturbation models, and (4) guaranteed understanding of the impact on overall output quality, as well as the resulting improvements in performance, energy efficiency, and resilience costs. While a comprehensive solution that meets all these requirements is still elusive, researchers have made substantial progress by relaxing certain cases. To understand how perturbations, either deliberate approximations or unintentional errors, generally affect the outcome of the computation, diving into micro-architectural metrics, which provides insight into the internal structure and behavior of the execution, and evaluating them concerning approximations for performance and energy dissipation improvements can clear up obscures. Micro-architectural metrics provide valuable information for interpreting and determining error resiliency by facilitating performance analysis, supporting fault injection studies, enabling predictive modeling, and guiding system optimization and design decisions.

The state-of-the-art GPU simulators like Multi2Sim (Ubal et al. (2012)), GPGPU-Sim (Khairy et al. (2020)), and gem5 (Power et al. (2015)) model a virtual GPU and execute a GPU code in a cycle-accurate manner. Likewise, GPU manufacturers offer profiling instruments such as NVIDIA Nsight Compute (NVIDIA (2022a)) and System Profiling (NVIDIA (2022d)) tools to evaluate application behavior and interaction with the hardware. Even if the simulators are open to expanding source code to obtain the application's micro-architectural behavior and performance logs depending on the research direction, such expansion still does not provide metrics for general-purpose evaluation capability. Moreover, both profilers and simulators lack runtime evaluation as the provided metrics correspond to the occupation and interaction of the GPU kernel functions with the hardware. In addition to the lack of a comprehensive observation tool, none of the tools directly reports GPU programs' dynamic hardware occupation, execution behavior and metrics, and power consumption at runtime.

The limitations of the simulators and profilers hinder deeply investigating error resiliency. The error tolerance analysis, such as determining fault occurrence probability, requires many repetitive experiments and may become impractical for long-running systems, especially on GPU simulators. Even if predicting the error vulnerability without running numerous fault injection tests on each target software has become viable for CPU systems (Guo et al. (2021); Jauk et al. (2019); Laguna et al. (2016); Lu et al. (2014); Oliveira et al. (2018); Öz and Arslan (2021)), the estimation approaches utilizing micro-architectural metrics are limited Kalra et al. (2018); Nie et al. (2018) for GPUs.

In addition, the approximate computing domain offers various hardware-, software- or compiler-based error-resilient approximation solutions (Gao et al. (2022); Garcia et al. (2021); Hoshino et al. (2018); Maier et al. (2019); Peroni et al. (2020); Wong et al. (2016); Zhao et al. (2020)) which control the error tolerably and aims to improve execution performance and energy usage by reducing or completely eliminating the performance limiting factors. However, GPUs still cannot meet the expected hardware performance of heavy memory workloads, especially for sparse data processing (Jiao et al. (2010)), and the limited GPU performance, which is slightly better than processing on multi-cores, cannot be preferable due to the high energy consumption. Moreover, those approximate computing approaches do not pay enough attention to how approximated values propagate through the execution.

## 1.1. Thesis Organizations and Contributions

In this thesis, we evaluate the impacts of micro-architectural metrics on performance and error resiliency based on soft error vulnerability prediction and approximate computing approaches. Before the evaluation, we first develop a tool through the simulator to investigate the runtime behavior of any execution on GPUs considering micro-architectural metrics. Developing such a tool improves our knowledge about the relationship between application behaviors and micro-architectural metrics occupying hardware with diverse architecture types and resources. In the second part of the thesis, we create a prediction framework to estimate the soft error vulnerability and verify that a comprehensive machine learning (ML)-based estimation approach can determine the faults which corrupt the output or stop the execution by employing micro-architectural metrics. Furthermore, we discover that the runtime value transitions such as faults either do not affect the output much when the transient data changes are local, and the operands have no critical effect on the output or do not affect the output at all for most of the benchmark applications. By benefiting from error propagation founding and realized performance bottlenecks with the runtime evaluation tool, we offer an approximate computing method based on runtime memory performance traces aiming to improve performance and energy dissipation by reducing performance-limiting factors within tolerable accuracy disturbances on output.

The main contributions of this thesis are summarized as follows:

- i. We introduce **GPPRMOn**, a runtime performance and power monitoring tool for GPU programs. GPPRMOn, a simulation-based framework, dynamically collects

certain micro-architectural metrics and reports performance and power consumption observations at runtime. GPPRMon provides a visualization interface that presents both spatial and temporal views of the execution. Additionally, we provide two different case studies with usage scenarios in which one experiments server model heavy GV100 (NVIDIA (2018b)) GPU, and the other Jetson AGX Xavier (NVIDIA (2019)) corresponds to the GPU used in embedded applications. To the best of our knowledge, GPPRMon, which enables the user to perform a fine-granularity evaluation of the target execution by observing instruction-level microarchitectural features, is the first work visualizing the execution for multiple user-configurable scenarios, relating memory hierarchy utilization with performance and tracking power dissipation at runtime. The GPPRMon research paper was accepted to be published at The 2nd International Workshop on Resource Awareness of Systems and Society (RAW, 2023) of the 29th International European Conference on Parallel and Distributed Computing and Minisymposia.

- ii. We present an ML-based soft error vulnerability prediction for GPU applications. Our research contributes to many fault injection experiments and discovers the patterns between fault rates and micro-architectural program characteristics collected via a simulator or profilers. To the best of our knowledge, this is the first study that uses a comprehensive set of regression and classification models supported by feature selection, preprocessing, and outlier elimination techniques and trained with micro-architectural indicators to predict soft error vulnerability at the kernel level instead of the whole GPU application. This research's initial version was published at the 30th Euromicro International Conference on Parallel, Distributed, and Network-based Processing (PDP) (Topçu and Öz (2022)), and the extended version was published in The Journal of Supercomputing (Topçu and Öz (2022)).
- iii. We develop memory-centric approximation-based approaches, ApproxTrackers, by detecting and flushing long-latency memory operations deliberately at runtime to reduce problems related to the memory wall. We conduct preliminary classification experiments to determine approximable target GPU applications by utilizing the microarchitectural metrics, and those applications mostly consist of memory-bound ones with irregular memory access behaviors. Although our approach targets memory-bound applications, we apply our solution to other domains. At the end of this chapter, we analyze the performance and energy consumption improvements within small accuracy deviations according to the experimental results. After defending this thesis, we will submit our hardware approximation tracker, Approx-



Tracker, to a conference or a journal.

Background chapter, *2nd Chapter*, provides baseline information for CUDA programming language, GPU architectures, micro-architectural metrics, simulators and profilers, soft error vulnerability, and approximate computing topics. In the *3rd Chapter*, we introduce the runtime performance and power monitoring tool mentioned above and present two case studies for analyzing runtime performance and power consumption with a server and an embedded application GPU. In the *4th Chapter*, we offer a comprehensive ML-based framework for predicting soft error vulnerability. We analyze the fault tolerance of GPU applications and associate these errors with micro-architectural metrics. In the *5th Chapter*, we propose an approximate computing approach that improves performance and reduces energy consumption based on error propagation and performance bottlenecks analysis. Lastly, we summarize the contributions of our work to the literature, explain expected future research directions, and share our conclusions in the *6th Chapter*.

# CHAPTER 2

## BACKGROUND

### 2.1. CUDA Programming Language

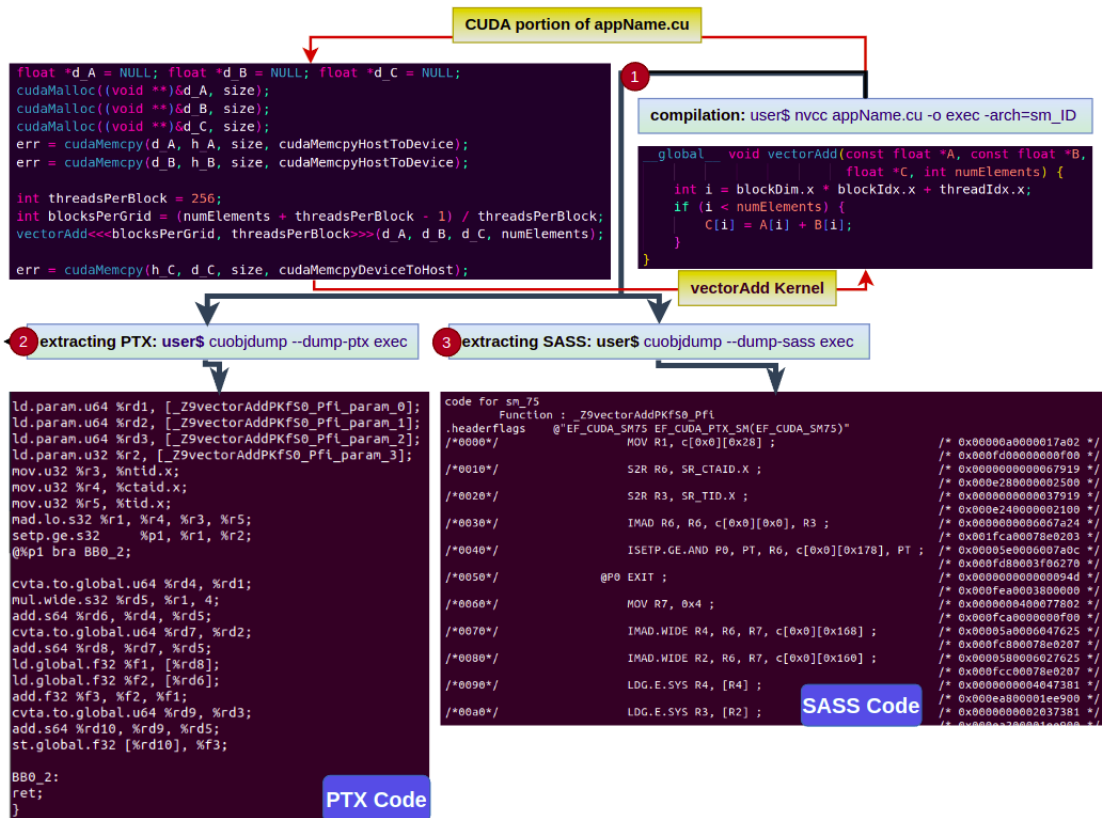


Figure 2.1. CUDA, PTX, and SASS code snippets for *vectorAdd* GPU code.

Compute Unified Device Architecture (CUDA) (NVIDIA (2023a)) designed with Fortran, C, and C++, provides a programming API to accelerate general-purpose computing, such as scientific simulations and machine learning with GPU functions, called GPU kernels, by leveraging the computational power of NVIDIA GPUs. Figure 2.1 demonstrates a GPU kernel example, given as the *vectorAdd* with CUDA code snippet. NVCC (NVIDIA (2023b)) compiles CUDA codes and generates a SASS executable where source and assembly (SASS, NVIDIA (2015)) corresponds to the real machine assembly instructions. [1] in Figure 2.1 reveals a compilation command where *sm\_ID* needs to be config-

ured depending on the GPU architecture. For example, in this thesis, we experiment with Pascal, Volta, and Turing GPUs and compile the CUDA codes with `sm_61`, `sm_70`, and `sm_75` identifiers, respectively. By CUDA-Object-Dump (cuobjdump) utility as in [2], we can obtain Parallel Thread Executable (PTX) instructions (NVIDIA (2023d)), which corresponds to virtual machine instructions and are capable of one-to-one representation of SASS. The code obtained with [3] displays the SASS instructions for vectorAdd GPU kernel. NVIDIA provides a CUDA debugger, `cuda-gdb` (NVIDIA (2022b)), and the CUDA developers can set breakpoints, observe states of hardware components, and change the stored values (i.e., in registers) through the debugger at runtime. In this thesis, we experiment with CUDA applications and NVIDIA GPU architectures and employ all the mentioned utilities through the experiments.

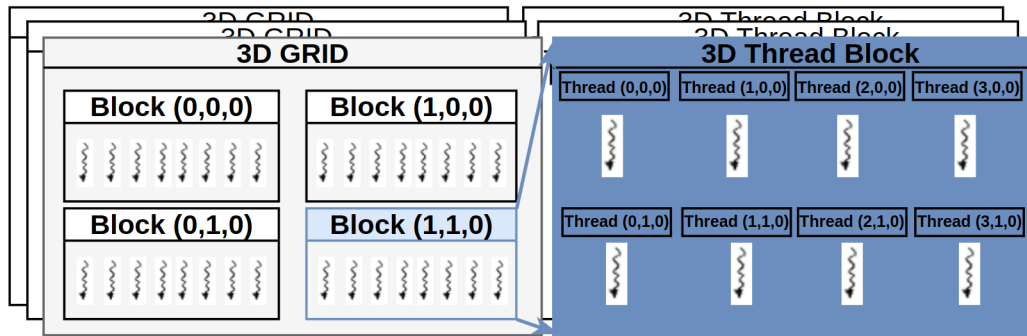


Figure 2.2. The 3D Grid and Thread Block representations of a GPU kernel workload.

Figure 2.2 represents a GPU kernel that consists of a 3D grid space, where each grid has a 3D thread block, including multiple threads. Before launching a GPU kernel, the developer agrees on a data management policy between CPU and GPU memory. Warps, consisting of groups of 32 threads within the same thread block, are the smallest execution element in the CUDA context and run in parallel. The kernel completes its execution after terminating all thread blocks. Although each thread operates with different data and operands, the warping aims to execute and complete the same instructions of grouped threads in parallel.

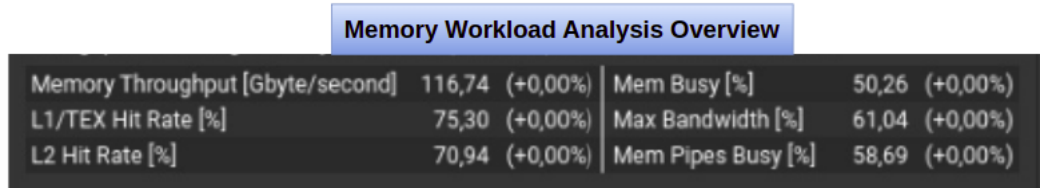
## 2.2. GPU Architectures

Modern GPU architectures, modeled in Figure 2.4, mainly comprise SMs (aka CUDA Cores), interconnection networks, and memory partitions with high bandwidth.

GPUs employ a single-instruction-multiple-thread (SIMT) execution in their Streaming Multiprocessor (SM) units (Aamodt et al. (2018)). When a kernel launches, the Gigathread engine (thread block scheduler) schedules thread blocks to SMs by exploiting the Round-Robin policy. The SM has a large register file holding thread operands and saving from long latencies during context switches, instruction dispatchers, warp schedulers, and functional units. The total number of registers determines the number of active thread blocks, some of which may be issued to the same SMs. SMs fetch the warp instructions from the instruction cache, place them into instruction buffers, and then decode. The warp schedulers determine the next instruction group within a warp, and the instruction dispatching unit issues decoded instructions to the functional units by considering opcodes. SM's functional units involve LD/ST units, SP/DP/INT ALUs, specialized units for complex mathematical operations, and tensor cores. While ALUs perform arithmetic and logic operations depending on operand type, tensor cores conduct vectorial/matrices load-multiplication-store operations such as fused multiply and add (FMA) with mixed precision. The LD/ST unit is responsible for executing memory-type instructions. The LD/ST unit includes a memory access coalescing unit, which combines multiple memory requests accessing the same data into one to reduce overhead access overhead. An interconnection network connects SMs and memory partitions where DRAM and L2 cache (Last-Level cache) are placed. While the cores inside the same SM can access the L1 cache, all the cores can communicate via the L2 cache structure. Unlike the older GPUs, a distinct scratchpad memory (i.e., shared memory) resource no longer exists. Instead, the L1 cache, which can still be configured as scratchpad memory, is the on-chip data memory, and the compiler conducts possible optimizations with hardware support automatically after Volta architecture. Memory accesses, whether load or store, look on the L1 cache first, and then the L2 cache and DRAM partitions via the interconnection network for the desired data. Data access gets slower for memory instructions as moving down the hierarchy. It is worth noting that GPU architectures are continually evolving, with each new generation introducing enhancements in performance, power efficiency, and specialized features. The above overview provides a general understanding of GPU architecture, but specific details and terminology may vary based on the GPU model and manufacturer. During the thesis, we experiment with the QUADRO P4000, GV100, and RTX1650 GPUs based on Pascal, Volta, and Turing architectures, respectively. The experimental setup in the following chapters mentions architectural and resource specs for these GPUs.

### 2.3. Micro-architectural metrics

GPU developers analyze applications' behavior and hardware occupation by employing metrics obtained via profilers through real GPUs or simulators configuring a virtual GPU. Since we conduct our research based on NVIDIA products, we utilize Nsight System (NVIDIA (2022d)) and Compute (NVIDIA (2022a)) profiler tools to collect metrics. In Nsight Compute tool, *Compute Workload Analysis* section presents utilization of functional units, elapsed and active IPC, and issue rate of operands type such as FP16, FP32, which helps to obtain SMs activity. The tool gives many detailed memory metrics such as the number of requests with instruction types on caches and device memory, bandwidth throughput (data size/second), memory utilization in percent, and hit/miss rates of caches in *Memory Workload Analysis* section as in Figure 2.3. We first exploit Nsight Compute tool in the hardware occupancy metric extraction part of the soft error vulnerability study. Furthermore, we use the tool's CLI support in the approximate computing section to determine target CUDA applications, to which we will apply approximation methods by taking care of the memory utilization metrics.



Memory Workload Analysis Overview					
Memory Throughput [Gbyte/second]	116,74	(+0,00%)	Mem Busy [%]	50,26	(+0,00%)
L1/TEX Hit Rate [%]	75,30	(+0,00%)	Max Bandwidth [%]	61,04	(+0,00%)
L2 Hit Rate [%]	70,94	(+0,00%)	Mem Pipes Busy [%]	58,69	(+0,00%)

Figure 2.3. A memory workload analysis overview section snapshot taken from Nsight Compute tool.

Moreover, GPGPU-Sim (Khairy et al. (2020)), a state-of-the-art open-source GPU simulator modeling NVIDIA products, reports general performance and architecture occupancy metrics. GPGPU-Sim gathers the number of individual memory access on all L1 and L2 caches with hit/miss rates and DRAM banks and general occupancy metrics such as IPC, SM, and memory like the Nsight Compute. In addition, NVML (NVIDIA (2023c)) and AccelWattch (Kandiah et al. (2021)) support energy dissipation metrics for real GPUs and GPGPU-Sim v4.2, respectively.

### 2.3.1. GPGPU-Sim

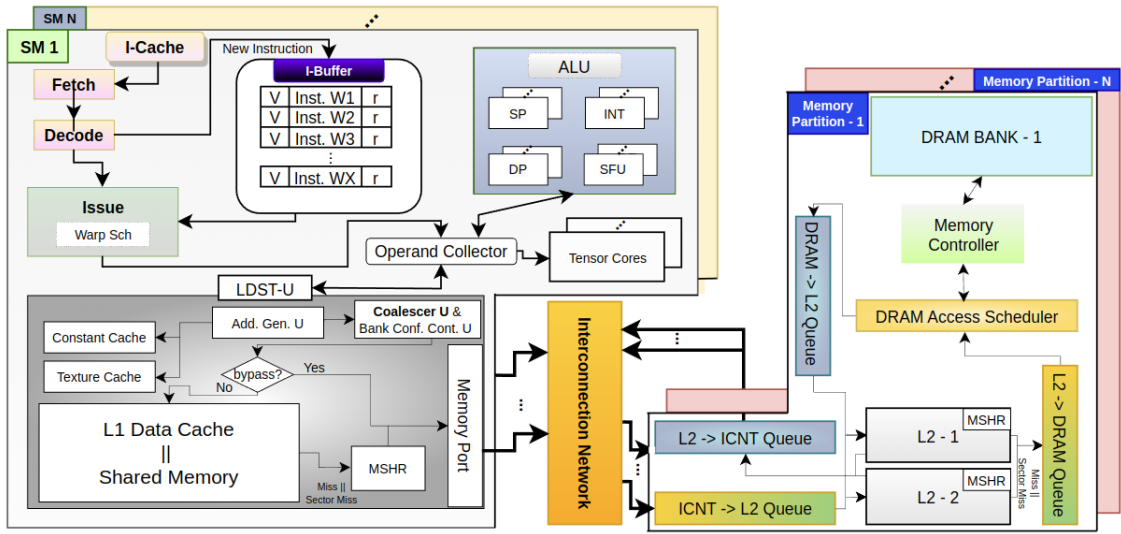


Figure 2.4. The simulator’s modern GPU modeling.

GPU researchers mainly exploit GPGPU-Sim (Khairy et al. (2020)) to conduct experimental studies targeting NVIDIA GPUs among the other simulators as it has evolved by tracking developments on the real hardware in the previous two decades, such as including tensor cores. Figure 2.4 displays the workflow of an NVIDIA-based GPU simulator executing either CUDA- or OpenCL-based applications. The simulator provides functional enabling developers to check the kernel’s functional correctness and performance-driven mode that simulates the kernel for the configured GPU in a cycle-accurate manner. The simulator runs PTX instructions extracted from the executable object by *cubobj-dump* since the machine decoding of SASS instructions onto the hardware is not publicly available. The simulator officially supports Fermi, Kepler, Pascal, Volta, and Turing architectures. However, computer architects may simulate any correctly configured GPU configuration on the simulator. In addition to cycle-accurate simulation, the AccelWatch power model, which supports Dynamic Voltage-Frequency Scaling (DVFS), measures energy dissipation for official GPUs with an accuracy above 90%. We develop the runtime performance and power monitoring tool by extending the simulator’s timing and power measurement models. Moreover, while utilizing the simulator to collect performance and instruction metrics in the soft error vulnerability study, we implement approximate computing approaches to the GPU architecture by modifying the simulator’s source code.

## 2.4. Soft Error Vulnerability

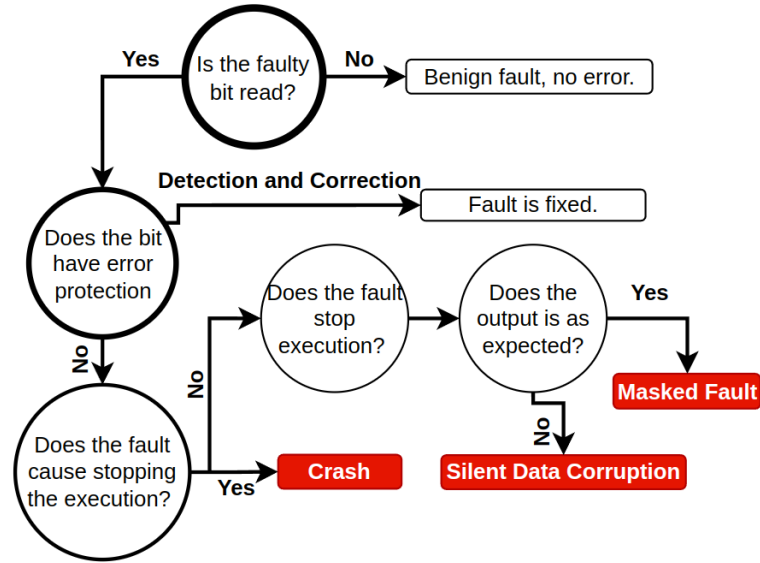


Figure 2.5. Fault detection and classification flowchart Öz and Karadaş (2022).

Soft errors are transient errors causing logic state changes on the hardware components (Mukherjee (2008)). Soft errors can occur for various reasons, such as cosmic rays and particle strikes, electromagnetic interference, voltage fluctuations, and thermal effects. The most prevalent way to evaluate the soft error vulnerability of an application is to perform fault injection (FI) experiments. To illustrate, one bit of one register is flipped at a random time during the execution of the application, and the output result is examined to observe the effect of the fault in our FI scenario. Figure 2.5 displays the fault detection and classification scheme. The outcome of the FI experiments can be classified into three categories: 1) *Correct Execution (Masked)*: The faults do not affect the output since the corrupted value is not used or overwritten in the remaining part of the program. 2) *Silent Data Corruption (SDC)*: The program terminates, but the program's output is not the expected output due to the corrupted data. 3) *Crash*: The program fails by terminating with an error code.

The FI tool developed by Öz and Karadaş (2022) pauses CUDA applications at runtime, performs fault injections into registers by flipping one selective bit, and resumes execution. FI tool conducts such a procedure uniformly 1000 times and classifies the error rates mentioned above for the CUDA applications. In this study, we enlarge the FI tests among the CUDA applications widely employed in literature and seek to capture the relationship between soft errors and GPU execution performance with the micro-architectural

metrics.

## 2.5. Approximate Computing

Approximate Computing (Anghel et al. (2018)) studies target to obtain high-performance and low-energy consumption-oriented computer systems by controlling error tolerance with diverse approaches. These mainly cover sub-circuits such as approximation-oriented ALUs Esmailzadeh et al. (2012), storage units by truncating the lower bits instead of storing exact data values or lowering refresh rate, Mittal (2016); Sampson et al. (2014), and software-level approximations such as memoization (Rahimi et al. (2013)) or loop perforation (Sidirolou-Douskos et al. (2011)). In addition to these approaches, as computer architecture research evolves, more complex system-level approximation studies obtain better system-level performance and lower energy consumption with tolerable errors. Whether domain-specific or not, system-based approaches intelligently identify performance bottlenecks, such as memory pipeline stalls at runtime, conduct approximations to boost performance, and reduce driven power without affecting output much. Since GPU devices are relatively complex systems, hardware-based approximation methods aim to solve local and overall performance problems synchronized. In the approximate computing side of error resiliency, we determine the memory performance bottlenecks by tracking micro-architectural metrics and offer a solution to detect and eliminate local temporal bottlenecks at runtime within a tolerable accuracy deviation.



## CHAPTER 3

# GPPRMON: GPU RUNTIME PERFORMANCE AND POWER MONITORING TOOL

Although GPUs have large computational power, their performance and energy efficiency may decrease mostly for workloads with memory-intensive characteristics. Besides several works that indicate the memory wall problem for the GPUs (Hong et al. (2022); Jain et al. (2019)), a significant amount of studies have clarified the different aspects of memory bottleneck points for GPU applications (Lew et al. (2019); O’Neil and Burtcher (2014)) and proposed various improvements (Jog et al. (2013); Koo et al. (2017); Vijaykumar et al. (2018); Zhao et al. (2019)). Additionally, the researchers have proposed methods for energy-efficient executions considering memory operations on GPU programs (Chen et al. (2014); Rhu et al. (2013)). Efficient execution of GPU programs involves considering both performance and energy improvements, but these two factors often contradict each other. Thus, making design decisions in this regard becomes crucial and complex, as it requires carefully evaluating the trade-offs between performance and energy efficiency (Guerreiro et al. (2019); Krzywaniak et al. (2022); Sun et al. (2018)).

Based on performance improvement and energy-efficiency concerns, reasoning a GPU application’s performance bottlenecks and interpreting power consumption during the execution requires more analytical measurements and rigorous evaluations. The evaluation of kernel performance and its relation to power consumption on GPUs often obscures valuable insights that can be gained from conducting a baseline analysis from multiple perspectives. To fully understand the execution of a GPU program, it is important to explicitly investigate each warp instruction as done in research on multi-core architectures. However, existing tools like NVIDIA’s GPU profiler (Nsight Compute Tool) and state-of-the-art GPU simulation tools (such as AccelSim and Multi2Sim) primarily operate at the kernel level and do not directly report dynamic performance, memory access behavior, and power consumption of GPU programs during runtime. Additional effort is required to collect relevant micro-architectural metrics from experiments for runtime performance and power consumption analysis. Researchers often develop their own target-specific monitoring solutions, leading to repetitive and redundant efforts in literature.

*GPPRMon*, a performance and power monitoring tool for GPU architectures,

monitors the program’s performance and driven power on hardware based on the dynamic behavior of memory accesses and thread blocks during the execution. Our simulation-based framework dynamically collects IPC, individual instructions’ issues/completions, and memory-centric microarchitectural metrics and reports achieved performance with energy dissipation for sub-components at runtime. Our visualization interface presents both spatial and temporal views of the execution, where the first demonstrates the performance and power metrics for each hardware component, including global memory and caches, and the latter shows the corresponding information at the instruction granularity in a timeline. *GPPRMon* enables users to perform a fine-granularity evaluation of the target execution by observing instruction-level microarchitectural features related to performance and reasoning dissipated power at runtime. To the best of our knowledge, this is the first work monitoring a GPU kernel’s performance by visualizing the execution of instructions for multiple user-configurable scenarios, relating memory hierarchy utilization with performance, and tracking energy dissipation at runtime. Our main contributions are as follows:

- We propose a systematic metric collection that keeps track of IPC per SM, instruction execution records for each warp to clearly observe issues/completions, detailed memory usage statistics per each sub-unit to interpret its effect on performance and power dissipation statistics for each GPU component at runtime with a configurable sampling cycle by extending the GPGPU-Sim v4.2 (newest, hereafter referred as the *simulator* throughout this chapter) framework.
- We design and build a visualization framework consisting of the following three perspectives. The visualizer interface concurrently runs with the simulator and displays the GPU kernel execution status by processing collected micro-architectural metrics at runtime.
  - i. **General View** displays the average IPC of active SMs, access statistics of active L1D and L2 caches, row buffer utilization of DRAM partitions, and dissipated power by the main components for an execution interval.
  - ii. **Temporal View** shows each thread block’s instructions issue and completion cycles separately at warp level. In addition to power statistics for the sub-components in an SM, we place L1 data cache access statistics to relate memory workload and the thread block’s performance within the execution interval.
  - iii. **Spatial View** demonstrates the access information for each on-chip L1 data

cache, L2 cache in each sub-partition, and row buffers of DRAM banks in each memory partition with average dissipated power distribution among the sub-units of the memory partitions in any execution interval.

- We demonstrate the potential usages of the GPPRMon framework and its visualizations by performing experiments for a memory-intensive graph and embedded application workloads to exemplify detailed performance and power analysis for the target GPU executions.

### 3.1. Related Work

AerialVision (Ariel et al. (2010)) visualizes runtime warp divergence, dynamic IPC, global memory access statistics, and active thread count with a mapping window between source code and exposed pipeline latency metrics of the kernel execution. GPPRMon is similar to AerialVision in terms of providing runtime performance metrics. While GPPRMon enables developers to dig into details of GPU execution within specific execution and architecture monitoring, such as instruction-level displaying at runtime, AerialVision profiles mostly micro-architectural metrics on a much longer execution scale. Moreover, the authors visualize overall GPU performance without per-component performance analysis, which may hide information about heterogeneous behavior inside kernel execution. Additionally, AerialVision does not support tracking energy dissipation.

Nsight Compute tool (NVIDIA (2022a)) runs a CUDA program on an NVIDIA GPU device and collects hardware occupation statistics on a kernel basis. Profiling with Compute tool eases interpreting the overall kernel performance and hardware utilization with too detailed micro-architectural metrics and a well-designed GUI. On the other hand, Compute tool does provide runtime monitoring of neither performance with hardware utilization statistics nor dissipated power for the kernel execution, while GPPRMon provides configurable tracking options for performance and power metrics during the runtime. Moreover, NVIDIA does not provide any detailed documentation for over 100000 micro-architectural metrics supplied by Compute tool.

TAU Performance System (Shende and Malony (2006)) is a performance profiling tool for hybrid parallel programs such as CUDA and OpenCL by intercepting the execution and calling metric collection functions. After gathering the results, TAU integrates them with data through instrumentation to display a performance picture of the execution. Like Nsight Systems tool (NVIDIA (2022d)), which provides performance

traces for high-level CUDA functions such as *cudaMemCpy*, TAU does not address detailed hardware usage and performance belongs to runtime execution. In addition, it lacks providing energy consumption of the hardware.

Daisen (Sun et al. (2021)) displays the overall occupancy on SM pipeline stages and memory components during the simulation. The authors aim to propose a performance-improving architecture by iterating an algorithm that benefits from the previously collected performance and hardware usage metrics. In other words, Daisen does not highlight the performance degrading points analytically. Instead, it addresses general bottlenecks and optimizes performance iteratively by re-configuring simulations. Similar to CHAMPVis (Pentecost et al. (2019)), their approach offers more systematic performance optimization points on GPU executions. On the other hand, GPPRMon focuses on monitoring the execution at the PTX instruction level and relating the execution with the memory occupancy during execution. Furthermore, while GPPRMon supports energy consumption metrics, Daisen does not provide them.

Candel et al. (2015) model a portion of the memory hierarchy of the AMD GPUs by investigating the behavior of MSHRs and coalescing unit for vector (warp for our case) memory requests by extending Multi2Sim. The authors find that the size and switching frequencies of MSHRs affect performance directly, especially for irregular workloads. Additionally, coalesced memory accesses, implemented in the GPGPU-Sim as well, reduce the repeated overheads of memory requests, significantly affecting performance for global memory accesses. Unlike the GPPRMon, their approach mostly focuses on modeling the behavior of the MSHR and coalescing unit, a portion of the memory hierarchy, of AMD GPUs instead of monitoring performance and energy consumption at runtime.

CHAMPVis (Pentecost et al. (2019)) offers a web-supported architectural performance monitoring tool that provides a hierarchical analysis of trends and bottlenecks for diverse application domains. The tool, which aims to analyze performance by evaluating metrics from a system view, generates predictive optimization speculations automatically in an application-specific manner. Different from GPPRMon, CHAMPVis does not employ any dissipated power tracking. Moreover, GPPRMon yields detailed execution statistics, whereas CHAMPVis highlights the overall trends.

Islam et al. (2019) present an analysis and visualization framework, namely DASH-ING, targeting exascale computing based on multi-core architectures. The authors provide user interaction models with some environment configuration options for analysis and visualization. Similarly, GPPRMon supports most of the official NVIDIA GPUs for multiple performance analyses and includes multi-functional metric collection and visu-

alization options depending on the user’s demand. However, we supply more low-level performance and energy dissipation analysis utility at runtime as described in *Temporal Overview*.

MemAxes (Giménez et al. (2018)) proposes an analysis framework on memory performance with various inspections for multi-core architectures. Its interface visualizes the analysis by obtaining performance metrics among different simulations and mapping them into a single visual. In addition, the authors offer memory utilization-based clustering research among the benchmark applications. On the contrary, GPPRMon allows evaluating GPUs performance and energy consumption in terms of performance and power dissipation through Spatial, Temporal, and General views during the runtime.

## 3.2. Methodology

GPPRMon enables monitoring and visualizing kernel performance and power consumption at runtime by evaluating hardware utilization. Figure 3.1 displays GPPRMon workflow consisting of two main stages: **1** Metric Collection, **2** Visualization. During any execution interval based on a user configuration, GPPRMon systematically calculates IPC rates of SMs, records thread block instruction’s issue completions within each SM, collects memory access statistics for each memory component, and tracks dissipated power among sub-hardware units. Parts **1-a** and **1-b** in Figure 3.1 demonstrate examples of the power and performance metrics, respectively. Part **2** reveals GPPRMon’s visualizer with three views to show general performance, memory access statistics, and instruction monitoring by processing the collected metrics. We build our framework on top of the simulator, which is compatible with official GPU configurations given as part of the simulator.

### 3.2.1. Micro-architectural Metric Collection

The *Micro-architectural Metric Collection* phase of GPPRMon, shown in Part **1** in Figure 3.1, describes the systematical records for performance and power metrics during the execution. The tool creates separate folders to collect each component’s statistics and distinct files for each sub-component with IDs. To reduce storage and access overheads during the kernel’s execution, we utilize CSV file format for recording metrics. GPPRMon provides multi-functional metric collection options, such as accumulat-

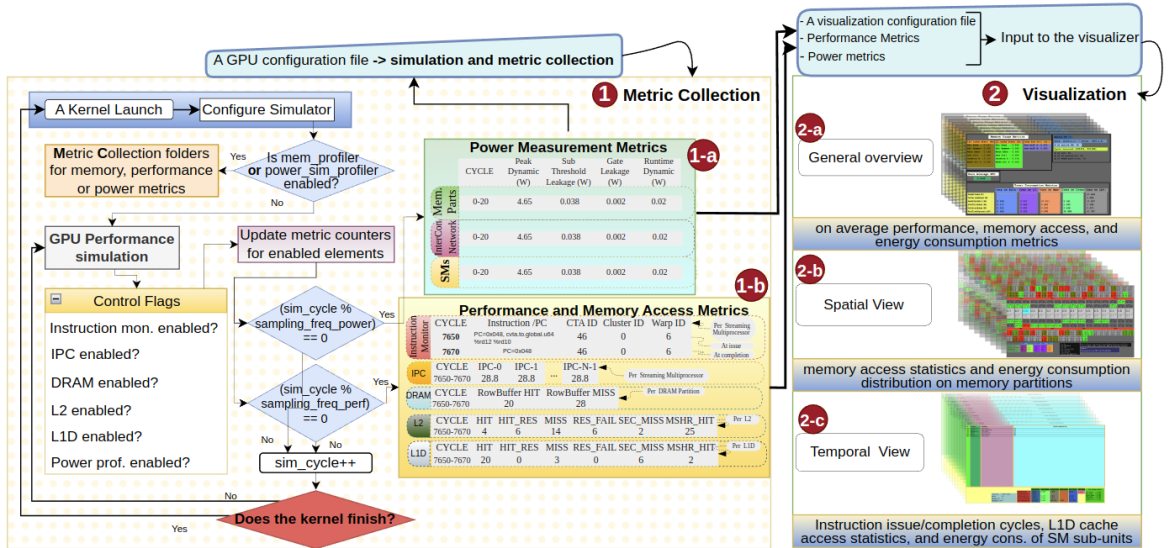


Figure 3.1. A workflow overview of *GPPRMon* framework.

ing all statistics during execution or discarding store operations from micro-architectural memory metrics. To provide multi-functional usage, we implement separate configuration options for power and performance metrics such that one must activate the runtime micro-architectural metric tracker mode for the desired metric in the configuration file. Concurrently to Part [1], the visualizer processes the data in CSVs and generates runtime monitoring visualizations, given as Part [2]. The following sub-sections explain the micro-architectural performance and driven power metrics as part of our tool.

### 3.2.1.1. Performance Metrics

**L1 Data and L2 Caches:** GPPRMon cumulatively tracks the cache access statistics and exports them at the end of the observation interval. A memory request's caches access status may be one of the following states: i) Hit: Data resides on the cache line; ii) Hit Reserved: The cache line is allocated, but the requested data is still on-flight; iii) Miss: Cache line, in which the request looks for data, causes many sector misses, result in a bigger dirty counter than the threshold, and the data is evicted from the cache line; iv) Reservation Failure: One of a line allocation failure, MSHR entry allocation failure, merging an entry with an existing in MSHR entry failure, or no-space on miss queue to hold new requests may cause a reservation failure; v) Sector Miss: A memory request cannot find the data in the sector of a cache line (i.e., a sector is 32B, whereas a cache line is 128B); vi) MSHR hit: When the upcoming request's sector miss has already been

recorded, and request can merge with the existing entry, MSHR hit occurs.

While hits infer that memory requests cause tolerable latency, misses generally refer to bigger latency for memory operations and increasing traffic on the lower level of the hierarchy. Handling intensive misses requires detailed analytic observations to deduct behavioral interpretations between application performance and cache utilization. For example, a reservation failure causes a memory pipeline stall meaning that all the sub-architecture is locked till the reservation failure is recorded by the responsible miss handlers. If the architectural performance suffers from the inevitable misses due to data sparsity, optimizing the cache such that it acts depending on runtime miss rate may save from the memory pipeline stalls by keeping reservations failure in a tolerable amount. Hence, cache access statistics reveal many insights about the performance and might also give clues about target solutions.

**Row Buffers of DRAM Banks:** Since row buffer hits save from the additional activation latencies of accessing DRAM cells among the global memory accesses, we support separate metric collections for the row buffers in each memory partition. Our approach supports observing row buffer access metrics for either just load or load and store operations together. As a result, GPPRMon enables tracking row buffer utilization for the memory accesses missed on L2 caches at runtime.

**IPC:** GPPRMon provides the per-sampling IPC rates for each SM to evaluate the overall performance of the SMs separately. For example, GV100 SMs include four SP ALUs with four pipe depths, as presented in Figure 2.4. When a thread block occupies all SP-ALU units, assuming an operation takes one cycle, the ideal IPC should be sixteen without taking care of other functional units. However, IPC mostly oscillates during the execution based on the utilization quality of functional units on SM. One can conduct IPC comparisons at runtime and investigate the reasons for the variation.

**Instruction Monitor:** The instruction monitor utility records the issue and completion cycles of instructions with opcode, operand, and PC at warp level for each SM separately. Since GPUs execute instructions for multiple threads concurrently by a common PC within a warp, we monitor issues and completions at the warp level. While the first row of the *Instruction Monitor* in Part [1-b](#) shows the issue statistics, the second displays the completion. Even if the dispatcher unit may issue the same instruction multiple times for a warp, we are sure that any two instructions, of which CTA\_ID, SM\_ID, local Warp\_ID, and PC are the same, cannot change the issue/completion sequence. Hence, we can obtain the correct issue/completion matching for each instruction.

### 3.2.1.2. Power Metrics

The comprehensive results of the dissipated power on GPU yield an analytical observation that gains significance. Therefore, we develop GPPRMon to systematically collect the power distribution on the sub-units of SMs, memory partitions, and the inter-connection network besides the performance. GPPRMon classifies SM’s power distribution as execution units, the register file, the beginning of the pipeline stages, functional units, and LD/ST units involving the on-chip L1D cache. Furthermore, GPPRMon distributes and displays DRAM power metrics among the front-end-engine and transaction engine of memory controllers (MCs), the physical connection between MC-DRAM.

We implement the power metric collection service on top of the AccelWatch (Kandiah et al. (2021)), which includes modeling for dynamic-voltage frequency scaling (DVFS) and is built through McPAT (Li et al. (2009)). GPPRMon assures the following measurements during runtime for each component apart from idle SM: *Peak Dynamic*(W), the maximum momentary power within the interval, *Sub-threshold Leakage* (W) and *Gate Leakage*(W), the leaked power (due to current leakage) from the junctions of MOSFETs, and *Runtime Dynamic* (W), the total consumed power. Moreover, GPPRMon supports collecting power metrics either cumulatively or distinctly for each sample, starting from a kernel’s execution. In addition to collecting power consumption for each interval independently, GPPRMon supports accumulating power consumption measurements by starting from a kernel’s execution.

### 3.2.2. Visualization

By processing the collected metrics in Part [1], GPPRMon depicts performance and power dissipation with three perspectives at runtime, as represented in Part [2] in Figure 3.1, and enables pointing out a detailed investigation of hardware utilization.

- i. **General View**, Part [2-a], presents the average memory access statistics, the overall IPC of GPU, and dissipated power among the major components with application- and architecture-specific information;
- ii. **Spatial View**, Part [2-b], shows the access statistics of all the memory units on the GPU device memory hierarchy and dissipated overall power among the memory partitions by enabling the monitoring of the entire GPU memory space;



On Average Memory Access Statistics						
<b>L1D Cache Stats (Av)</b>		<b>L2 Cache Stats (Av)</b>		<b>DRAM Row Util. (Av)</b>		
Hit Rate	0.003	Hit Rate	0.312	Row Buffer H	0.383	
Hit Reserved R	0.001	Hit Reserved R	0.000	Row Buffer M	0.617	
Miss Rate	0.038	Miss Rate	0.464	Kernel ID: 0		
Reserv. Failure R	0.944	Reserv. Failure R	0.000	Cycle Interval: [55000, 56000]		
Sector Miss R	0.013	Sector Miss R	0.223	Grid:(1784,1,1) Block:(256,1,1)		
MSHR Hit R	0.008	MSHR Hit R	0.005	# of active SMs: 80		
<b>Average IPC on SMs : 1.08</b>						
<b>Dissipated Power</b>	InterCon.	Net	L2	Mem Part.	SMs	GPU
Peak Power (W)						185.63
Total Leakage (W)						17.346
Peak Dynamic (W)	0.338		4.687	137.55	25.704	168.264
Sub-Threshold Leak (W)	0.067		0.138	1.316	13.474	14.995
Gate Leakage (W)	0.011		0.013	0.016	2.168	2.352
Runtime Dynamic (W)	64.618		2.537	823.184	205.174	1095.513

Figure 3.2. *General View* displaying average performance and power consumption measurements at runtime.

- iii. **Temporal View**, Part [2-c](#), demonstrates instruction execution statistics with activation intervals at warp-level for user-specified thread blocks, L1D cache access characteristics, and power distribution among the sub-components of SMs.

GPPRMon includes three visualization configuration options and an interval of sampling cycle to divide runtime execution into portions. Since *Temporal View* may require scanning of many data for all thread blocks, especially in large architectures, GPPRMon provides a *Temporal View* option among the thread blocks determined with IDs. We explain the remaining configuration options of the GPPRMon tool in detail with introductory test results on our GitHub<sup>1</sup>. Depending on the configuration, GPPRMon starts tracking collected metrics for each kernel and systematically saves images in PNG format per execution interval.

Figure 3.2, an example of the *General View*, presents the overall measurement of *Kernel 0* in the Sparse Matrix-Vector Multiplication (SPMV) (Xu et al. (2019)) program on GV100. It displays average memory access statistics of active L1D caches, L2 caches, and DRAM banks; average IPC value among the active SMs; dissipated power on major sub-GPU components within an execution interval. The view includes grid (i.e., 1764 thread blocks) and thread block dimensions (i.e., 256 threads per block) with the number of actively used SMs so that the users can realize issue mappings of thread blocks to the SMs. To illustrate, Figure 3.2 shows that *Kernel 0* executes with the IPC rate of 1.08 and

<sup>1</sup><https://github.com/BT-MasterThesis-2020-23/GPPRMon>

utilizes the memory hierarchy inefficiently due to high miss and reservation failure rates in the interval of [55000, 56000] on GV100 whose SMs support concurrent execution of 2048 threads. Considering that GV100 SMs contain 16 amounts of each SP/INT/DP ALUs, SFUs, and Tensor Cores, we can notice the low performance. The ideal IPC can be much more than 1.08 with 640 active thread blocks (8 thread blocks per SM) for the given interval. The long-latency memory operations may slow down instruction completion and cause a reduction in IPC for *Kernel 0*. Moreover, memory partitions cover 75% of total power, which validates that SMs mostly stay idle for the displayed execution portion in Figure 3.2. Our *General View* supports two additional visuals that show the time spanning of memory access statistics and the relationship between IPC and power metrics at runtime as in Figure 3.5 and 3.8 (examples given as part of our case studies), respectively.

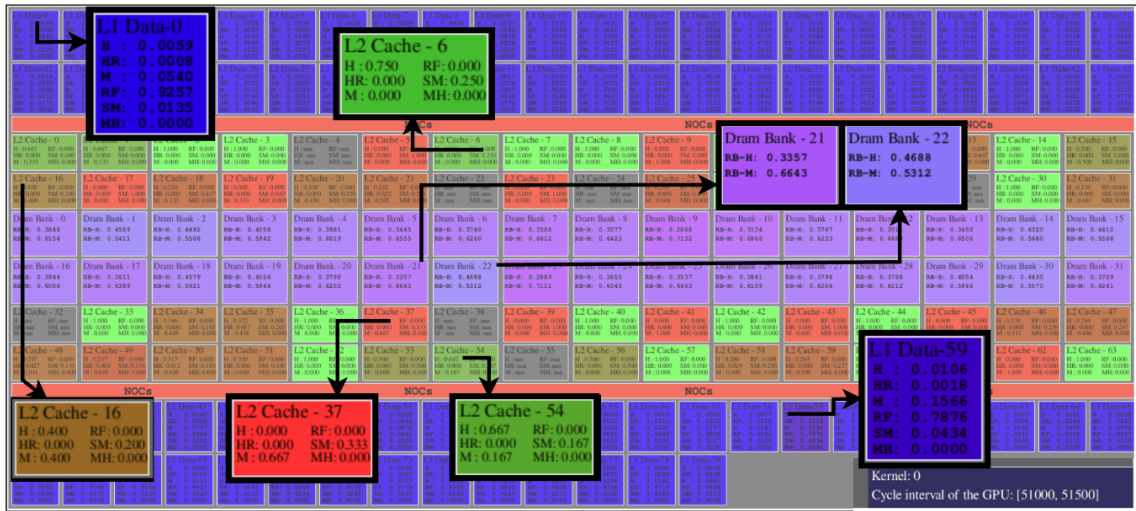


Figure 3.3. *Spatial View* revealing memory access statistic at runtime.

Figure 3.3, an example of our *Spatial View*, shows the memory access statistics across the GPU memory hierarchy on L1D caches, L2 caches, and DRAM. On caches, the green refers to hit and hit reserved accesses concentration, the red indicates miss and sector miss intensity, and the blue states many reservation failures through miss queues or MSHR. Similarly, DRAM bank pixels are colored with a mixture of red and blue to specify the row buffer misses and hits, respectively. *Spatial View* demonstrates data access statistics per memory unit, resource quantities, architecture types, and driven power on memory partitions (all features can be found in our GitHub repository). To detail the view, we zoom in on some memory units in Figure 3.3, which presents statistics in the cycles of [51000, 51500] belonging to the *Kernel 0* of SPMV. In that interval, distinct L1D caches behave similarly, such that almost all L1D caches turn blue due to reserva-

tion failure concentration. To illustrate, the reservation failure rate of which miss access requests cannot be transferred to lower memory levels is 0.92 and 0.78 on L1D-0 and L1D-1, respectively. On the contrary, statistics on L2 caches imply heterogeneous data accesses caused by multiple reasons, such as data sparsity. L2 Cache-6 and -54 bring hit rates of 0.75 and 0.67, and L2 Cache-37 causes 0.67 miss and 0.33 sector miss rates. Furthermore, the *Kernel 0* does not exploit row buffers of DRAM banks since many turn to purple in the same interval. Gray color among the units regards no access occurring on the Spatial View during the corresponding execution interval.

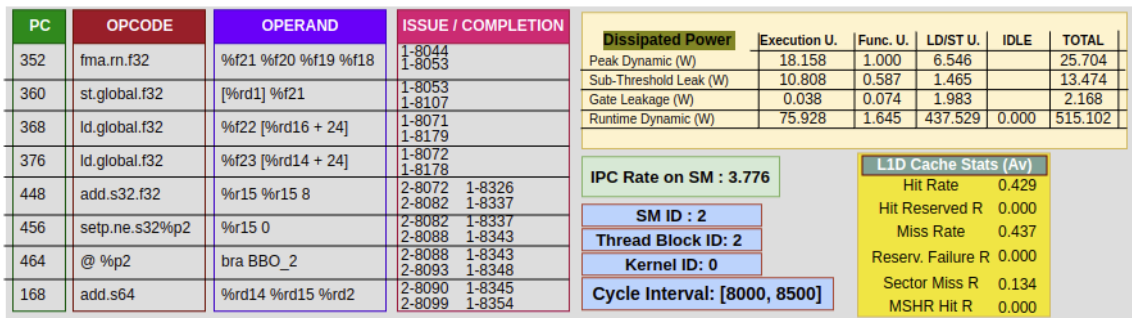


Figure 3.4. *Temporal View* monitors instruction executions together with the performance and power consumption of the corresponding SM.

Figure 3.4, an example of our *Temporal View*, displays a thread block's execution statistics at warp-level with SM's ID and IPC, L1D cache statistics, and dissipated power of core components in any configurable execution interval. It presents each warp's PTX instruction sequence, with opcodes, operands (source/destination registers and immediate values if they exist), and the program counter (PC) shared through all threads. The *Issue/Completion* column exposes the execution start and writeback times of warp instruction segments within any thread block. For instance, Figure 3.4 reveals the execution monitoring of the 2nd thread block on SM2 for the *Kernel 0* in the cycle range of [8000, 8500]. The instruction dispatcher unit issues two SP loads with PC=368 and PC=376 at cycles 8071 and 8072, and they are completed at cycles 8179 and 8178, respectively. *Temporal View* allows tracking execution duration per instruction in this manner. Considering that L1D cache hits should be completed lower than 25 cycles, these load instructions lasting above 100 cycles are the misses or sector misses on the L1D cache of SM2. In addition, the view enables us to relate IPC, instruction statistics, and power metrics. The fact that the rate of memory instructions to all instructions is 0.37 and inefficient use of the L1D cache within the given interval significantly degrades the IPC on SM2. Furthermore, the

LD/ST unit dissipates nearly 92% of SMs total power in the corresponding interval because of the pressure on the L1D cache. As a result, one can analyze the data locality in a multi-perspective by utilizing access statistics of caches and row buffers on *Spatial View* and tracing the issue/completion times of memory operations on *Temporal View* during the runtime.

### 3.3. Case Studies

#### 3.3.1. Performance Bottleneck Analysis and its Power Impacts for a Memory-Intensive Workload

We evaluate CUDA implementation of the *Page Ranking* (PR) algorithm given in Gardenia Benchmark suite Xu et al. (2019) to analyze a memory-bound GPU program and irregular memory access statistics with GPPRMon. PR algorithm assigns weights to graph nodes describing the relative importance among them. We perform the experiment on Volta architecture-based GV100, commonly used in HPC systems and GPU architecture research.

Table 3.1. GV100 Based on Volta Architecture Configuration Specifications.

Streaming Mutlipro- cessor Specs (80)	Reg. bank size, # of reg. bank	65536 32-bit regs., 16 reg. banks
	SP, SF, DP, INT, TC, LD/ST (WB-Depth)	4, 4, 4, 4, 4, 1(8)
	Warp Scheduler	4 (LRR) per SM
	L1D Cache, #of banks, latency, line size	128KB, 4, 20 cycles, 128B
	L1I Cache, #of banks, latency, line size	128KB, 1, 20 cycles, 128B
Memory Partition Specs (32)	L2 Cache, #of banks, latency, line size	96KB, 2, 160 cycles, 128B
	DRAM, #of banks, latency (after L2)	1GB, 16, 100 cycles, 128B
	DRAM scheduler	First-ready, first-come first-service

*SP*: Single Precision, *SF*: Special Functional, *DP*: Double Precision, *INT*: Integer, *TC*: Tensor Core, *LD/ST*: Load / Store, *WB*: Write back

Table 3.2 presents the PR algorithm’s performance overview on the GV100 with the memory access statistics using the simulator’s naive version. The algorithm iterates

with the *Contribution Step* (K0), *Pull Step* (K1), and *Linear Normalization* (K2) kernels throughout the execution, and the number of iterations may vary depending on the data size. At the beginning of PR, *K0* causes a high miss rate on caches due to compulsory misses. Since the required data mostly fit into the L2 cache for *K0*, we do not observe row buffer locality information among DRAM accesses. In other words, the first L2 misses do not access the same row of DRAM banks within any memory partition. Over and above these, total elapsed cycles indicate that *K1* dominates the execution at 99.7%. Hence, accelerating *K1*, whose IPC and occupancy values are much lower than *K0* and *K2*, directly affects the overall performance and power consumption. While the average miss and reservation failure rates imply pollution on L1D caches, a comparably small miss rate occurs on L2 caches with low reservation failures. The row buffer locality is around 0.65 among the DRAM accesses for *K0*. Furthermore, the simulator’s performance mode does not distinguish memory usage of load and store operations; thus, Table 3.2 represents the cumulative memory occupancy statistics.

Table 3.2. Page Ranking kernel performance statistics.

Kernel	GPU IPC	GPU Occupancy	L1D		L2		DRAM	Total Cycle
			Miss Rate	RF. Rate	Miss Rate	RF. Rate	RB. Loc.	
Contrib K0	715.59	82.76%	1.000	0.819	0.333	0.0	-nan	8670
PullStep K1	3.007	5.55%	0.584	0.400	0.156	0.011	0.658	8677889
LinNorm K2	1297.6	77.108%	0.501	0.285	0.457	0.001	0.724	11718

*RF*: Reservation failure, *RB Loc.*: Row buffer locality

GV100 includes 870GB/s bandwidth migrating 217.5 Giga SP float to the SMs and 14.8-SP/7.4-DP TFLOPS peak computational power NVIDIA (2018b). That is, the time for a load complies with the execution of 68 SP float operations on SMs, ideally (i.e., without L1D and L2). On the contrary, the fact that *K1* has a memory instruction intensity of around 0.2 in PTX code validates intrinsic memory workload bounds *K1* performance. Not only *K1* is memory-bound, but also inefficient memory usage severely impacts performance. In this manner, we investigate runtime performance limitations with execution statistics and hardware specs through GPPRMon, for *K1*. We evaluate the PR algorithm through GPPRMon with the execution sampling intervals as 100, 500, 1000, 2500, 5000, 10000, 25000, 50000, and 100000 cycles (entire data can be found in our GitHub). Increasing the visualization frequency hides the key observations because the

execution converges to average micro-architectural statistics. Hence, we identify runtime performance-degrading factors at low-frequency execution snapshots.

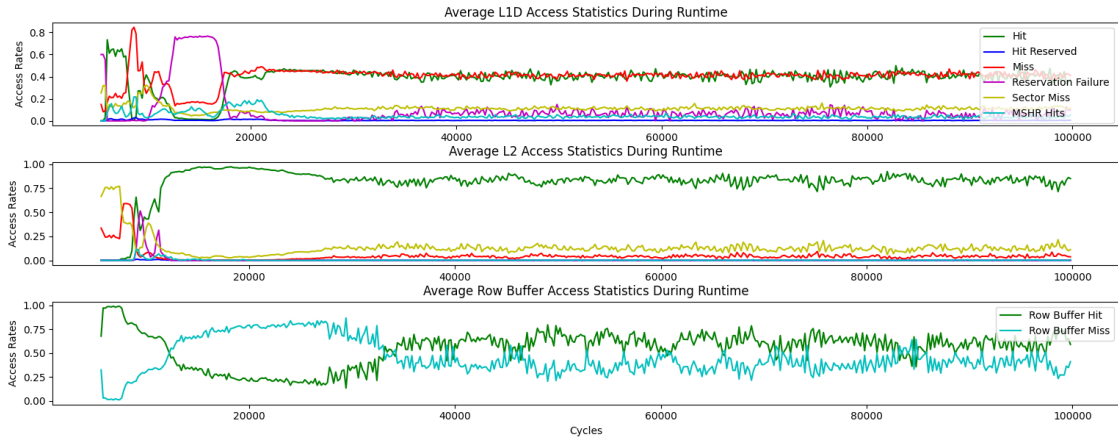


Figure 3.5. Average memory access statistics in the cycle range of [5000, 100000].

While the average IPC is around 0.3 for *K1, General View* results where visualization frequency is 500 cycles indicate that IPC oscillates in the range of [0.1, 9.54], with the highest peak of 53.44. Figure 3.5, which is part of *General Overview*, shows average access statistics on memory units in [5000, 100000] cycles. After caches warm-up (i.e., 10000 GPU cycles), while the average miss rate on L1D caches oscillates in [0.14, 0.51], sector misses, which the simulator does not provide separately, vary in [0.05, 0.31] with without accumulating statistics. These observations reveal that data pollution exists on the L1D caches, which breaks exploiting cache locality during runtime. For example, *K1* does not benefit from the spatial locality on the L1D cache since the MSHR hits among the sector misses oscillate slightly in [0.03, 0.08] during the execution without covering outlier statistics. Furthermore, the overall hit rate on L2 caches is quite high according to *General View* when we use the web-Stanford dataset Leskovec et al. (2009a), whose graph size is five times larger than the L2 cache capacity. While the performance metrics in Table 3.2 hides the statistics of L2 as it counts misses before warming up at kernel launch, the actual L2 hit rate oscillates in [0.82, 0.95] during the runtime with sampling per 500 cycles as displayed in Figure 3.5. Additionally, the row buffer hits and misses vary in [0.2, 0.85] in an unstable manner which verifies data sparsity throughout the execution. Moreover, *General View* visuals point out that GPU dissipates power in the range of [3076W, 46738W] within every 10000 cycles. The SMs dissipate most due to the rich number of on-chip gates in functional units and register. Elapsed total power increases in parallel with rising SM occupancy and IPC.

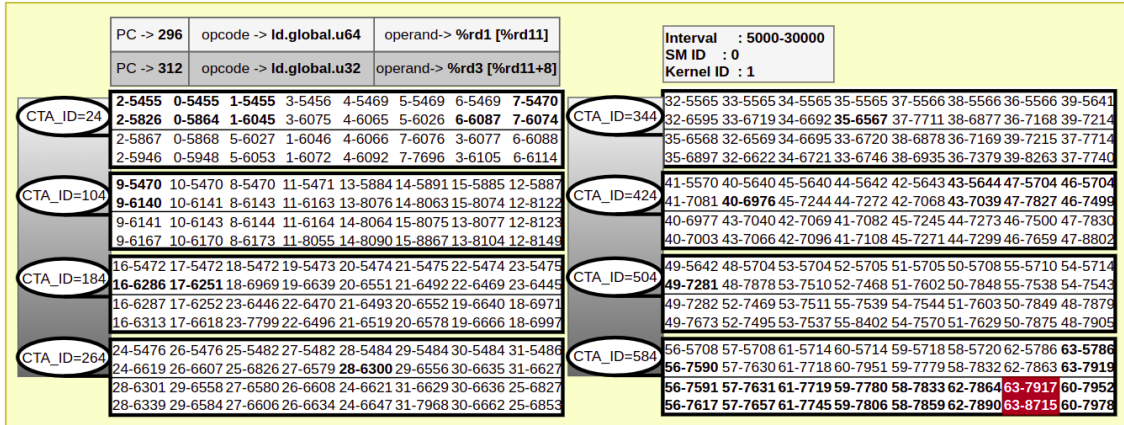


Figure 3.6. Instruction monitoring for the load instructions on SM0 in the cycle range of [5000, 30000].

The *KI* comprises 1102 thread blocks (TB<sup>2</sup>), with 256 threads per block, and GV100 support 16 TBs to run simultaneously on each SM. Register resources per SM limits the number of TBs such that only 8 can execute concurrently during the execution of *KI*. Hence, the GigaThread engine initially assigns 640 TBs to the SMs in a round-robin fashion (RRB), and the remaining 462 TBs are sequentially launched later. *KO*'s last TB executes on SM53, and *KI* issues TBs starting from SM54 due to RRB policy. Similarly, SM0 starts with the execution of TB-24 and allocates the TBs sequentially, starting from TB-104 and continuing with TB-184 until it keeps eight.

Figure 3.6 displays the instruction issue/completion cycles of 8 TBs running on SM0, and the first and second lines point to the load instructions whose PC=296 (loads DP) and PC=312 (loads SP), respectively. We merge multiple snapshots of *Temporal View* in Figure 3.6 belonging to TBs in SM0 to evaluate the performance of load instructions. Figure 3.7, a snapshot of *Spatial View*, shows the memory access statistics of representative components within the same interval. We follow the access statistics on the memory hierarchy with Figure 3.7 and relate the observations with the issue/completion duration of loads in Figure 3.6.

After the kernel launch, each thread collects thread-specific operands from parameter memory which takes 250-450 cycles to obtain pointers for target data addressed with the thread's private registers. The warp schedulers dispatch the load instructions pointed to by the PC=296 (*ld.global.u64*), and all eight warps of TB-24 start executing the instruction after Cycle 5455. Furthermore, Figure 3.6 reveals that SM dispatches load instructions from the remaining TBs in the interval of [5470, 5786] after issuing the load

<sup>2</sup>Throughout the case study section, thread block term is abbreviated as TB

instructions of TB-24. Figure 3.7 reveals that no access occurs on some L1D caches, and none of the L2 caches and DRAM banks are accessed during the preparation time in [5000, 5500] in part [1]. None of the data brought to the L1D cache of SM0 by the warps of TB-24 after Cycle 6087 (Warp 6) enables the early completion of the instruction pointed with PC=296, belonging to the TB-104, TB-184, TB-264, TB-344, TB-424, TB-504, and TB-584. We highlight this observation with the bold fonts representing the earliest completion times within each TB in the second row of the first instructions. Additionally, a high reservation failure and no MSHR hit rates on the L1D cache of SM0 in Part [2] of Figure 3.7 confirms that the locality utilization between TBs is dramatically low for the first load. If the locality utilization existed on the L1D cache, we should have observed larger MSHR hit rates and completion of the same instructions just after Cycle 6087. [2,3,4,5] parts in Figure 3.7 reveal that reservation failed requests pointed by PC=296 cause miss on L2 caches without MSHR merging. Thus, memory requests of the same instruction from different SMs cannot benefit locality on the L2 cache partitions and cause more traffic in the memory hierarchy.



Figure 3.7. Memory performance overview in the cycle range of [5000, 9500].

Parts [3,4,5,6] in Figure 3.7 reveal that the access status of L1D mostly turns to the



hit after Cycle 6000. Unlike the load instructions at PC=296, loads of threads at PC=312 (*ld.global.u32*) usually hit; thus, the second line for each TB in Figure 3.6 shows that the completion takes much fewer cycles for the loads at PC=312. To illustrate, while TB-504 completes the first load instructions within 2133 cycles, it takes 26 cycles for the second instructions, apart from Warp 63, whose requests cause misses on both the L1D and L2 caches. While the loads at PC=296 complete the execution in the range of [350, 2250] cycles, the loads at PC=312 take less than 50 cycles for most of the warps due to the increasing hits on the L1D cache. However, loads at PC=296 delay the issue of second load instructions with excessive latencies. One may follow that the overall miss rate on each L1D cache reduces by tracking the activation cycles of the second load instructions in 3.6 and investigating accesses statistics in [3,4,5,6] parts of Figure 3.7. Furthermore, considering the L2 latency is over 150 cycles, warps within the same TB use the data previously brought to the L1D cache. Therefore, load instructions issued by different TBs may exploit the data locality arising from other TBs loads on the L1D cache.

GigaThread engine issues the remaining TBs beginning with the 641st to SMs after 55000 cycles. With the observation that a TB occupies 50000 cycles on an SM for the *Web-Stanford* graph, which can easily fit into DRAM, the schedule of any waiting TB delays around 2000 cycles. Such delays affect the performance of a TB by 4%, besides causing inefficient usage of memory and degrading the overall performance of K1. In this manner, an approach such as adaptive TB scheduling by throttling the LD/ST unit issue amount depending on the access statistics of caches can eliminate the performance bottlenecks and increase the overall performance.

The dissipated power in Table 3.3 obtained with GPPRMon metric collection tool coincides with performance observations. Registers load thread-specific data (from the parameter memory) such as thread ID during the 5000-5500 cycles, causing higher power consumption on SMs. In the following 4500 cycles, the memory partition's power dissipation gets more than the SMs. Intense memory operations and pressure on the on-chip L1D cache increase the consumed energy on LD/ST units. The remaining functional units do not cause too much energy dissipation as they stay mostly idle after Cycle 5500. According to observations via GPPRMon in Table 3.3, DRAM contains most of the dissipated power in the memory partitions with intense usage of high-bandwidth NVIDIA (2018c). As a result, when irregular memory accesses exist, which stalls the memory pipeline and keeps SMs idle, released power gets lower along with performance. The fact that nearly half of the load instructions operate at a quarter speed compared to the ideal performance infers that the inefficient access behavior on memory usage wastes.

Table 3.3. Dissipated power measured during K1’s execution on GV100 in the cycle range of [5000,10000].

Cycles	Streaming Multiprocessor					SM Total
	Exec. Units	Funct. Units	LD/ST. Unit	SM Idle	SM Total	
5000, 5500	2637.57	54.30	35.67	23.75	2751.30	
5500, 6000	597.03	6.31	860.01	0	1463.69	
6000, 6500	614.408	12.41	399.89	0	1026.71	
6500, 7000	708.63	14.38	464.29	0	1187.312	
7000, 7500	686.78	13.90	463.94	0	1164.62	
7500, 8000	795.81	16.26	487.37	0	1299.44	
8000, 8500	543.02	10.19	335	0	888.21	
8500, 9000	354.47	5.58	249.28	0	609.34	
9000, 9500	474.91	5.34	455.42	0	935.67	
9500, 10000	446.76	4.76	475.46	0	926.99	

Cycles	Memory Partition					NoCs	Mem. Total
	MC FEE	PHY Int.	MC TE	DRAM	L2		
5000, 5500	3.74	8.17	4.59	0	0	0.67	16.51
5500, 6000	177.24	17.77	9.39	557.15	3.36	26.92	764.92
6000, 6500	56.06	31.28	16.14	1346.38	3.06	92.60	1452.94
6500, 7000	65.52	31.40	16.20	1354.31	3.05	94.17	1470.52
7000, 7500	65.57	31.37	16.19	1354.91	3.07	94.39	1471.15
7500, 8000	69.97	29.68	15.34	1264.33	3.70	96.66	1383.07
8000, 8500	60.43	30.52	15.76	1341.36	4.4	124.73	1451.96
8500, 9000	51.96	31.10	16.05	1362.57	19.04	148.04	1480.74
9000, 9500	80.23	27.75	14.38	1096.85	66.13	216.84	1284.35
9500, 10000	78.17	23.83	12.42	843.20	41.01	153.79	968.65

*MC*: Memory Controller, *PHY Int*: Physical Interface, *FEE*: Front-end engine, *TE*: Transaction Engine, *NoC*, Network-on-Chip.

### 3.3.2. Performance-Power Analysis of an Embedded Application

Embedded applications targeting artificial intelligence, computer vision, and advanced graphics require high computational power, and Jetson AGX Xavier of NVIDIA provides a System-on-Module (SoM) that meets these demands with a Volta-based GPU.

According to NVIDIA’s specifications NVIDIA (2019), we configure the AGX Xavier, which contains an 8GB/16GB unified memory with a high-bandwidth interface to the GPU, and a 512KB shared L2 cache exists in the memory hierarchy. Its GPU includes 8 SMs involving a 128KB on-chip cache per SM. Since the AGX Xavier is based on the Volta architecture, many of the power measurement approaches for GV100 are also applicable to its GPU component. Moreover, AccelWatch supports the changes in the memory hierarchy, such as limiting memory bandwidth and changing the L2 cache structure to track power measurements. Therefore, our configuration represents the AGX Xavier for timing and power measurements with the accuracy given in (Kandiah et al. (2021); Khairy et al. (2020)).

Table 3.4. Fast Fourier kernel performance statistics.

Kernel	GPU IPC	GPU Occupancy	L1D		L2		DRAM		Total Cycle
			Miss Rate	RF Rate	Miss Rate	RF Rate	RB Loc. (all)	RB Loc. (LDs)	
FFT K0	31.76	62.52%	0.67	0.92	0.65	0	0.21	0.89	990276
FFT K1	100.22	80.25%	0.1	0.75	0.20	0	0.33	0.92	235390
FFT K2	49.20	85.28%	0.1	0.837	0.21	0	0.41	0.92	239755

$K$ :-  $i$ -th kernel,  $RB$ : DRAM Row buffers.

We execute the CUDA version of the Fast Fourier Transform (FFT) application from the GPU4S embedded benchmark suite Kosmidis et al. (2020). Table 3.4 presents the overall performance metrics for the execution of the first three kernels belonging to the FFT with 2 billion float data. A maximum of 8 TBs, which includes 256 threads per TB, can concurrently execute on the SMs since the register file limits the number of TBs.

FFT completes the execution by launching the same kernel sequentially through the entire data. Since the kernels other than the  $K0$  occupy the hardware similarly, they result in similar performance results as in the metrics for  $K1$  and  $K2$  as deduced by Table 3.4. Moreover, FFT fits better in distributing the workload to GPU and utilizing resources in a balanced way. Momentary observations during runtime mostly overlap with the overall execution behavior for kernel basis because of the smooth workload behavior. Thus, we focus on explaining the relationship between performance and power consumption for  $K0$  by utilizing the results in *General View* of the GPPRMon.

Figure 3.8 displays  $K0$ ’s IPC and power measurements sampled without accumu-

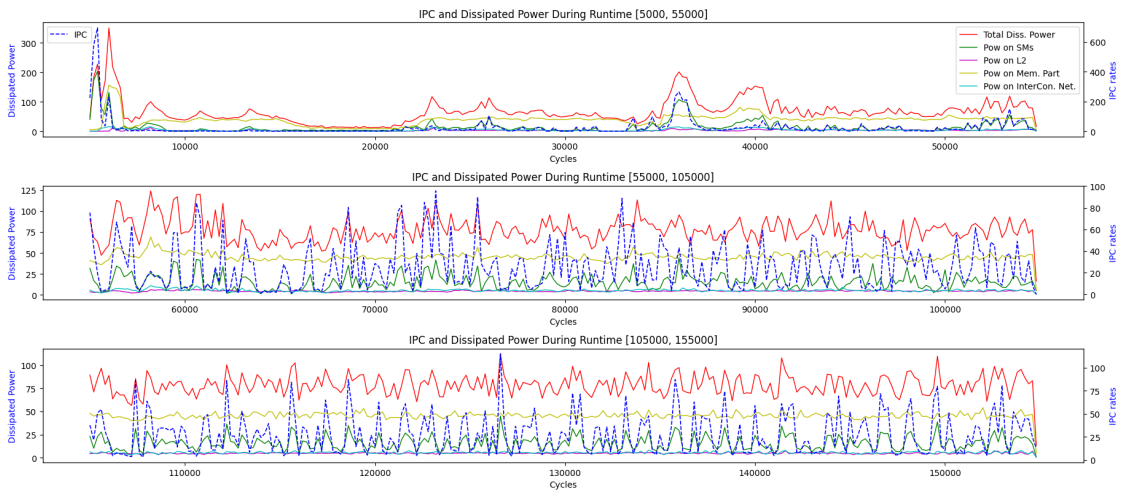


Figure 3.8. IPC and dissipated power metrics throughout the [5000, 155000] cycle interval of Kernel 0.

lating at runtime. To avoid losing observation details related to IPC and power metrics of the kernel, we report the relationship in three execution intervals. During the cycles [5000, 6500], the dissipated power per execution cycle and IPC values exceed the overall results since SM registers will be written back with thread-specific identifier data, causing a high activation on register files and functional units (i.e., parameter memory). Figure 3.9, GP-PRMon’s *General View*, displays four loads, three data movements, and one multiply-add instruction executed by 16384 threads concurrently (with 64 TBs where each contains 256 threads) in that interval.

PC	OPCODE	OPERAND
0	ld.param.u64	%rd1 [_Z21binary_reverse_kernelPKfPfli_param_0]
8	ld.param.u64	%rd2 [_Z21binary_reverse_kernelPKfPfli_param_1]
16	ld.param.u64	%rd3 [_Z21binary_reverse_kernelPKfPfli_param_2]
24	ld.param.u32	%r2 [_Z21binary_reverse_kernelPKfPfli_param_3]
32	mov.u32	%r3 %ntid.x
40	mov.u32	%r4 %ctaid.x
48	mov.u32	%r5 %tid.x
56	mad.lo.s32	%r1 %r3 %r4 %r5

Figure 3.9. Instructions preparing threads for kernel task with thread-specific data in the cycle range of [5000, 6500].

Throughout the execution of the *K0*, the high miss rate on L1D and L2 caches

in Table 3.4 results in several global memory accesses among the requests. Our *General View* reveals the effect of those accesses on power dissipation with the yellow line in Figure 3.8. The memory partitions dissipate half of the total power, around 50W, due to constant intensive activation at runtime. Furthermore, IPC and power dissipation increase with the active SMs moments. By tracking instances where IPC increases between [55000, 105000] and [105000, 155000] cycles, we can see parallel increments in power metrics dissipated by SMs and GPU. While the overall IPC value for *K0* is 31.76, runtime IPC varies in the range of [5, 75] as in Figure 3.8. The concurrently issued load/store instructions cause small latencies, oscillating the overall SM throughput during runtime.

### 3.4. Summary

Briefly, *GPPRMon* proposes a systematic runtime micro-architectural metric collection for instruction monitoring, performance, memory access, and driven power concerns. It provides the multi-perspective visualizer framework that displays performance, execution statistics of the workload, occupancy of the memory hierarchy, and dissipated energy results to conduct baseline analysis on GPUs at runtime. *GPPRMon* reliably reveals all the interactions between hardware and application at runtime with these properties and helps explicit observations of the execution at the assembly instruction level. *GPPRMon* has multiple user configuration capabilities concerning simulation and visualization overhead, such as settling the runtime metric sampling frequency or accumulating the statistics. Furthermore, the visualizer provides a set of properties for determining execution monitoring intervals and a point of view of the execution behavior or hardware occupancy among the General View, Spatial View, and Temporal View. As a result, *GPPRMon* allows analyzing performance by relating execution behavior through hardware utilization and assembly instructions and supports driven power with its distribution among GPU components at runtime.

## CHAPTER 4

# SOFT ERROR VULNERABILITY PREDICTION OF GENERAL PURPOSE GPU APPLICATIONS

The GPUs which reduce the execution times significantly exhibit a higher vulnerability to soft errors due to their complex structures, especially in extreme conditions such as in highly radiated or high-temperature conditions. Hence, soft error reliability becomes a critical concern for GPU applications. Various fault tolerance approaches (Dimitrov et al. (2009); Mahmoud et al. (2018); Mittal and Vetter (2016)) have been employed to enhance the reliability of GPUs, but they induce cost and performance overhead. Thus, assessing the soft error vulnerability of GPU programs becomes critical when deciding on the optimum fault tolerance approach.

The fault injection (FI) approach that introduces faults into the system with configurable runtime conditions (Fang et al. (2016); Hari et al. (2017)) is a widely used vulnerability evaluation technique based on controlled experiments. FI injects faults into hardware structures during the program execution, then tracks the program to see how the injected fault affects the program outcome. Using silent data corruption (SDC) rates gathered from FI experiments as a soft error vulnerability metric, one can determine whether explicit fault tolerance strategies are required for the GPU program execution. In contrast, fault injection (FI) techniques require numerous experiments to quantify the vulnerability, which can be impractical, especially for long-running systems. Therefore, there is a need to predict the susceptibility to soft errors without relying on extensive FI testing for each target software. This chapter proposes a study that utilizes machine learning (ML) to predict the soft error vulnerability of GPU applications. The goal is to save time and resources by leveraging ML algorithms to discover patterns between fault rates, performance, hardware usage, and program characteristics derived from micro-architectural metrics collected through simulation or program profiling. While several ML-based prediction mechanisms are available for CPU systems (Guo et al. (2021); Jauk et al. (2019); Laguna et al. (2016); Lu et al. (2014); Oliveira et al. (2018); Öz and Arslan (2021)), the existing approaches targeting GPU applications are limited (Kalra et al. (2018); Nie et al. (2018)).

To the best of our knowledge, this is the first study that uses a comprehensive set of regression and classification models supported by feature selection, preprocessing,

and outlier elimination techniques and trained with micro-architectural and performance indicators to predict soft error vulnerability at the kernel level for GPU programs. Our major contributions are as follows:

- We conduct FI experiments for forty-five GPU kernel functions selected from PolyBench and Rodinia benchmark suites by utilizing the FI tool (Öz and Karadaş (2022)) developed by the PARS research group.
- We collect metrics to cover micro-architectural interactions of hardware and program, performance statistics, and PTX instructions to trace meaningful patterns between programs and fault rates. We use the GPGPU-Sim (Khairy et al. (2020)) simulator and Nsight Compute tool (NVIDIA (2022a)) profiler to collect mentioned metrics.
- We employ regression models to predict masked fault rates, while we utilize classification models for predicting SDCs and crash rates, which are hard to predict. We enrich ML-based models with feature selection, outlier elimination, and data preprocessing stages to increase our prediction accuracy values.
- We achieve 95.91%, 88.46%, and 85.71% prediction accuracy results for masked fault, SDC, and crash rates, respectively.

## 4.1. Related Work

Since reliability is less investigated than performance issues and vendors generate precautions without taking care of the optimizations' overheads, there is also less research investigating the soft error vulnerability of the GPU applications. Even if there are interesting works about the reliability of multiprocessors evaluating both the architecture vulnerability factor (AVF) and performance vulnerability factor (PVF), GPU literature lacks in evaluating error reliability with multi-perspectives as they are comparably new hardware devices.

Sabena et al. (2014) propose a reliability evaluation for GPU-based FFT algorithm by disturbing the device with radiation experiments. First, they divide FFT into fifteen stages and determine each stage's error proneness with different floating numbers, which affect register usage. They evaluate the reliability of GPUs with and without enabling the L1 cache, and they obtain the most reliable GPU hardware configuration by turning off

the L1 cache usage. Their approach aims to find the most reliable hardware configuration for the FFT algorithm. In this work, we evaluate the application’s error proneness and predict these errors via ML-based models. We include varying GPU applications and benefit from both hardware and performance metrics to predict the error proneness of the GPU kernels.

PRISM (Kalra et al. (2018)) presents a framework built upon SASSIFI (Hari et al. (2017)) to predict the error vulnerability of GPU programs. They use linear regression and similarity-based approaches. While we choose our applications from Polybench and Rodinia benchmark suites, their benchmark consists of selected GPU applications from Parboil, Rodinia, and CUDA SDK. Even if they use many GPU programs, they do not analyze features from different perspectives. They solely examine the instruction type in the prediction experiments as we include them in our simulator features. In addition to an extensive feature space from the profiler and the simulator, we also try to select the most beneficial ones for the prediction and classification experiments. Moreover, we apply preprocessing methods to our dataset and outlier elimination to ignore irrelevant data samples. Furthermore, we investigate the effect of ML-based schemes on those predictions by using commonly used regression (*GB*, *RF*, *SVM*) and classification (*GB*, *SS+SGD*, *RF*) approaches. In contrast, they only use *Linear Regression* and *K-Nearest Neighbour* (K-NN) algorithms. Lastly, we provide fault predictions on the kernel basis while their prediction results are based on the complete GPU program. Observing results on a GPU application basis can cause different results regarding the propagation of the corrupted data since it can be more critical for long-term executions such as the GPU programs, including multiple kernel launches. They provide 90% accuracy rates for the prediction of masked faults while we obtain 95.9% accuracy. Their work has no numeric accuracy result for predicting SDCs and DUEs because there is a large standard deviation in prediction accuracy. Unlikely, we can achieve 75.67% and 73.91% accuracy in classifying 3-class SDCs and crashes, respectively.

Du et al. (2019) analyze the relationship between the soft errors and the major architectural GPU structures such as register files, streaming multiprocessors, and register files. Then, they find relationships between architectural properties and soft error vulnerabilities. However, when defining those relationships, they do not include the program’s architectural usage metrics of the belonging GPU applications. In our work, we exploit both PTX instructions and hardware usage amounts of the corresponding GPU program to capture the reasonable patterns which increase predicting the soft error vulnerabilities. In other words, their AVF metrics to describe soft error reliability for the GPU programs do



not cover detailed architecture properties. However, we also provide the statistical results for correlation, preprocessing, outlier elimination, and score analysis between hardware usage/instruction type metrics and soft errors.

Nie et al. (2018) present ML-based approaches to predict fault rates in HPC systems. First, they correlate the GPU faults with their workload and different system characteristics. Then, they create temporal and spatial feature groups. Error rates are trained with the selected features to obtain a prediction model built on *Neural Network* (NN), *Support Vector Machine* (SVM), *Gradient Boosting Decision Tree* (GBDT), and *Logistic Regression* (LR). They focus on feature selection instead of the prediction results to find and boost the most relevant features. They also investigate the overhead of the prediction models in terms of execution time. For example, while *LR* finalizes in 4.8 seconds, *SVM* takes 1.04 hours to obtain the model. They represent the prediction results both in time and space. However, their feature set, including power consumption and temperature metrics, is limited compared to ours' which consists of results obtained from both profiler and simulator. Their main focus is to compare the prediction model and trade-offs by comparing accuracy results, overheads, scores for precision, and boosts to the most relevant features. In contrast, our work aims to find a way to obtain the most successive prediction model with a vast space basis and varying GPU kernels.

Wei et al. (2019) evaluate the soft error resilience of PTX instructions instead of the whole application. Their investigation is a bit more detailed than ours because we investigate the soft error resilience of the GPU applications at the kernel function level. The main motivation is to decrease the protection overhead by applying precautions for the error vulnerability of instructions. They classify instructions as float, integer, load, and store. They show that load-type instructions result in SDCs with the lowest probability among those instruction types. In contrast, float-type instructions have a higher probability of exposing SDCs than the other instructions. Their findings match ours since we find that masked faults are highly correlated with the amount of global memory writes. Intuitively, one can imply that the disruption will be masked after the write operation even if the corresponding register is disrupted during the fault injection.

Öz and Arslan (2021) apply a similar approach as ours except for target architecture and programs. Their work provides fault prediction models for the 30 multi-threaded applications from PARSEC, SPLASH, and ParMiBench benchmark suites executed on CPUs. Their features include performance, parallel programming, data sharing, and thread communication characteristics. While they utilize *RF Regression*, *SVM*, and *GB Regression* ML-based algorithms for the regression experiments, they benefit from

*SVM*, *GB*, and *RF* algorithms for the classification algorithms. They also provide precision and scoring results to verify their results' success. They reach 88% accuracy levels with the *GB* algorithm with reliable recall, precision, and F-score values.

Prediction-based approaches for soft error vulnerability evaluation have been proposed in the literature to deal with long-running fault injection experiments. Our work demonstrates a detailed soft error vulnerability prediction analysis and aims to broaden the research about soft errors and their predictions in the GPU literature.

## 4.2. Methodology

Our main aim is to estimate fault rates by analyzing the relationship between the soft error vulnerability of GPU applications and metrics consisting of micro-architectural ones, instruction characteristics, and performance. We use machine learning-based regression and classification models to predict the impacts of soft errors on program execution. Figure 4.1 depicts a high-level overview of our experimental framework, divided into six main stages. First, we execute FI experiments to soft error rates and collect metrics for our target programs. Subsequently, we perform preprocessing stage, whose output is shown as preprocessed data and feature selection/reduction among the collected metrics. After feature selection, we eliminate outlier samples from the dataset using selected features and fault rates. The gray boxes in Figure 4.1 represent specific methods for feature selection and outlier elimination stages. We continue with prediction experiments after the elimination of outlier samples. The prediction stage takes selected features and fault rates as input and generates prediction results as output. Brief descriptions corresponding to the six main stages are as follows:

1. **Fault injection:** We conduct FI experiments with a recently proposed framework (Öz and Karadaş (2022)) by the PARS research group to obtain kernel-level soft error rates for a diverse GPU application set. Specifically, we collect three soft error outcomes: *masked faults*, *crashes*, and *SDCs*.
2. **Metric collection:** We utilize simulation and profiling methods to collect micro-architectural and performance metrics. For simulation, we use GPGPU-Sim (Khairy et al. (2020)) that simulates CUDA- or Open-CL-based programs on a virtual hardware environment, and provides performance indicators hardware usage metrics, and extracts PTX level instructions. For profiling, we run selected benchmark GPU

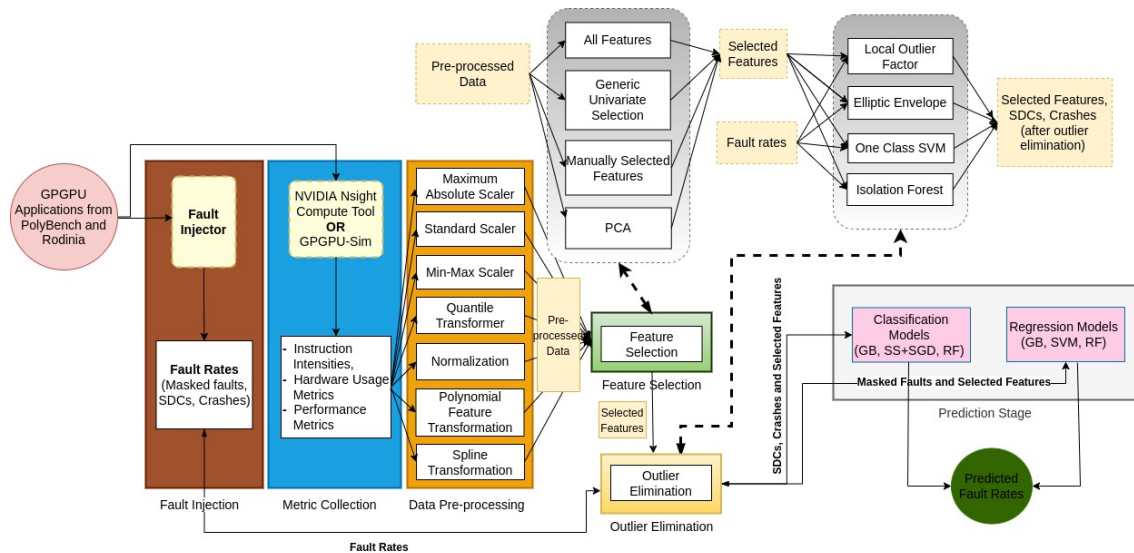


Figure 4.1. General overview of fault prediction framework.

programs through the Nsight Compute tool (NVIDIA (2022a)) that generates detailed micro-architectural metrics for the target GPU programs.

3. **Data preprocessing:** We perform various preprocessing techniques, including normalization, scaling, and transformations, to provide a more reasonable basis between the collected metrics and the fault rates for feature selection and prediction phases.
4. **Feature selection:** Since there are many metrics related to performance, hardware usage, and program behavior, some can create disruptive effects for the prediction phase. We perform feature selection to identify and extract the most helpful features before the prediction phase.
5. **Outlier elimination:** Similar to the irrelevant metrics, there are also outlier GPU kernels that cause less prominent fault rates than the others. Removing such kernels from the dataset contributes to accuracy rates for predictions.
6. **Fault rate prediction:** We use machine learning-based regression and classification algorithms to predict masked faults and SDCs/crashes, respectively.

We explain the details of those steps in the following sections.

### 4.2.1. Fault Injection Framework

We use the FI tool developed by the PARS research group (Öz and Karadaş (2022)), which allows regional soft error vulnerability evaluation for target GPU programs. The FI region is specified by the user on the source code of the GPU kernel function. The code must be compiled by enabling debugging (with `-g` and `-G` for activating debug in host code and device code, respectively) with `cuda-gdb` (NVIDIA (2022b)). The tool first generates a golden output by regularly executing the program with target input. Then, it generates a fault map for FIs by profiling the application. Lastly, it starts running the application until it reaches the FI point’s breakpoint. At the breakpoint, the tool pauses the execution and changes a register value depending on the fault map before executing the specified instruction. Later, it activates the execution to continue where the execution is paused. If the execution terminates without an error, the tool compares the output and golden output and decides whether the fault causes an SDC or does not affect the program output. If an error occurs during the execution, the tool reports it as a crash state.

We conduct FI experiments for each kernel function to obtain generic soft error vulnerability characteristics among target GPU programs. Figure 4.2 shows the target GPU kernels on the x-axis and the corresponding soft error vulnerability rates on the y-axis.

### 4.2.2. Metric Collection

We collect two sets of metrics from the simulation and profiling of the target GPU applications. Specifically, we gather performance, micro-architectural usage, and instruction intensity (by taking care of opcodes) metrics from the simulation environment. At the same time, we focus on detailed micro-architectural occupancy metrics and obtain them from the profiler tool. The main motivation behind creating two different datasets is to investigate the relationship between the simulator and the profiler metrics and interpret their impacts on the results.

We simulate the target benchmark GPU programs and collect a set of metrics quantifying and characterizing the execution using the performance simulation model of the GPGPU-Sim (Khairy et al. (2020)) (hereafter referred to as *the simulator* in this chapter). Moreover, we profile the same benchmark programs through Nsight Compute



Figure 4.2. SDC, crash, and masked fault rates for target GPU kernels.

(hereafter referred to as *the profiler* in this chapter) and gather precise micro-architectural occupation and performance metrics. We intuitively collect only the metrics that can potentially reveal meaningful patterns between fault rates of the target GPU kernels and metrics. For instance, we include the metrics representing main program features, such as SM throughput or the intensity of memory operations for a kernel; however, we discard more specific metrics, such as DRAM row utilization rate. The reason for not including all the gathered metrics is to exclude the ones that would mess up the dataset and break the meaningful patterns.

Table 4.1. The metrics collected from the simulator.

<b>Metric types</b>	<b>Metric descriptions</b>
load_inst	Loads instructions
store_inst	Store instructions
shMem_inst	Instructions using shared memory
paramMem_inst	Instructions using parameter memory
total_inst	Total instructions
ipc	Instruction per cycle
sim_rate	Simulation rate (Simulation per wall time)
globMem_read	Total global memory reads
globMem_write	Total global memory writes
bwUtil	Average bandwidth utilization
warpOcc	Average warp occupancy on SMs
ctrlFlowInsts	Control flow inst. / total inst. in PTX
memInsts	Memory inst. / total inst. in PTX
aritInst	ALU inst. / total inst. in PTX
sfuInsts	Special Function unit (trigonometric) inst. / total inst. in PTX
mAddInsts	Fused (i.e., add+mul) inst. / total inst. in PTX
textureMemInsts	Texture inst. / total inst. in PTX
binaryInsts	Binary inst. / total inst. in PTX
others	Other inst. / total inst. PTX
fpInsts	32-bit Floating-Point inst. / total inst. in PTX
signedInsts	32-bit Signed-Integer inst. / total inst. in PTX
unsignedInsts	32-bit Unsigned-Integer inst. / total inst. in PTX

Table 4.1 and 4.2 present the intuitively collected metrics from the simulator and

the profiler, respectively. Since the simulator and the profiler serve different purposes, it is unlikely to collect all the same metrics. The profiler generates in detail micro-architectural and performance metrics statistically, whereas the naive simulator reports hardware performance measurements and kernel characteristics with the target kernel's PTX instruction. Even if they differ in usage scenarios, both supply some metrics, such as IPC, achieved occupancy in SMs, and the total number of instructions executed for a kernel. We consider each group of metrics independently since they represent different executions, and we build separate prediction models for each dataset.

Table 4.2. The metrics collected from the profiler.

<b>Metric types</b>	<b>Metric descriptions</b>
sol_sm	SM throughput
sol_mem	Compute memory pipeline throughput
sol_L1TexCacheSOL	L1 texture memory throughput
sol_L2Cache	L2 cache throughput
sol_Dram	GPU DRAM throughput
duration	Total duration in msec
elapsed_cycle	Total cycles where GPU is active
ipc	Total issued warp instructions per cycle
sm_busy	SM throughput as percentage
mem_thput	Total accessed bytes on DRAM (Gbyte/sec)
l1TexHitRate	Hit rates on L1 and texture caches
l2HitRate	Hit rates on L2 caches
memBusy	Internal activity of memory partitions on DRAM
maxBand	Reached maximum bandwidth connected to DRAM
activeWarpSch	Active warp throughput on schedulers
warpCycInst	Total cycles where warps instructions are resident
execInst	Total executed instructions
regPerThread	Register allocation per thread
achedOccup	Achieved overall occupancy in percentage
achedActiveWarp	Cumulative number of warps in flight on average over the runtime

### 4.2.3. Data Preprocessing

ML-based approaches mostly make predictions by running classification and regression methods such as trees and regressors. During training, non-scalable numbers can cause bias and flatten internal algorithm parameters for the estimation. For example, while instruction intensities range in  $[0, 1]$ , the number of instructions for kernels and the total number of memory operations comprise enormous values. To make sure that the numbers are scalable to each other, we perform preprocessing techniques provided by the built-in functions of scikit-learn (Pedregosa et al. (2011)).

We utilize *Maximum Absolute Scaler*, *Standard Scaler*, and *Min-Max Scaler* as scaling methods. The scalers scale numeric values into the same interval. For example, the *Maximum Absolute Scaler* scales all numbers after finding the maximum absolute value such that the scaled values of variables are calculated using Equation 4.1a. Similarly, Equation 4.1b shows the formula for *Min-Max Scaler* algorithm. Furthermore, the *Standard Scaler* method scales benefiting from both mean value and standard deviation as in Equation 4.1c.

$$x_{scaled} = \frac{x}{\max(X)} \quad (4.1a)$$

$$x_{scaled} = \frac{x - \min(X)}{\max(X) - \min(X)} \quad (4.1b)$$

$$z = \frac{x - \mu_1}{\sigma} \quad (4.1c)$$

In addition to scaling methods, we apply normalization and transformation techniques for preprocessing methods. *Normalization* scales the metrics into the specified intervals. On the other hand, the *Quantile Transform* transforms the metric space to the uniform probability distribution. *Polynomial Feature* generates a new feature space consisting of all polynomial combinations of the metrics with the user-specified degree. For example, if the metric space is two dimensional and of the form  $[x, y]$ , the second degree *Polynomial Features* are  $[1, x, y, x^2, xy, y^2]$ . The *Spline Transform* method generates a univariate basis spline with minimal support for given preconditions by the user.

---

<sup>1</sup> $\mu$  and  $\sigma$  stand for the mean and standard deviation, respectively.



#### 4.2.4. Feature Selection

To figure out the decisive metrics and filter the irrelevant ones, and increase the prediction accuracy for fault estimation experiments, we perform the following feature selection methods: 1) *Generic Univariate Selection*, 2) *Principle Component Analysis (PCA)*, 3) *Pearson and Spearman Correlations*.

*Generic Univariate Selection* evaluates features individually and determines the strength of the relationship of the metric with the target output. Based on the univariate statistical tests, it captures the best features among all metrics. It also allows performing feature selection with a configurable strategy enabling hyper-parameter search for the prediction tests (Pedregosa et al. (2011)).

While working with high dimensional datasets, as in our case, *PCA* reduces dimensions from the feature space (Abdi and Williams (2010)), beneficial for pattern capturing complexity. It transforms the correlated metrics to linearly independent (orthogonal) metrics so that significant relationships between metrics and target is captured to enhance the accuracy of ML-based prediction experiments while reducing the dimensionality. In this context, we aim to save from the irrelevant metrics that can create noise effects on internal parameter settling.

*Spearman* correlation coefficient describes the monotonic relationship between the metrics and the fault rates without considering whether the relationship is linear. Equation 4.2 represents *Spearman's* rank correlation coefficient formula. The approach scales relevance in the range [-1, 1], where the closeness to -1 or 1 describes a high relation between those two parameters. Additionally, *Pearson's* correlation coefficient describes a linear relationship between the two inputs, and the results are scaled in between [-1, 1] as in *Spearman's*. The closeness to -1 or 1 describes that those two metrics are correlated, while closeness to 0 reveals the dissimilarity between those metrics. Equation 4.3 shows the *Pearson* correlation coefficient formula.

$$\rho_s = 1 - \frac{\sum_{i=1}^n d_i^2}{n^3 - n} \quad (4.2)$$

$$\rho_p = \frac{E[(X - \mu_X)(Y - \mu_Y)]}{\sigma_X \sigma_Y} \quad (4.3)$$

---

<sup>2</sup> $d$  and  $n$  stand for the difference between two rankings and the number of observations, respectively.

<sup>3</sup> $E$ ,  $\sigma$ , and  $\mu$  stand for the expectation, the standard deviation, and the mean, respectively.

Figure 4.3 demonstrates the correlation between micro-architectural metrics and fault rates. For example, *regPerThread* metric represents the allocated number of registers per thread. Those registers can be used for indexing memory locations or temporary usages, and the faults can be masked or affect the output depending on the register's usage purpose in the program. As demonstrated in Figure 4.3, its correlation results with *Pearson* method are -0.11 for SDC, 0.53 for the crash, and -0.39 for the masked faults, respectively. Moreover, the amount of global memory writes is crucial since the faults can be masked due to the store operations. The corresponding correlation result is 0.18 with the *Pearson* method.

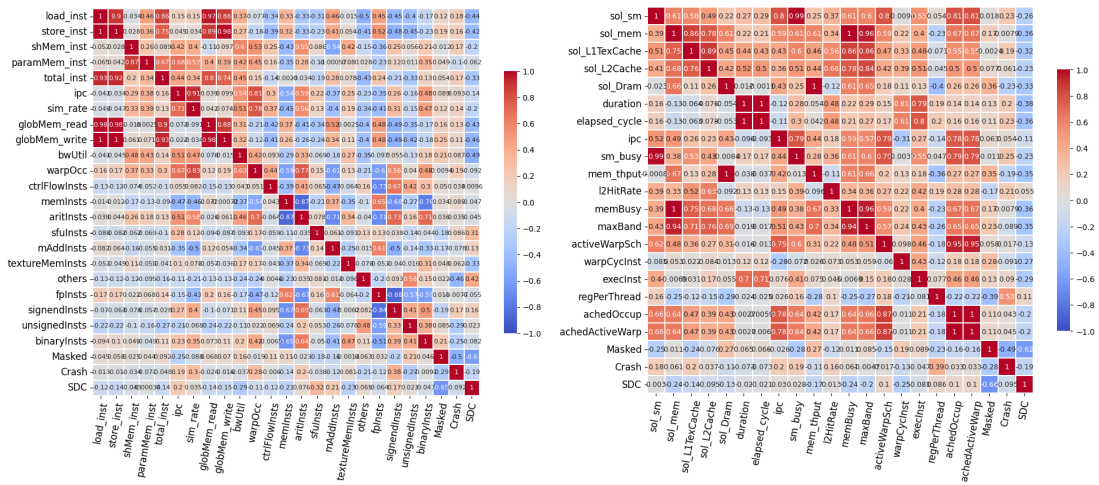


Figure 4.3. The upper and lower triangles show Pearson and Spearman correlation results, respectively, between the simulator/profiler metrics and the fault rates (left is with the simulator and right is with profiler features).

Warp occupancy is another significant metric describing the relationship between the faults and the metrics; thus, selected as a feature. *Spearman* correlation coefficient between the *warpOcc* metric from simulator metrics and crashes is 0.28.

Additionally, the instruction intensity metrics obtained by classifying instructions depending on the PTX operational codes in Table 4.1 are also highly correlated with the fault rates compared to the other metrics. To illustrate, *memInsts* and *sfnInsts* intensity metrics result in -0.21 and 0.32 correlations with SDC faults, respectively. As a result, we determine the feature set among the metrics whose either *Pearson* or *Spearman* correlation result is bigger than 0.2 with the fault rates (shown in Figure 4.3). Table 4.3 presents our selected features among simulator and profiler metrics.

Table 4.3. Selected features from the collected metrics.

	For SDC	For Crash	For Masked
<b>From the simulator</b>	load_inst store_inst shMem_inst total_inst ipc globalMem_Read globalMem_Write bwutil sfuInsts textureMemInsts memInsts	others signedInsts binaryInsts sim_rate warpOcc aritInsts unsignedInsts	globalMem_write bwutil binaryInsts ipc signedInsts unsignedInsts
<b>From the profiler</b>	sol_sm sol_Dram sol_L1TexCache sol_L2Cache execInst duration elapsed_cycle sm_busy mem_thput memBusy warpCycInst achedOccup achedActiveWarp	sol_L1TexCache ipc regPerThread sol_sm sol_Dram duration elapsed_cycle sm_busy l2HitRate	sol_Dram mem_thput maxBand warpCycInst regPerThread

#### 4.2.5. Outlier Elimination

Similar to including irrelevant metrics in the prediction experiments, outlier kernel samples can distort the ML model parameters and degrade the prediction accuracy. Hence, we eliminate outliers before beginning prediction tests. In this manner, we utilize *Local Outlier Factor (LOF)*, *Elliptic Envelope*, *One Class SVM*, and *Isolation Forest* algorithms from the scikit-learn library (Pedregosa et al. (2011)).

- *LOF* algorithm measures the locality distance among the user-specified  $k$  nearest neighbor and eliminates the outlier elements. While  $LOF(k) < 1$  represents the inlier elements,  $LOF(k) > 1$  corresponds to outlier elements where  $LOF(k)$  is the local reachability density.
- *Elliptic Envelope* algorithm creates a virtual elliptical area with the provided dataset. As one can infer, the values that fall inside the envelope are evaluated as useful, while the others (outside the envelope) are the outlier elements.
- *One Class SVM* algorithm, which is different from the *Supervised SVM*, captures the boundaries for the given data samples and helps border the outliers.
- *Isolation Forest* algorithm implements recursive partitioning among the dataset, where different from the other outlier eliminators. It examines the data samples to determine whether it adds to the isolated trees. The remaining external nodes are the outlier samples.

#### 4.2.6. Prediction Model Evaluation

We build ML-based prediction models to predict the soft error vulnerabilities for the target GPU programs at the kernel function level. Scaling down the investigation from the complete GPU programs to the individual kernel functions provides a better opportunity to observe the possible reasons for the occurrence of the faults. We utilize regression and classification approaches for our fault prediction task.

As the first approach, we apply regression algorithms to predict masked fault, SDC, and crash rates by exploiting the selected features. Specifically, we use *Random Forest (RF)*, *Support Vector Machine (SVM)*, and *Gradient Boosting (GB)* algorithms. *RF* benefits from the multiple decision trees by selecting more accurate ones from the training set. *SVM* starts with a curve type such as linear or parabolic and a certain amount of error range called epsilon. Then, it tries to fit the specified curve considering the training data by calculating the absolute value error between the expected and the actual fault rate. *GB* builds an additive curve model forward stage-wise to optimize arbitrary differentiable loss functions as in deep neural network algorithms. In each stage, a regression tree is fitted on the negative gradient of the given loss function, and the resultant stable curve is used for the prediction tests of the remaining samples of the dataset.

We build 1536 experiment configurations with the combinations of the four feature selection methods, four different outlier elimination techniques, eight different pre-processing techniques, and three ML algorithms for the regression approach to predict masked faults. Furthermore, we individually train the regression models with features deduced from the simulator and the profiler metrics.

For our regression models, we calculate the accuracy results according to the following formula:

$$\left(1 - \frac{|error_{predicted} - error_{observed}|}{error_{observed}}\right) * 100 \quad (4.4)$$

Even if we can reach accurate prediction results for the masked fault rates, which range [0.682, 0.939], the regression models do not yield similar accuracy results for the crash and the SDC rates with the values in the range of [0.012, 0.263] and [0.008, 0.173], respectively. Since the masked fault rates are pretty large compared to SDC and crash rates, the little oscillations around the fitted curve for the test samples are acceptable. However, similar slight deviations are not acceptable for the SDC and the crash rates because these deviations cause more significant prediction errors. Hence, we build classification models for predicting the SDC and the crash rates as an alternative approach.

For the classification approach, similar to the previous work Öz and Arslan (2021), we define different classes by considering the SDC and crash rates and predict the class of each kernel function in our evaluation. Specifically, in the *two-class* model for SDC prediction, we define two classes by considering the SDC values of the target kernel functions. For example, the kernel functions with SDC rates between [0.010, 0.050] and [0.051, 0.200] are classified as *Not Vulnerable* (with lower SDC rates) and *Vulnerable* (with higher SDC rates), respectively. Accordingly, we can predict whether GPU kernels are vulnerable to soft errors. As a result, we create two different classification experiment models by dividing our dataset into two and three classes so that each class has nearly the same data.

In addition to *RF* and *GB* algorithms, we utilize an ensemble classification model configured by cascading the *Standard Scaler (SS)* and *Stochastic Gradient Descent (SGD)* classifier. In addition to *SS* explained in the Data Preprocessing section, *SGD* algorithm calculates the gradient loss estimated for each sample, and the algorithm parameters are updated along with the learning rate accordingly. Moreover, *SGD* algorithm enables using L1 and L2 regularizations to disturb algorithm parameters and prevent overfitting.

We build 384 classification models to predict the SDC and the crash rates. We use one configuration for each ML model instead of building them with different hyperparameters. We evaluate classification success depending on the accuracy, simply the correct classification rate.

Since our dataset has a limited number of data samples (i.e., 45 kernel functions), we prefer to use the K-fold cross-validation method for the prediction work where K corresponds to the test sample number. The 1-fold method uses 44 (K-1, where K = 45) kernels to train ML models and one kernel to predict the fault rate for each fault type by passing through trained models.

### 4.3. Experimental Study

#### 4.3.1. Experimental Setup

For the experimental evaluation, we select 23 CUDA applications, where 13 of them are from Polybench (Grauer-Gray et al. (2012)), and 10 of them are Rodinia (Che et al. (2009)) benchmark suites, respectively. Table 4.4 presents the descriptions of the target programs.

We perform FI experiments by targeting each kernel function of the programs and similarly collect the metrics for individual kernel functions. We conduct FI tests on an Intel-based workstation with an NVIDIA Quadro P4000 (NVIDIA (2022c)). We use 1000 FIs per kernel function by using the statistical approach (Leveugle et al. (2009)), which provides a confidence level of 95% and an error margin of 3%. Since we use NVIDIA GPUs in our experiments, we use the NVCC compiler and cuda-gdb debugger for the FI and compilation processes. While we exploit from the Nsight Compute tool, version 2019.4.0, our simulation environment is the GPU-Sim v4.0 simulator. We configure the Quadro P4000 device based on hardware resources (NVIDIA (2018a, 2022c)) on the simulator by specifying the parameters such as SM amount (SIMT cluster), warp scheduler, DRAM memory channel partitions, L1 and L2 cache sizes. Afterward, we conduct simulations in the performance mode of the simulator. We share our configuration files and collected metrics from both the profiler and the simulator in our GitHub repository<sup>4</sup>.

---

<sup>4</sup><https://github.com/BT-MasterThesis-2020-23/SoftErrorVulnerabilityPrediction-GPGPUs>

Table 4.4. CUDA applications used in our experiments.

	Application	Description
Taken from PolyBench	2DConvolution	Convolution with 2D input data
	3DConvolution	Convolution with 3D input data
	3mm	Multiplication of 2 3D matrix
	Atax	Matrix transpose and vector multiplication
	Bicg	Sub kernel of BiCG Linear Solver
	Correlation	Correlation computation
	Covariance	Covariance computation
	Fdtd-2d	2-D Finite different time domain kernel
	Gramschmidt	Gram-Schmidt decomposition
	Gemm	Matrix multiplication $C=\alpha.A.B+\beta.C$
	Gesummv	Scalar, vector, and matrix multiplication
	Mvt	Matrix vector product and transpose
	Syrk	Symmetric rank-k update
Taken from Rodinia	Backprop	ML algorithm trains weights and nodes
	Bfs	Graph traversing algorithm for all graph components
	Gaussian	Gaussian elimination method implementation
	Heartwall	Tracking of the movement over an ultrasound image responding to the stimulus
	Hotspot	Thermal simulation tool to estimate processor architecture
	Hybrid Sort	Fusion of merge sort on bucket sort algorithms
	Needleman - Wunsch	Global optimization method for DNA sequence alignment
	NN-Euclid	Nearest neighbor finder algorithm
	Streamcluster	Online clustering
	Srad version 2	Diffusion algorithm based on partial differential equations to remove speckles in an image

First, we apply preprocessing to our dataset since the correlation results among all micro-architectural metrics are not promising, and their values are not scalable compared to each other. Based on preprocessing, we interpret how scaling, normalization, and transformation affect our prediction accuracy results. Secondly, we apply feature selection/reduction methods before prediction. Specifically, we use *Genetic Univariate Selection*, *PCA*, and *Manual Feature Selection*, where we select the features by using *Spearman* and *Pearson* Correlation results in addition to prediction models with all features. Like

Table 4.5. Preprocessing, Feature Selection/Reduction, and Outlier Elimination methods and their hyper-parameter configurations.

		Hyper-parameters
<b>Preprocessing Methods</b>	Maximum Absolute Scaler	With default parameters
	Standard Scaler	With default parameters
	Min-Max Scaler	With default parameters
	Quantile Transform	Output distribution = 'normal', randomness = 0
	Normalization	Norm = 'l2'
	Polynomial Features	Degree = 3
	Spline Transform	Degree = 2, n_knots = 3
<b>Feature Selection &amp; Reduction Methods</b>	Generic Univariate Selection	Mode = 'percentile', param = 50
	Manual Feature Selection	Spearman Corr $\geq 0.2$    Pearson Corr $\geq 0.2$
	Principle Component Analysis	With 9 dimensions from all dimensions
<b>Outlier Elimination Methods</b>	Local Outlier Factor	With default parameters
	Elliptic Envelope	contamination = 0.01
	One Class SVM	nu = 0.01
	Isolation Forest	contamination = 0.1

preprocessing tools, scikit-learn includes built-in mentioned feature selection and reduction functions. Lastly, we apply outlier detection methods to remove outliers due to their noisy effects on the prediction experiments. We try with the *LOF*, *Elliptic Envelope*, *One Class SVM*, and *Isolation Forest* methods for outlier elimination. Table 4.5 presents all the hyper-parameters for the configuration of preprocessing, feature selection/reduction, and outlier elimination methods.

We employ three different machine learning algorithms for the regression (*SVM*, *RF*, and *GB*) and the classification (*Pipelined SS+SGD*, *RF*, and *GB*) experiments, where each algorithm has four different configurations based on hyper-parameters as part of scikit library Pedregosa et al. (2011). The hyper-parameter values used in our evaluations are as follows: The learning rates used for the *GB* Regression are 0.001, 0.01, 0.1, and 0.25. For the *RF* Regression, we configure the effect of randomness as 20 and 50, the number of estimators as 100, 1000, and 10000, and keep other parameters as default. For the *SVM* regression, we assign the kernel function as polynomial curves with two and



three degrees, sigmoid and radial curves. For the classification, the maximum depth of trees is two, and the total iteration amount is 1000 for each algorithm. In addition, the epsilon error is 0.001 (i.e., minimized error threshold), and the number of parallel workers is 20.

## 4.3.2. Experimental Results

### 4.3.2.1. Preprocessing Methods

Figure 4.4 demonstrates the highest prediction accuracy results for the preprocessing methods for our classification experiments. The predictions with *Polynomial Features* bring the most accurate predictions for 2-class SDCs with simulator features, while *Quantile Transform* generates the most accurate ones with the profiler features. In addition, *Quantile Transform* achieves the most successful estimations among the classification experiments with the profiler features except for the prediction of 3-class SDCs.

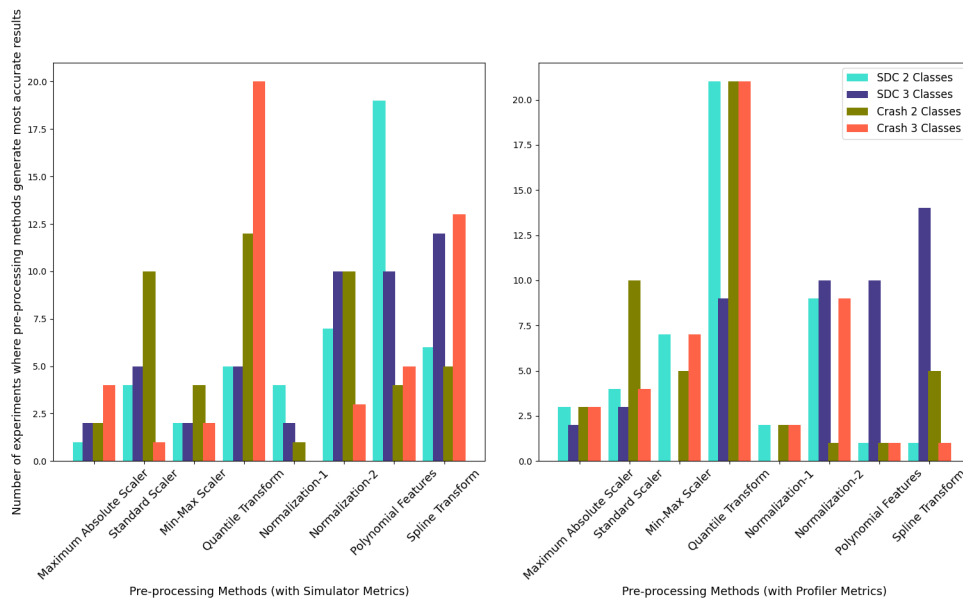


Figure 4.4. Prediction experiments, where preprocessing methods result in the highest accuracy values by keeping feature selection, outlier eliminator, and ML classifier the same.

With simulator features, the *scalers* cannot generate as accurate results as transformers. In contrast, transformers have low accuracy rates except for the *Quantile Transform* for most predictions. We can conclude that *Quantile Transform* increases prediction accuracy with profiler metrics. If developers use the simulator metrics, they need to act by taking care of which method to use; for instance, if they predict 2-class SDCs, they should use *Normalization* as preprocessing.

### 4.3.2.2. Feature Selection

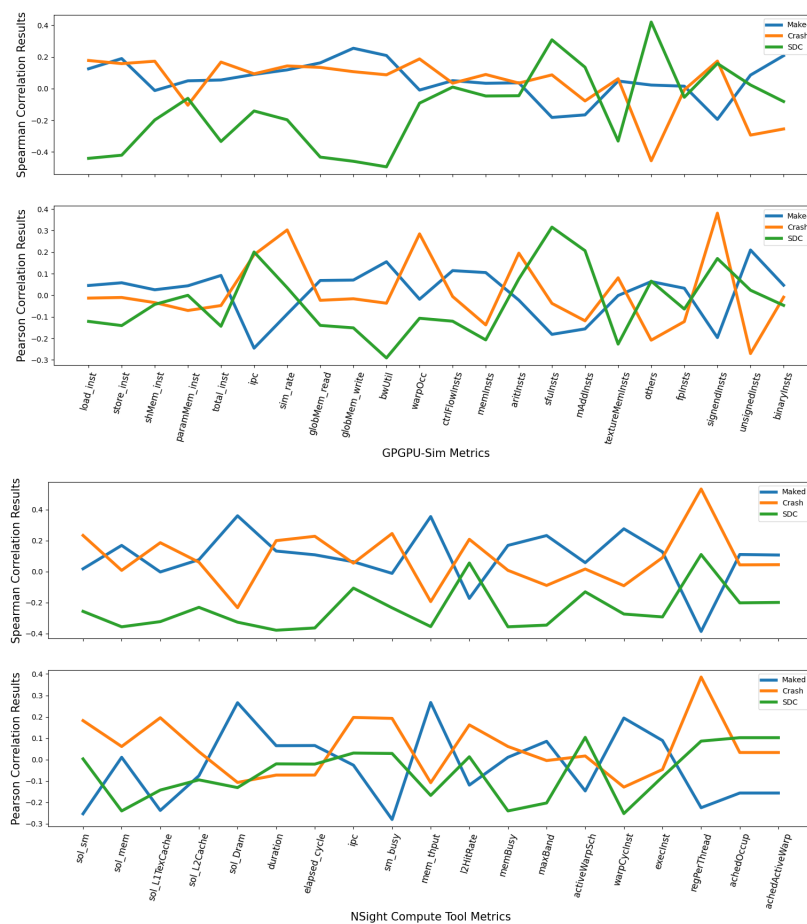


Figure 4.5. Spearman and Pearson correlation results between the features and the fault rates.

After collecting the micro-architectural and performance metrics from the simulator and the profiler, we apply the feature selection methods before eliminating outlier kernels from the dataset. Figure 4.5 presents *Spearman* and *Pearson* correlation results to show the statistical relationship between fault rates and features extracted by both simulator and profiler. The relationship of any feature is not the same for all fault types. For

example, while *Spearman* correlation coefficient calculated with compute unit memory bandwidth throughput (*sol\_mem*) and SDC rate is above 0.35, this feature has a correlation value lower than 0.2 for the crashes and the masked faults. Thus, we separately apply feature selection approaches to datasets for each fault type. For *Manual Feature Selection*, we select the metrics whose either *Spearman* or *Pearson* correlation coefficient is greater than 0.2 with SDCs, crashes, or masked faults.

Figure 4.6 demonstrates how each feature selection/reduction approach affects the prediction accuracy. *PCA* results in the highest accuracy values for the experiments except for the 3-class SDC classification with simulator metrics.

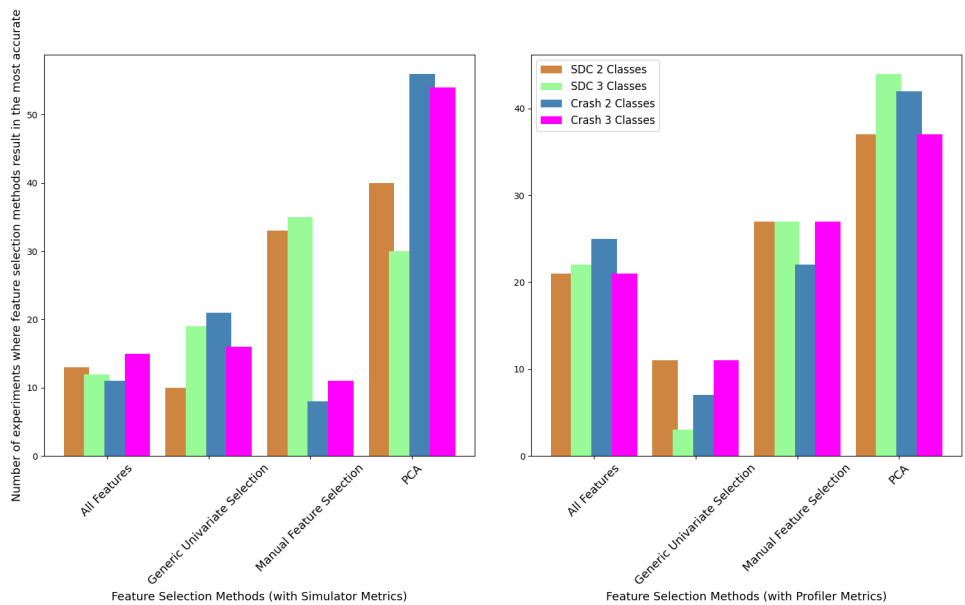


Figure 4.6. Prediction experiments, where feature selection methods result in the highest accuracy values by keeping preprocessing, outlier eliminator methods, and ML classifier the same.

Feature selection and reduction methods act differently. While the feature selections select some metrics as representative features, the reduction method tries to reduce dimensions by investigating the internal relationship among the features. In our case, *PCA* solves the eigenvalue problem on our dataset to create nine different components. The main reason behind the highest accurate scores is that resultant components (fused features) after the *PCA* are orthogonal to each other, giving identical classification indicators for the classification trees. Hence, classification models perform better with the *PCA* method.

### 4.3.2.3. Outlier Elimination

Figure 4.7 shows the effect of the outlier elimination methods on our prediction accuracy values. With simulator features, while *Elliptic Envelope* results in the worst prediction results, *One Class SVM* has the best accuracy rates with both the simulator and the profiler features on average. For the prediction of 2/3-class SDCs, *Isolation Forest* and *One Class SVM* are the most successive outlier eliminators, respectively, when we use features obtained from the simulator for training.

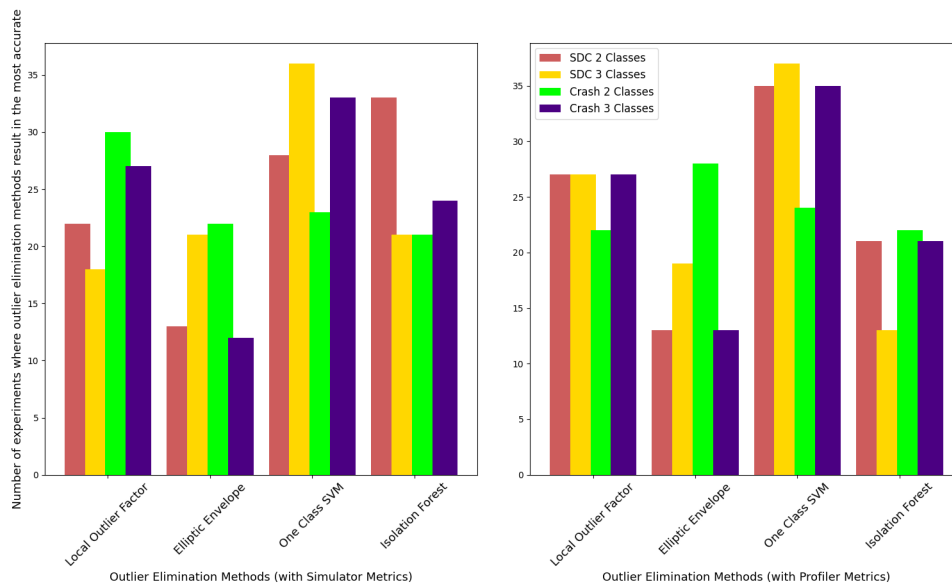


Figure 4.7. Prediction experiments, where outlier elimination methods result in the highest accuracy values by keeping preprocessing, feature selection methods, and ML classifier the same.

Additionally, *LOF* and *One Class SVM* are the most reliable outlier eliminators for 2/3-class crashes. Furthermore, the *One Class SVM* eliminator provides the best estimation results for the prediction of 2/3-class of SDCs and 3-class of crashes, with the profiler features. *Elliptic Envelope* provides the most accurate predictions for classifying crashes with 2-class.

#### 4.3.2.4. Regression Results

We use the regression-based ML models to predict the masked fault rates. Table 4.6 shows the accuracy results of the regression algorithms based on the simulator and the profiler metrics. We experiment prediction of masked faults with all the mentioned outlier elimination and preprocessing methods. However, we list the results by selecting the most accurate results with the specified feature selection and classifier methods. While all algorithms have prediction accuracy values larger than 90%, the *GB* algorithm results in the highest prediction accuracy, 95.905%, with the *Manual Feature Selection* from simulator metrics preprocessed with *Standard Scalar* and *One Class SVM* outlier elimination method. In addition, the *RF* algorithm results in 96.323% with all profiler features preprocessed with *Min-Max Scaler* and *One Class SVM* method.

Table 4.6. Regression accuracy results for masked faults on each machine learning algorithm.

Algorithm	The simulator metrics				
	All Features	Gen. Uni. Sel.	Man. Feat. Sel.	PCA	
RF	95.827	95.748	95.827	95.748	
GB	95.905	95.695	<b>95.905</b>	95.900	
SVM	93.952	93.952	91.747	93.572	
Algorithm	The profiler metrics				
	RF	<b>96.323</b>	95.796	95.827	95.394
	GB	96.297	95.993	95.905	95.939
	SVM	95.399	95.399	93.952	93.149

#### 4.3.2.5. Classification Results

We build classification models for SDC and crash rate predictions to estimate the vulnerability class instead of predicting the exact value. We present our results for both 2-class and 3-class models, where we define different classes by considering the fault rate intervals of the kernel functions. Instead of predicting the probability of taking place SDCs or crashes, classifying their occurrence probability as high or low while the

application is running and obtaining reliable results gives an idea about the soft error-proneness of the GPU application. This approach illustrates the proneness of a GPU kernel to the SDCs or crashes with reliable prediction results, especially with 2-class experiments.

Table 4.7. SDC and Crash rates classification results for 2/3-class evaluations among all preprocessing methods.

	Algorithm	Feature Selection Method	Prediction Accuracy	
			Simulator Fea- tures	Profile Fea- tures
2-Class SDC / Crash	RF	All Features	80.769 / 72.727	77.143 / 83.333
		Generic Univariate Selection	80.769 / 80.000	76.316 / 79.545
		Manual Feature Selection	76.923 / 70.270	76.000 / 81.818
		PCA	84.615 / 79.545	<b>84.000</b> / 79.545
	GB	All Features	78.378 / 75.676	80.556 / 85.000
		Generic Univariate Selection	76.923 / 78.049	75.000 / 79.545
		Manual Feature Selection	83.784 / 72.222	75.676 / 82.500
		PCA	<b>88.462</b> / 82.500	76.000 / 79.545
	SS+SGD	All Features	70.732 / 80.769	75.000 / <b>85.714</b>
		Generic Univariate Selection	73.684 / <b>84.211</b>	79.545 / 78.947
		Manual Feature Selection	80.488 / 78.378	73.684 / 83.784
		PCA	79.545 / 78.947	79.545 / 82.857
3-Class Crash	RF	All Features	63.415 / 60.976	65.909 / <b>73.913</b>
		Generic Univariate Selection	65.714 / 68.571	65.909 / 71.053
		Manual Feature Selection	68.293 / 60.976	65.909 / 69.565
		PCA	66.667 / 68.889	65.909 / 68.889
	GB	All Features	67.568 / 64.865	67.500 / 66.667
		Generic Univariate Selection	62.162 / 67.857	62.500 / 71.053
		Manual Feature Selection	<b>75.676</b> / 65.854	68.889 / 74.286
		PCA	67.568 / <b>71.053</b>	70.000 / 71.053
	SS+SGD	All Features	61.765 / 70.270	65.909 / 68.571
		Generic Univariate Selection	70.968 / 68.571	68.293 / 68.182
		Manual Feature Selection	73.529 / 63.415	68.421 / 68.182
		PCA	68.293 / 68.182	<b>73.684</b> / 69.565

Table 4.7 presents our 2-class and 3-class evaluations with varying feature selection and classification approaches. Each classification approach provides different accuracy results for each feature selection approach. To illustrate, in a scenario where we use simulator features while we obtain 88.46% prediction accuracy to classify 2-class SDCs with the *PCA* and *GB* algorithm, the obtained maximum prediction accuracy for 2-class Crashes is 84.21% with pipelined *SS+SGD* algorithms. Various feature selection approaches change the accuracy results for each classification target even if we use the same classification algorithm. For example, *GB* generates the most accurate prediction results for both 2-class and 3-class SDCs, where we reach these results with *PCA* and *Manual Feature Selection*, respectively. Based on these results, we can conclude that no generic ML-based classification approach or feature selection method fits all scenarios with different targets. Adversely, each classification has its unique characteristics due to varying fault rates.

The most accurate prediction results with the simulator features are 88.46%/84.21% and 75.67%/71.05% for 2/3-class SDCs and crashes, while the same prediction results with profiler features are 84%/85.71%, 73.68%/73.91%, respectively. Unlike the predicting masked faults, the results confirm that simulator features generate more accurate predictions than the profiler metrics for predicting SDC and crash rates. Since the fault injection tool injects faults into registers by assuming that memory cells are saved from the transient faults with ECCs, register usage metrics are more effective for the predictions. Simulator metrics include instructions and register usages such as instruction types and intensities. In general, we can say that simulator metrics characterize the kernel behavior in addition to the hardware usage metrics with the PTX code. In other words, the simulator metrics are more beneficial in representing the application's structure. Rather than the GPU hardware usage metrics provided by the profiler, metrics that provide quality and quantity regarding the application's register usage are more crucial according to the results of the prediction experiments.

The classification results reveal that we can utilize the classifiers for soft error vulnerability prediction of GPU programs. When we define the problem as a classification problem to obtain the vulnerability level of the target program, we can predict in which range we expect to see SDC or crash conditions. This evaluation enables us to understand how vulnerable the program is, even if we cannot predict the absolute SDC or crash rates. We can decide whether to perform fault tolerance techniques based on the classification outcome.

## 4.4. Summary

To sum up, we predict occurrence rates of SDCs, crashes, and masked faults with the help of a comprehensive ML-based framework trained with the simulator and the profiler metrics. Since SDCs and crashes are observed dramatically less than the masked faults, we use a classification method to determine the error vulnerability conditions instead of the regression approach. We benefit from various preprocessing tools, feature selection/reduction approaches, and outlier eliminators to increase the accuracy results of the prediction experiments. We carry out 384 experiments for the prediction of 2/3-class SDCs and crashes separately and 1536 experiments for the prediction of masked faults. These experiments provide detailed comparisons for preprocessing methods, feature selection/reduction approaches, outlier eliminations, machine learning model selections, and evaluation of the relationship between micro-architectural metrics and soft error vulnerability. While the experiments with the simulator features result in more reliable prediction accuracy results for the 2/3-class SDCs and crashes, profiler features provide more accurate prediction results for predicting masked faults with the regression approach. As a result of the experiments, we achieve 95.91%, 88.46%, and 85.71% prediction accuracy results for masked fault, SDC, and crash rates, respectively.



## CHAPTER 5

### **APPROXTRACKER: MEMORY-DRIVEN GPU APPROXIMATOR ENHANCING PERFORMANCE, ENERGY EFFICIENCY, AND DATA UTILIZATION**

A GPU executes thousands of parallel threads that utilize memory concurrently and seriously cause memory traffic in memory partitions degrading memory utilization and performance. GPUs may become chronically inefficient in data processing tasks where the data is sparse to benefit from the locality on caches, and today's several data-parallel tasks cannot employ heavy-processing power effectively. That problem has been tried to be optimized to achieve better performance and energy dissipation in literature (Chatterjee et al. (2014); Koo et al. (2017); Singh and Nasre (2020); Vijaykumar et al. (2018); Zhao et al. (2019)). OWL (Jog et al. (2013)) controls the number of active thread blocks issued to the SMs (i.e., decreasing parallel executing threads) depending on the memory workload reduce memory limitations. Furthermore, recent near-/in-memory research (Pattnaik et al. (2019)) reduces the negative impacts of heavy memory workload and destructive irregular behavior without overloading memory hierarchy by spatial memory requests. Enhancing the performance and energy consumption of irregular computational workloads with software or hardware solutions eliminates the drawbacks and widens their usage among real-world applications, and various approximate computing methods considering execution error resiliency investigate the generic usability limits.

Some state-of-the-art approximate computing techniques on hardware utilize small lookup tables to benefit from operand similarity on SMs, which eliminate redundant and unnecessary computations (Garcia et al. (2021); Rahimi et al. (2016)), kernel perforation to increase memory performance (Maier et al. (2019)) and in-memory data comparison (Choi et al. (2022)) which reduces memory workload and obtains compute-memory workload balance within tolerable accuracy borders. On the other side, approximating either gradient loss computation (Wang et al. (2019)) or matrix multiplications (Imani et al. (2019)) iteratively for deep learning applications and partially processing graphs with approximating graph attribute values (Singh and Nasre (2018)) are viable software-based approximation offerings on GPUs. In addition, ML-based strategies enable selecting optimal approximate computing methods depending on the usage requirement considering

performance and accuracy (Aktılav and Öz (2022)). By viewing the realized approaches, monitoring the micro-architectural metrics at runtime together explicitly reveal chronic performance bottlenecks, which may differ depending on the application domain, and inform the propagation of the approximated values throughout the execution on a GPU. We investigate and clarify that parallelizable applications processing massive datasets, especially with irregular memory access behavior like graphs, worsen the performance and energy consumption through *GPPRMon* tool in Section 3.3.1.

Enhancing the functioning of memory hierarchy with approximate computing can allow for overcoming memory bounds and bring compute-memory workload balance. This study solves the commonly faced memory exploitation problems with approximation-based solutions. We propose multi-functional ApproxTrackers, which mitigate the memory workload by tracking per-memory component locality information and skipping the memory requests at local memory regions temporally at runtime. To the best of our knowledge, this is the first study that temporally flushes memory requests based on the local memory components' performance considering the error propagation at runtime. Our major contributions are as follows:

- We develop multi-functional approximate computing approaches, ApproxTrackers, which flush memory requests depending on the regional memory traffics with the feedback obtained via L1D and L2 caches at runtime. We experiment ApproxTrackers for two scenarios: 1) ApproxTrackers with various GPU algorithms, which allows evaluating the performance and energy impacts of proposed approaches on different execution domains, and 2) ApproxTrackers with the sparse matrix multiplication algorithm and various graphs, which enables interpreting performance and energy impacts of processing different datasets.
- ApproxTrackerL1D, which operates on L1D caches, improves the performance of applications from various domains by 18.6% and reduces energy consumption by 14.8%.
- ApproxTrackers acquires 1.495× performance increase on a sparse matrix multiplication algorithm experimented with various data by flushing memory requests depending, and lower the overall energy consumption by 28.4%.

## 5.1. Related Work

Garcia et al. (2021) provide a framework, TruLook, that leverages approximate computing techniques to enhance GPU acceleration by minimizing redundant and unnecessary exact computations. It achieves this by utilizing computation reuse and approximate arithmetic operations. The framework incorporates small lookup tables near the stream cores of the GPU architecture, enabling the retrieval of pre-computed values for both exact and potentially inexact matches. The level of inexact matching is controlled by a threshold determined by the number of mantissa bits involved in the search. The hardware configuration of TruLook can be adjusted at runtime to ensure the required level of accuracy for a given application. In experimental evaluations on the ImageNet dataset using a quality loss budget of 0% and less than 1%, TruLook demonstrates an average energy-delay product improvement of 2.1× and 5.6× over four popular networks. TruLook differs from our approach in terms of handling the approximation targets during the execution. While our approach, ApproxTrackers, aims to eliminate memory bottleneck problems, TruLook offers reducing SMs' computation workload.

Nvalt (Imani et al. (2018)) utilizes a nonvolatile approximate lookup table to accelerate computations on general-purpose GPUs by storing frequently occurring input patterns within an approximation. Nvalt searches for and retrieves the stored data that best matches the input to generate an approximate output. To measure the similarity between binary representations, they leverage the analog characteristics of nonvolatile content addressable memory and define an appropriate similarity metric. By controlling the ratio of application execution between the approximate Nvalt and the accurate GPU cores, the design allows for tuning the level of accuracy based on user requirements. As a result of their evaluations, Nvalt demonstrates an average energy improvement of 4.5× and a performance enhancement of 5.7× compared to a baseline GPU while maintaining an average relative error of less than 10%. On the other side, our approach tracks the runtime memory performance via L1D cache and L2 caches and conducts flushing among the memory requests by keeping the accuracy distraction at most 2%.

Maier et al. (2019) and Maier et al. (2018) focus on approximating memory-bound applications running on mobile GPUs and computationally rich GPUs, respectively. Both approaches introduce kernel perforation, which takes advantage of the fast local memory available in GPUs to achieve high performance while maintaining more accurate results. They experiment with kernel perforation to six different applications for mobile GPUs and conduct evaluations on the Qualcomm Adreno 506 and the ARM Mali T860 MP2

architectures. They state that even when the local memory is not specialized as dedicated fast memory in the hardware, kernel perforation still yields a speedup of  $1.25\times$  due to improved memory layout and caching effects on mobile GPUs. For the bigger GPUs, their approach first skips the loading of parts of the input data from global memory and uses reconstruction techniques on local memory later to reach higher accuracy. Experimental results with bigger GPUs demonstrate the effectiveness of our approach in accelerating the execution of diverse applications, and their speedup ranges from  $1.6\times$  to  $3\times$ , indicating a significant performance improvement. ApproxTrackers targets to solve similar memory bottleneck problems at runtime by skipping memory requests depending on the memory utilization feedback from both on-chip and off-chip memory units.

Singh and Nasre (2019) addresses irregular memory accesses and control-flow-associated challenges for graphs by introducing approximations to mitigate their impact. They employ several techniques to enhance memory coalescing, reduce memory access latency, and minimize thread divergence. They renumber and replicate nodes to improve memory coalescing, facilitating better data alignment and coalesced memory accesses. Adding edges among specific nodes brought into shared memory mitigates memory latency, reducing the time required to access data. Additionally, they normalize degrees across nodes assigned to a warp, mitigating thread divergence and improving parallelism. Their approximations for coalescing, memory latency, and thread divergence result in mean speedups of  $1.3\times$ ,  $1.41\times$ , and  $1.06\times$ , respectively, while maintaining the accuracy bounds at 83%, 78%, and 84%, respectively. ApproxTrackers offer a better performance targeting the same application domain while they conduct approximations on the hardware. Unlike their research, by keeping the accuracy distraction below 2%, our research accelerates the graph workloads by an average of  $1.75\times$ .

## 5.2. Methodology

Our purpose is to obtain memory/compute workload balance on GPUs through the execution, resolving bottlenecks in memory hierarchy at runtime to improve performance and mitigate power consumption. ApproxTrackers mainly eliminate memory operations selectively by depending on the runtime performance of components tracked with the micro-architectural metrics and enhance GPU memory performance for error-resilient applications. Since GPUs employ on-chip memory units privately within each SM and off-chip memory partitions among all execution workloads, local metrics cannot address performance bottlenecks throughout the entire memory hierarchy; and determin-

ing representative micro-architectural metrics to track memory performance locally and temporally is essential. To illustrate, solely tracking misses in the memory unit of an SM provides information only for the performance of that particular SM and does not represent the utilization of other memory components' performance. Instead, hybrid and synchronized tracking of each unit's memory usage will enable us to manage each memory unit separately, such as applying solutions depending on the local behavior of each component at runtime. Therefore, we developed the ApproxTrackers to track each memory component's local utilization at runtime by benefiting from the detailed representative micro-architectural metrics and decide to flush a memory request from the pipeline accordingly.

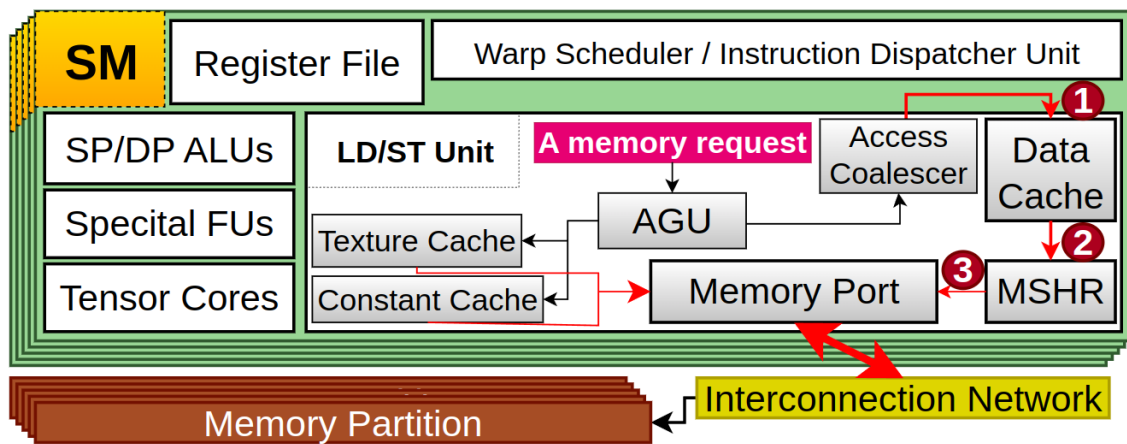


Figure 5.1. Execution flow of a memory request starting from LD/ST unit.

Figure 5.1 illustrates the execution flow of memory requests on SMs and off-chip memory partitions, whose internal structure is detailed in Figure 2.4 (Aamodt et al. (2018)). These figures describe how a memory request is handled within a GPU and do not depict the resource quantities and internal structures for any real architecture. LD/ST units on SMs set the target address of the memory request in the address generator unit (AGU). If a memory request needs to access a specialized memory unit, such as constant memory, LD/ST unit may direct it accordingly. Otherwise, LD/ST unit transmits memory requests to the L1D cache after coalescing. When the memory request's access status is hit or hit\_reserved on the L1D cache, the request returns on the 1st path in Figure 5.1 and reaches the writeback stage of SM's execution pipeline. Conversely, memory requests follow the 2nd path when the looked-for data does not exist on the L1D cache, in which the access status is either miss or sector miss. The Miss-Status-Holding-Register

(MSHR) unit tries to merge the missed request with other misses that look for the same data. Additionally, reservation failures due to resource insufficiency can occur on 2<sup>nd</sup> and 3<sup>rd</sup> paths, and these failures cause memory pipeline stalls because the GPU must continue execution without losing the missed accesses.

### 5.2.1. ApproxTrackerL1D

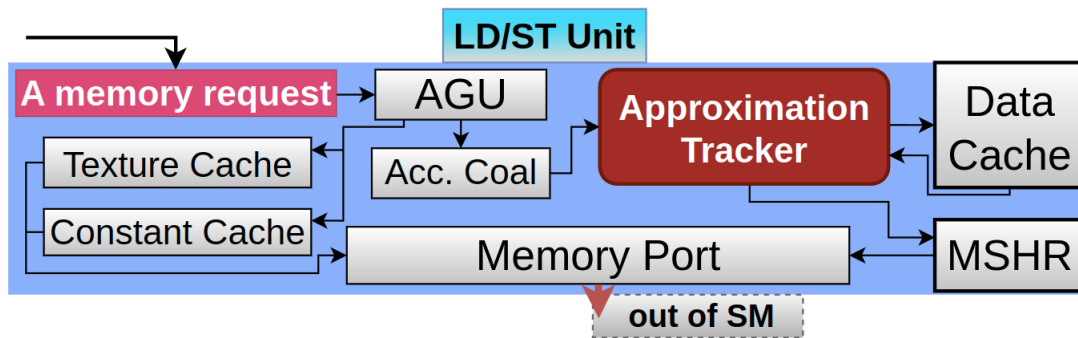


Figure 5.2. Locating ApproxTrackerL1D within the LD/ST unit on SMs.

Resolving the bottlenecks occurring on only one L1D cache does not significantly impact other SMs' or lower memory partitions' memory workload. To manage the whole memory workload, we propose the *ApproxTrackerL1D* approach at first, which directly manages the bottlenecks of all L1D caches comprehensively, and Figure 5.2 shows where it resides on an SM.

ApproxTrackerL1D algorithm decides to either flush the memory request or keep it in the memory pipeline based on the access statistics of L1D caches that are informative about the memory traffic on SMs directly and on memory partitions implicitly and Algorithm 1 shows the approximation procedure followed by ApproxTrackerL1D. The *m\_stats* array, at 2<sup>nd</sup> Line, counts the number of hit, hit\_reserved, miss, and sector\_miss accesses consecutively at runtime for each L1D cache. ApproxTrackerL1D tracks per L1D cache statistics depending on the status information of consecutive accesses temporally, *m\_stats*, by employing a slicing interval. Employing a slicing window, counted by *accessCounter*, allows detecting temporal performance degradation at the runtime without converging the entire kernel behavior. For example, assigning the slicing window for 500 consecutive memory accesses through an L1D cache provides acting depending on the temporal statistics of the corresponding accesses.

---

**Algorithm 1:** ApproxTrackerL1D approximation check controller algorithm

---

```
1 bool checkFlushOnL1DCache ()
2 totalAcc  $\leftarrow m_{stats}[0] + m_{stats}[1] + m_{stats}[2] + m_{stats}[3]$ 
   // Calculates total accesses within the slicing interval
3 currStats  $\leftarrow 0$ 
4 if totalAcc then
   // currStats stands for the miss rate within the slicing interval
5    $currStats \leftarrow (m_{stats}[2] + m_{stats}[3])/totalAcc$ 
   /* 1) currStats  $\geq$  miss rate threshold for the corresponding kernel || 2)
   Wait for the first 16 accesses within the slicing interval || 3)
   flushCounter controls the error percentage */
6 if (currStats  $\geq$  L1DMissThreshold[*kernel_id]) & (accessCounter  $\geq$  16) &
   (flushCounter[*kernel_id]  $\geq$  0) then
7    $\left\{ \right.$  return true
8 return false
```

---

Additionally, deciding to flush a memory request from a local memory pipeline portion before temporal warm-up at runtime may prevent utilizing data locality. Thus, ApproxTrackerL1D waits for 16 consecutive memory requests on the corresponding cache before deciding to flush any memory request till the L1D cache temporally warms up. Waiting for at least 16 consecutive accesses, tracked with *accessCounter* at 6th Line, aims to track temporal locality on L1D caches and conduct flushes according to inter- and intra-warp locality feedback (Rogers et al. (2013)). In this way, ApproxTrackerL1D deploys the approximation algorithm temporally, considering data locality per L1D cache. We design ApproxTrackers so developers can configure the data locality threshold for each kernel and each L1D cache.

The Algorithm 2 shows the counter statistic updater algorithm of ApproxTrackerL1D. ApproxTrackerL1D updates statistics depending on *afterFlush* flag. If no flush occurs (i.e., *afterFlush* is false), ApproxTrackerL1D updates the *m\_stats* counter depending on the access status of the corresponding memory request by executing the block beginning at 10'th Line. It increments the *accessCounter*, which controls the slicing interval limits. If the *accessCounter* reaches the pre-configured slicing window size, all statistics are cleared to track the following consecutive memory accesses. On the other side, if the ApproxTrackerL1D activates the updater just after flushing a memory request in which *afterFlush* flag is true, the code block starting at 4'th Line is executed, which decrements the error tracker (*flushCounter*), increments the slicing window flush counter

(*intervalCheck*). Additionally, *ApproxTrackerL1D* clears all the statistics and starts a new observation interval again if the flushing of 8 memory requests has been made within the same interval. In this way, *ApproxTrackerL1D* deletes memory requests whenever the locality threshold is exceeded and waits some time to observe the upcoming execution behavior of the L1D cache. The LD/ST unit pipe depth of Volta GPUs is capable of 8 consecutive memory requests, and we aim to capture all the missed ones in the pipeline when the locality is lower than the threshold.

---

**Algorithm 2:** *ApproxTrackerL1D* tracking update algorithm

---

```

1 void updateApproximatorL1D (status, afterFlush)
2 accessCounter  $\leftarrow$  accessCounter + 1           // Increment slicing window counter
3 if afterFlush then
4     flushCounter[kernel_id]  $\leftarrow$  flushCounter[kernel_id] - 1 // Decrement number
        of expected flush counter
5     intervalCheck  $\leftarrow$  intervalCheck + 1     // Increment number of flush counter
        within the slicing interval
6     if intervalCheck > 8 then
7         // If 8 accesses flushed within the slicing interval, clear stats
8         intervalCheck  $\leftarrow$  0, accessCounter  $\leftarrow$  0
9         for i  $\leftarrow$  0 to NUM_CACHE_REQUEST_STATUS by 1 do
10             $\left[ \begin{array}{l} m_{stats}[i] \leftarrow 0 \end{array} \right.$ 
11 else
12     m_stats[status] ++           // Update cache statistics considering the status
13     if accessCounter  $\geq$  accessSliceInterval then
14         /* Clear stats for the new slicing window interval */
15         accessCounter  $\leftarrow$  0
16         for i  $\leftarrow$  0 to NUM_CACHE_REQUEST_STATUS by 1 do
17             $\left[ \begin{array}{l} m_{stats}[i] \leftarrow 0 \end{array} \right.$ 

```

---

*ApproxTrackerL1D*'s updater module employs a 64-bit *flushCounter* to control the maximum memory access to be deleted from the corresponding SM for a kernel. Setting the *flushCounter* to a pre-determined value limits the disruption on the output and enables us to handle accuracy. Even if it does not directly control the accuracy bounds because flushing a memory request may disturb output in a nondeterministic manner, it essentially allows adjusting accuracy within tolerable limits. The developer sets an error



percentage, and each ApproxTrackerL1D module assigns an upper bound for maximum flushable accesses obtained with  $error \times 0.01 \times naiveAccess_{ith\_sm}$  where the number of  $naiveAccess_{ith\_sm}$  corresponds to the number of memory requests on  $i$ 'th SM's L1D cache resulting from the pre-simulation. That is, each SM has its own approximation limitation through the above algorithms. Besides the methodology of flushing memory requests, ApproxTrackerL1D carries out approximations by replacing data that would be loaded with zeros in the writeback execution pipeline stage. Consequently, ApproxTrackerL1D implements an L1D cache-based approximation by following the temporal data locality utilization of each L1D cache and reducing the memory workload at critical times, which improves performance and power consumption.

## 5.2.2. ApproxTrackerL2

According to the cycle-accurate GPU simulator official configuration details<sup>1</sup> (Khairy et al. (2020)), the average latency to access the DRAM row buffers (without activating DRAM cells) is fifteen times higher than to L1D cache and three times higher than the L2 cache for Volta GPUs respectively, and the access latency to memory units varies among the GPU architectures. In addition to the latency variation among the GPU devices, the memory traffic that depends on application type in memory partitions significantly affects the performance, as demonstrated in Section 3.3.1. ApproxTrackerL1D, which attempts to handle off-chip memory traffic by tracking L1D cache exploitation, might not become comprehensive enough without considering off-chip memory partitions, especially if no specific traffic patterns exist in the memory partitions.

ApproxTrackerL2 tracks memory partition behaviors, removes memory requests selectively from the pipeline before to be issued to the off-chip by LD/ST memory ports, and aims to reduce the off-chip memory traffic whenever the memory performance is bottlenecked at runtime. ApproxTrackerL2 conducts approximations similar to ApproxTrackerL1D in terms of approximation algorithm and approach. It tracks the temporal efficiency on each L2 cache within sub-memory partitions with a slicing window counted with *accessCounter* and a statistic counter array (i.e.,  $m_{stats}$ ) as in the code Algorithm 1. ApproxTrackerL2 conducts the approximations by flushing the memory requests in SMs without forwarding them to off-chip memory partitions. ApproxTrackerL2 operates based on the same rules as ApproxTrackerL1D, such as tracking 16 consecutive memory requests on the same L2 cache before starting flushing or flushing a maximum of 8

<sup>1</sup>[https://github.com/gpgpu-sim/gpgpu-sim\\_distribution/tree/master/configs/tested-cfgs/SM7\\_QV100](https://github.com/gpgpu-sim/gpgpu-sim_distribution/tree/master/configs/tested-cfgs/SM7_QV100)

memory requests within the same slicing interval at most.

The hardware implementation of ApproxTrackerL2 deploys to architecture more complexly than ApproxTrackerL1D. It employs synchronized multiple modules across the coalescer unit, interconnection network, and sub-memory partitions. The module in the coalescer unit leverages similar address decoders of the interconnection network to obtain the target off-chip memory partitions to which memory requests are transmitted. Hence, ApproxTrackerL2 makes flush decisions for memory requests whose off-chip target addresses are already known within SMs accordingly, which enables acting each memory partition individually depending on the performance bottlenecks.

Before beginning execution on GPU, the updated coalescer unit takes the miss rate thresholds for each L2 cache and holds them in a list. Copying address decoders to the coalescer unit does not bring performance overhead since memory requests are still routed through the interconnection network decoders. We develop ApproxTrackerL2 such that developers can configure the slicing window size, maximum flushable memory accesses, and error percentage depending on their application’s performance requirements and accuracy limits. Lastly, ApproxTrackerL2 writes zero to the operands of the flushed memory requests.

## **5.3. Experimental Study**

### **5.3.1. Experimental Setup**

For the experimental evaluation, we test ApproxTrackerL1D/L2 with six applications as in Table 5.1 from PolyBench, Rodinia, TangoDNN, and Gardenia benchmark suites (Che et al. (2009); Grauer-Gray et al. (2012); Karki et al. (2019); Xu et al. (2019)). We mainly target memory-bound applications, but we include various contemporary domains and observe ApproxTrackers’ performance and energy consumption impacts on them. While TangoDNN, PolyBench, and Rodinia applications come with proper test datasets, Gardenia has trivial datasets. To evaluate the impact of data size and sparsity dependency, we test ApproxTrackers with the SPMV algorithm on Higgs Twitter, lp1 sparse matrix, web-Google and live journal online social network (soc-LiveJournal) data collected from the Suite-Sparse Matrix Collection datasets as in Table 5.2.

Table 5.1. CUDA applications used in our experiments.

Application		Description
<b>PolyBench</b>	2DConvolution	Convolution 2D input data and 2D mask.
<b>Rodinia</b>	HybridSort	Fusion of merge sort on bucket sort algorithms.
<b>TangoDNN</b>	GRU	Gated Recurrent Neural Network
	LSTM	Long/Short term memory Recurrent Neural Network
<b>Gardenia</b>	Connected Components (CC)	Afforest & Shiloach-Vishkin algorithms
	SPMV	Sparse Matrix-Vector Multiplication

Table 5.2. Graph specifications.

Graph		Description
<b>Higgs Twitter</b> , (Domenico et al. (2013))	Main Social Network	#V = 456626, #E = 14855842.
	Retweet	#V = 256491, #E = 328132.
	Mention	#V = 116408, #E = 150818.
<b>Others</b> , (Leskovec et al. (2009b))	LP1	#V = 534388, #E = 1109032.
	LiveJournal online social network ,	#V = 4847571, #E = 68993773.
	Web graph from Google	#V = 875713, #E = 5105039.

As simulating all applications require too much time, we first profile 60 CUDA workloads utilizing Nsight Compute tool (NVIDIA (2022a)) on NVIDIA’s RTX 1650, extract the micro-architectural metrics<sup>2</sup>, and create an experimental target application set where we will investigate the ApproxTrackers’ impact explicitly. Afterward, we naively simulate target applications, collect performance and driven-runtime power metrics, and obtain the exact outputs. To evaluate approaches, we configure two experiment scenarios for ApproxTrackerL1D in which each SM deletes its 2% and 1% of memory requests that will probably bring misses on L1D caches. Similarly, we create another two test setups for ApproxTrackerL2 that flushes 2% and 1% of memory requests that will cause misses on L2 caches and create traffic on off-chip memory partitions, respectively.

During experiments, we set the slicing window size to 5000 memory requests that are lower than 1% of overall memory accesses of a cache on average among target applications. The purpose is to act temporally (i.e., neither too short nor too small)

<sup>2</sup>[https://github.com/BT-MasterThesis-2020-23/Application\\_PreProfiling\\_onNCU](https://github.com/BT-MasterThesis-2020-23/Application_PreProfiling_onNCU)

depending on runtime performance during the execution. Furthermore, we utilize the individual memory exploitation metrics obtained through the base simulation of each kernel for the per-cache threshold values to determine the flushing of a memory request. We specify the maximum flushable memory requests per memory unit based on the number of missed memory requests on the corresponding unit observed during the base execution. For example, if the execution causes 234567 misses and sector misses among the memory requests with a 0.4 miss rate on the L1D cache of SM0, ApproxTrackerL1D assigns the threshold of SM0’s L1D cache as 0.4 and can flush either 1% or 2% for those misses at runtime.

We implement ApproxTracker by extending GPGPU-Sim v4.2 (Khairy et al. (2020)) and collect the power metrics by AccelWattch (Kandiah et al. (2021)) extension of the simulator. Then, we simulate the target CUDA applications through RTX2060 GPU configuration (TechPowerUp (2020)). Table 5.3 shows the RTX2060 hardware resources. Since we conduct experiments through NVIDIA GPUs, we utilize the CUDA Toolkit version 11.6 and the corresponding NVCC compiler. One can reach results for pre-profiling experiments with Nsight Compute tool, simulation results for naive and each ApproxTracker experiment, and the updated version of the simulator source code via the link<sup>3</sup>.

Table 5.3. RTX2060 GPU configuration based on Turing architecture.

Streaming Multipro- cessor Specs (30)	Register bank size, # of register bank	65536 32-bit registers, 16 register banks
	SP, SF, DP, INT, TC, LD/ST (WB-Depth)	4, 4, 4, 4, 4, 1(8)
	Warp Scheduler	4 (LRR) per SM
	L1D Cache, #of banks, latency, line size	128KB, 1, 32 cycles, 128B
	L1I Cache, #of banks, latency, line size	128KB, 1, 32 cycles, 128B
12 mem, 24 sub-mem partitions.	L2 Cache, #of banks, latency, line size	128KB, 2, 194 cycles, 128B
	DRAM, #of banks, latency (after L2)	512MB, 12, 96 cycles, 128B
	DRAM scheduler	First-ready, first-come first-service

*SP*: Single Precision, *SF*: Special Functional, *DP*: Double Precision, *INT*: Integer, *TC*: Tensor Core, *LD/ST*: Load / Store, *WB*: Write back

<sup>3</sup><https://github.com/BT-MasterThesis-2020-23/ApproxTracker>

### 5.3.2. Experimental Results

We evaluate the experiment results with two different approaches. While we test ApproxTrackerL1D/L2 on applications from various domains in terms of performance and power results, we experiment ApproxTrackers on a memory-bound program by changing the processed data to reveal the significance of processed data on the execution and how ApproxTrackers handles them. In all results below, ApproxTrackers' -2 and -1 suffixes stand for flushing missed requests of 2% and 1%, respectively.

We evaluate the accuracy of applications in the PolyBench, Rodinia, and Gardenia benchmark suites based on data corruption in the whole output. Assuming that applications from Gardenia Rodinia and PolyBench suites produce a total of  $X$  data as output, we define accuracy based on the percentage of corrupted data in the output applications. For example, the fact that the application generates 100 output values and two of them are corrupted during the experiments states that ApproxTrackers completes the execution at 98% accurately. In TangoDNN applications, accuracy is defined based on the prediction success depending on the trained DNN models. In these experiments, we define accuracy based on how much ApproxTrackers distorted the prediction accuracy.

In addition, we designed ApproxTracker to flush only load operations among both stores and loads. The purpose is to reduce the probability of worsening accuracy caused by deleting data to be stored. However, in some CUDA applications, the access we flushed coincided with pointers, causing the applications to crash. To illustrate, Breadth First Search (BFS) and Connected Components of Afforest & Shiloach-Vishkin algorithms (CC) from the Gardenia benchmark terminate with the crash state when experimented with ApproxTrackers.

In applications mentioned in 5.1, skipping memory requests either directly corrupts or several flushes together corrupt a value in the output. Therefore, by configuring 2% and 1% of the overall memory requests to skip, we limit the errors to below 2%. Hence, the accuracy deviation for the results is at most 2%. Furthermore, flushing memory requests for some applications corrupts the output by around 0.1% because errors do not propagate throughout the execution. One can find all the results in the experimental\_results file in the previously mentioned GitHub repository.

### 5.3.3. ApproxTracker Performance and Driven-Power on Various GPU Applications

Table 5.4. Baseline performance overview among target applications.

Application	Total Cycle	IPC	Occup.	L1D Miss	L2 Miss	RB Loc.	Stall Cycle
SPMV	21449970	5.90	35.07%	0.553	0.324	0.182	4 989 317
CC	3608674	14.55	40.75%	0.626	0.108	0.128	60 786
2DConv	1295864	582.24	60.85%	0.246	0.705	0.936	0
GRU	16015599	2.85	48.51%	0.975	0.001	0.818	50 893
LSTM	509423	88.61	46.79%	0.223	0.014	0.961	0
HybridSort	1336354	217.05	45.22%	0.786	0.805	0.876	342

Table 5.4 shows performance overview metrics for target applications. *Total Cycle* elapsed during the simulations informs about the execution duration, and *Stall Cycles* stand for the cycles wasted during the memory pipeline locks during the runtime. *Occupancy* metric describes the overall GPU utilization depending on SM and memory hierarchy throughput values. SPMV, CC, GRU, and HybridSort applications cause high miss rates on L1D and low utilization on DRAM row buffers, and the low memory performance decreases *IPC* rates for these applications. 2DConvolution employs the hardware the most efficiently among the target applications considering the performance metrics such as *IPC* and *Occupancy*. The GRU and LSTM applications from TangoDNN result in similar *Occupation* values, while GRU completes the execution dramatically longer than LSTM. Some GRU's kernels limit the *Occupation* for some of the kernels, which extends the execution (i.e., *Total Cycle*) and lowers the GRU's *IPC*. Conversely, LSTM completes the execution fastest among the target applications since it has a small workload compared to others. Even if HybridSort seems to cause too many misses on both L1D and L2 caches, the corresponding memory requests are compulsory misses. According to the *IPC* and *Occupancy*, the overall performance of HybridSort outperforms SPMV and CC, although the application cause many misses.

Figure 5.3 displays the cache miss rates on L1D and L2 caches experimented with ApproxTrackers, which flushes 2% and 1% of memory requests depending on the runtime

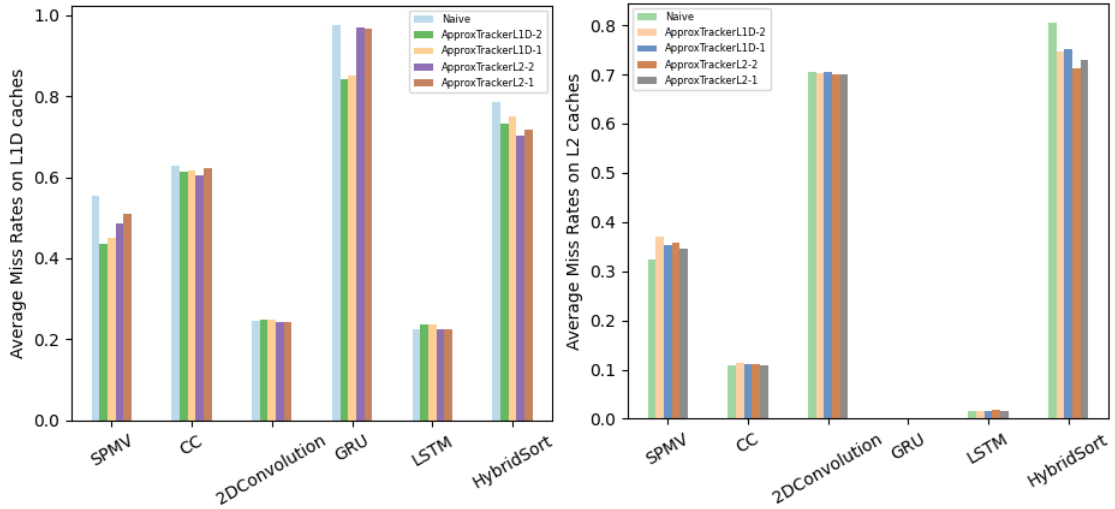


Figure 5.3. Average miss rates on L1D and L2 caches experimented with naive, ApproxTrackerL1D and ApproxTrackerL2 versions.

memory performance, and Table 5.5 shows the number of memory access statistics on caches. ApproxTrackers generally decreases the miss rates on L1D caches, as explicitly shown in SPMV, GRU, and HybridSort results. ApproxTrackerL1D significantly reduces the L1D cache miss rates because it flushes more memory requests based on directly each L1D cache utilization on each SM. For example, ApproxTrackerL1D decreases the average miss rate on L1D caches from 0.55 to 0.43 and 0.97 to 0.82 for SPMV and GRU applications, respectively. Moreover, ApproxTrackers can diminish the miss rates on L2 caches (sub-memory partitions) for the applications such as HybridSort. In contrast, the proposed approaches do not further lower the miss rates of the applications that can utilize L2 caches well. The L2 cache usage behavior for some applications, such as 2DConvolution and LSTM, does not oscillate much during the execution, so ApproxTrackers cannot detect an imbalanced workload on off-chip memory partitions for flush during runtime.

Table 5.5 shows the number of memory accesses on caches with naive with and ApproxTrackers. One can realize that applications whose runtime performance is significantly affected by ApproxTrackers, like HybridSort and SPMV, have fewer memory access reaching L1D and L2 caches. For the applications with few instances of miss rates above the threshold, ApproxTrackers still decrease the number of memory accesses. However, since the changed runtime cache behavior does not affect the remaining accesses, they yield similar cache access rates. For example, even though ApproxTrackers reduce the number of accesses reaching L1D caches, the corresponding overall miss rates do not further lower. Additionally, applications that process small datasets directly can

utilize L2 caches well because the data fits into it, and ApproxTrackerL2 does not change such applications' memory usage behavior drastically as in GRU and LSTM. Regarding reducing the memory workload, ApproxTrackerL2 outperforms ApproxTrackerL1D for applications utilizing other on-chip memory components like texture cache more since ApproxTrackerL1D performs flush operations based on L1D cache statistics. To illustrate, HybridSort cause more pressure on memory partitions than L1D caches because most on-chip memory requests access to the texture cache. Hence, ApproxTrackerL2 can decrease the off-chip memory workload more which is more crucial for performance. According to the results, we observe that applications that employ caches efficiently and cause low miss rates do not expose too many distractions on cache behavior, and those miss rates generally exist throughout the entire kernel without oscillating. Therefore, ApproxTrackers can slightly change these applications' runtime behavior and performance. On the other hand, ApproxTrackers seriously change the execution behavior of applications with irregular memory accesses on the memory hierarchy.

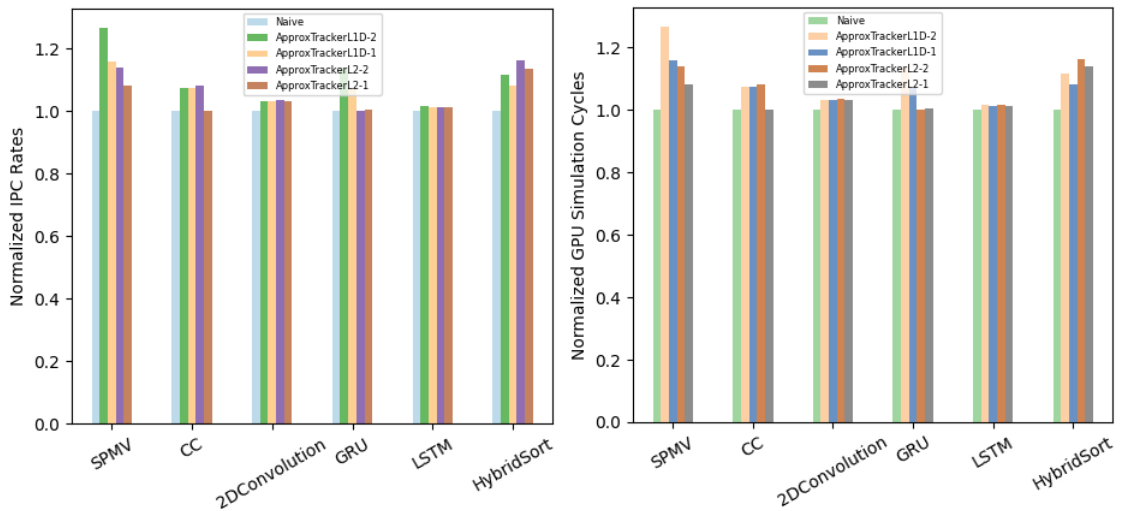


Figure 5.4. Normalized IPC and GPU Simulation Cycle experimented with naive, ApproxTrackerL1D, and ApproxTrackerL2 versions.

Figure 5.4 shows ApproxTrackers' impact on the IPC and GPU simulation performance rates. GPU simulation performance stands for the total simulation cycles elapsed during the execution, and we normalize each experimental result according to the baseline experiment result. ApproxTrackerL1D, particularly configured to flush 2% of memory requests, directly impacts overall IPC among SMs and execution performance by solving high miss rates on caches during the execution. ApproxTrackerL1D raises the average



Table 5.5. The number of memory accesses on caches experimented with naive, ApproxTrackerL1D, and ApproxTrackerL2 versions.

<b>Applications</b>	# Accesses on L1D C.	# Misses on L1D	# Accesses on L2 C.	# Misses on L2
<b>SPMV base</b>	44 213 215	24 455 284	24 460 819	7 941 303
ApproxTrackerL1D 2%	39 369 223	17 117 427	17 123 426	6 348 684
ApproxTrackerL1D 1%	40 686 921	17 902 245	17 965 172	6 647 113
ApproxTrackerL2 2%	41 929 299	20 380 188	20 385 879	7 318 237
ApproxTrackerL2 1%	42 968 839	21 937 632	2 1943 339	7 600 659
<b>CC base</b>	10 976 414	6 880 696	6 899 275	751 512
ApproxTrackerL1D 2%	10 643 965	6 308 565	6 329 020	747 175
ApproxTrackerL1D 1%	10 869 477	6 496 750	6 519 004	749 200
ApproxTrackerL2 2%	11 227 687	6 801 718	6 822 170	751 771
ApproxTrackerL2 1%	11 090 420	6 836 569	6 855 846	752 007
<b>2DConvolution base</b>	24 080 908	5 945 560	5 945 560	4 193 280
ApproxTrackerL1D 2%	23 912 164	5 895 392	5 931 264	4 151 884
ApproxTrackerL1D 1%	23 945 881	5 890 686	5 890 686	4 503 568
ApproxTrackerL2 2%	24 043 132	5 866 524	5 874 183	4 100 179
ApproxTrackerL2 1%	24 045 166	5 891 065	5 893 034	4 125 123
<b>GRU base</b>	20 598 784	20 087 140	20 089 172	32 300
ApproxTrackerL1D 2%	20 598 784	17 302 978	17 308 601	32 300
ApproxTrackerL1D 1%	20 397 921	17 349 232	17 360 886	32 300
ApproxTrackerL2 2%	20 598 157	20 083 251	20 084 963	32 300
ApproxTrackerL2 1%	20 597 028	20 082 102	20 084 172	32 300
<b>LSTM base</b>	3 420 928	765 196	765 196	11 309
ApproxTrackerL1D 2%	3 409 074	806 970	807 038	13 456
ApproxTrackerL1D 1%	3 416 730	806 901	806 969	13 459
ApproxTrackerL2 2%	3 420 712	768 633	768 731	13 227
ApproxTrackerL2 1%	3 420 832	766 274	766 372	12 120
<b>HybridSort</b>	20 757 084	16 321 621	46 828 781	37 702 607
ApproxTrackerL1D 2%	18 518 931	14 074 387	44 491 162	32 478 548
ApproxTrackerL1D 1%	18 960 722	14 410 148	44 990 875	33 293 247
ApproxTrackerL2 2%	18 126 851	12 960 698	43 594 103	31 387 754
ApproxTrackerL2 1%	18 403 488	13 618 581	44 261 753	32 311 079

performance improvement for SPMV and GRU applications to over 20%. ApproxTrackers solve the memory bottlenecks on both on-chip and off-chip memory portions at runtime, exploiting local performance trackers for the applications whose memory usage worsens.

We include applications that efficiently utilize GPU resources to extend the target domains and observe the performance impact of ApproxTracker approaches on these domains. As these applications generally do not face a significant memory bottleneck problem during runtime by using existing resources in a balanced way, ApproxTrackers have minimal impact on execution behavior and performance. In this manner, neither ApproxTrackerL1D nor ApproxTrackerL2 results in similar IPC and overall performance improvements for the applications such as CC and LSTM, which process relatively small datasets. As the small workloads do not bring performance-damaging memory traffic on off-chip memory partitions, ApproxTrackers have slight variations in execution performance. Consequently, ApproxTrackerL1D leads to an average improvement performance at 18.6% among the target applications by enhancing memory performance at runtime.

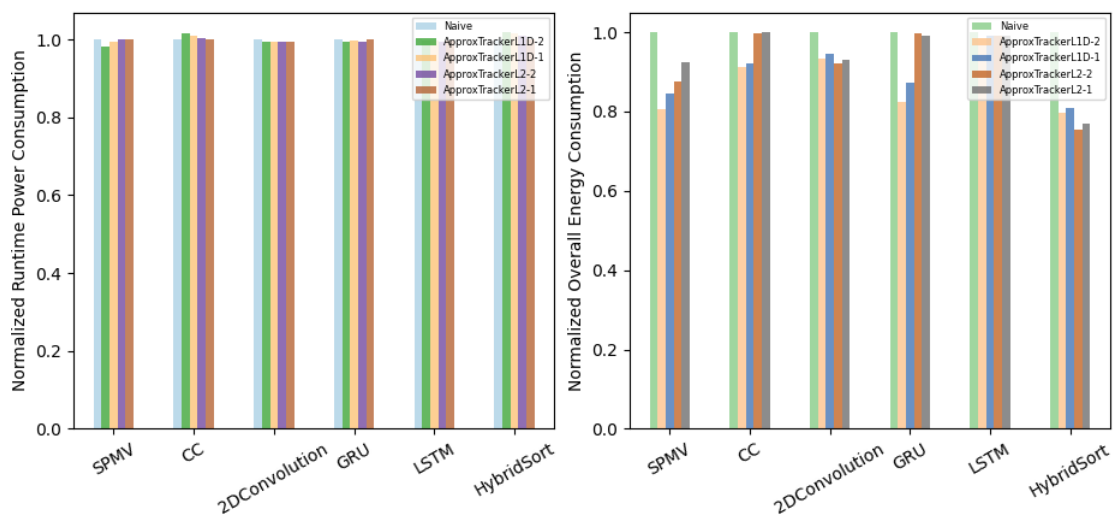


Figure 5.5. Normalized runtime power dissipation and energy consumption obtained through naive ApproxTrackerL1D and ApproxTrackerL2 versions.

Figure 5.5 displays on-average runtime power dissipation and the overall energy consumption observed through the execution of GPU kernels. Runtime power consumption of a GPU application with ApproxTrackers does not significantly change depending on CUDA application variation, and the power measurements are slightly different from the normalized naive version results. Only the applications experimented with flushing

2% of memory requests within ApproxTrackerL1D reach a slight decrease in the GPU average power consumption because that configuration increases SM utilization by solving the off-chip memory workload bottlenecks. As the power consumption measurements do not vary significantly, the total energy consumption improvements overlap with the performance. As a result, ApproxTrackerL1D, the most powerful in terms of performance enhancement among all configurations, achieves 14.8% of energy savings on target applications.

### 5.3.4. ApproxTracker Performance and Driven-Power Improvements on Various Datasets

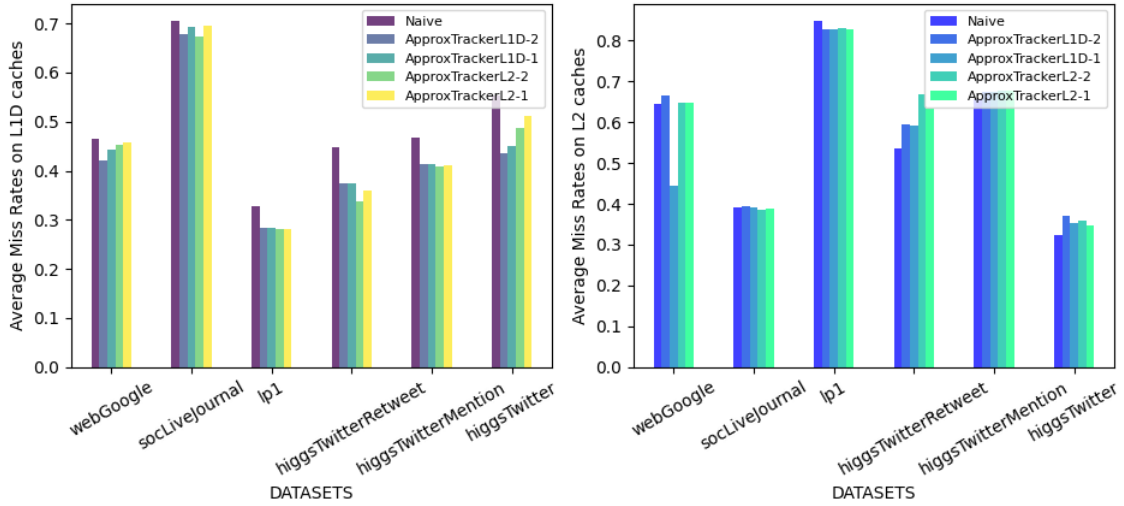


Figure 5.6. Average miss rates on L1D and L2 caches for various data types experimented with naive, ApproxTrackerL1D and ApproxTrackerL2 versions.

Figure 5.6 reveals the miss rates on the L1D and L2 caches for the SPMV algorithm processing varying datasets through naive and ApproxTrackerL1D/L2 simulator versions. ApproxTrackerL1D, configured to remove 2% of memory requests, significantly reduces miss rates on L1D caches at runtime. For instance, the average cache miss rate among 30 L1D caches decreases from 0.56 to 0.43 for the *higgsTwitter* and from 0.45 to 0.37 for the *higgsTwitterRetweet* data through the execution of SPMV with ApproxTrackerL1D, respectively. Moreover, ApproxTrackerL1D, which is allowed to flush 1% of memory requests, slightly lower the average miss rate than in experiments with 2%. Even if, ApproxTrackerL2 does not perform as promising as ApproxTrackerL1D

for reducing the average miss rates on L1D caches for *webGoogle* and *higgsTwitter* data, it can reach similar miss rate decreases for *lp1*, *higgsTwitterRetweet*, and *higgsTwitterMention* data. The fact that the sparse data size gets smaller diminishes the pressures on L1D caches, and off-chip memory partitions reach the performance limits before on-chip memory components. The reason is that off-chip memory partitions try to service multiple sparse data, which quickly creates traffic, and SMs' memory components do not face such a limitation easily, especially if corresponding SMs hold a small number of thread blocks. As ApproxTrackerL1D flushes more memory requests in total without issuing them to the off-chip region, it reduces misses rates on L1D caches more than ApproxTrackerL2 for comparably smaller graphs like *webGoogle* and *higgsTwitter*. Even so, each ApproxTracker increases data locality utilization on L1D caches.

In contrast, ApproxTrackers cannot diminish the miss rates on L2 caches apart from the *lp1* data and the exceptional case observed with *webGoogle* and ApproxTrackerL1D. In Figure 5.7, we display the total memory accesses on the sub-memory partitions to evaluate how ApproxTrackers change the number of accesses with corresponding misses.

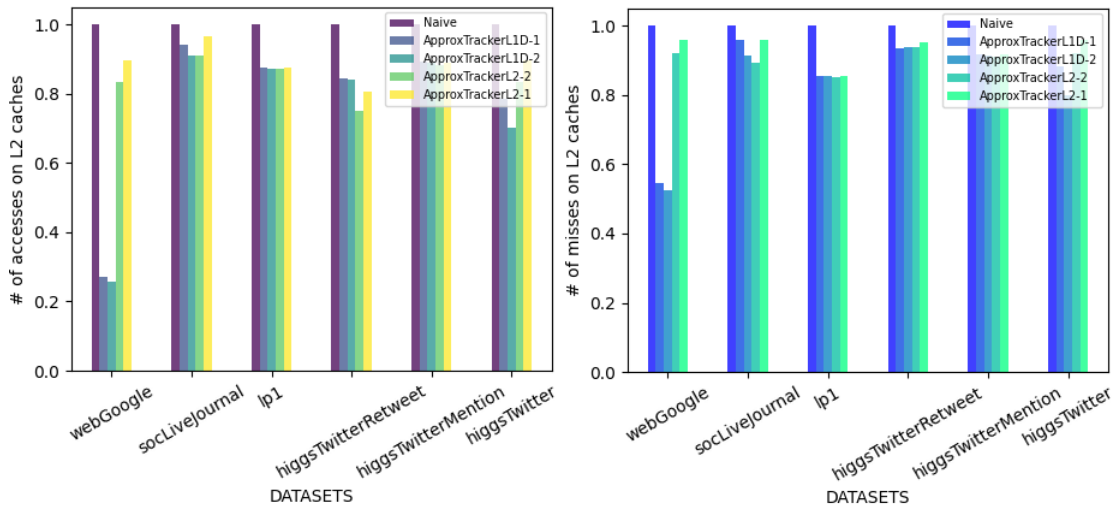


Figure 5.7. The number of L2 cache accesses and misses obtained through naive, ApproxTrackerL1D, and ApproxTrackerL2 versions among various data types.

While the left sub-figure shows the total L2 cache accesses normalized to the baseline results, the right represents the change among the misses for all accesses on L2 caches. Even if ApproxTrackers seems to either raise the miss rates on L2 caches *higgsT-*

*twitter* and *higgsTwitterRetweet* or does not change those rates significantly for remaining data, both the number of accesses and missed ones on L2 caches reduce for all datasets. As a result, both ApproxTrackerL1D and ApproxTrackerL2 reduce the number of memory accesses on L2 caches by more than 15% compared to the baseline results despite the 2% and 1% flush configurations.

Lastly, *socLiveJournal* is a 1GB graph which is nearly ten times bigger than *higgsTwitter* and *webGoogle* graphs whose data size are around 70/150MB. Whereas ApproxTrackerL1D drastically decreases off-chip memory traffic for *webGoogle* and *higgsTwitter* graphs, it does not mitigate processing the memory workload with *socLiveJournal*. Even if processing *socLiveJournal* causes high miss rates on both L1D and L2 caches, ApproxTrackers' flushing commands do not spread to the execution and fade away too earlier than the completion of the kernel.

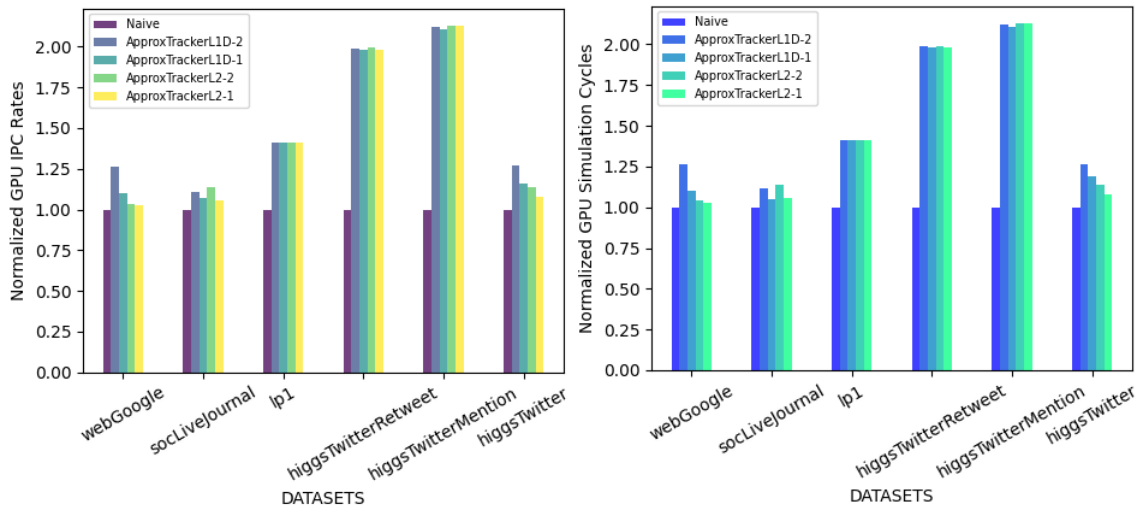


Figure 5.8. IPC and GPU simulation rates with ApproxTrackerL1D/L2 approaches for various data types.

Figure 5.8 displays the normalized IPC and GPU simulation cycles depending on the baseline execution results to evaluate the impact of ApproxTrackers on performance. According to the IPC results, increasing the utilization of L1D caches by reducing the miss rates directly speeds up the SMs due to completing memory requests faster, regardless of the data types being processed. While ApproxTrackerL1D can lead the IPC improvements on *webGoogle* and *higgsTwitter* data by increasing utilization on L1D caches more, ApproxTrackerL2 has the same impact on the execution performance with ApproxTrackerL1D for the remaining datasets. We should note that even if one expects a

relational increase in IPCs depending on the number of flushed requests, such a change in performance may not occur for all cases due to either harming future locality with flushed requests or application behavior. There are no significant differences in IPC results obtained with ApproxTrackers either 2% or 1% flush configurations results in similar IPC rates because skipping 1% of memory workload suffices to fix memory bottlenecks at runtime, and raising the number of flushes does not further improve performance.

The overall GPU execution performances generally increase in experiments, as shown in Figure 5.8. Interpreting the overall performance requires more comprehensive reasoning, different from the rising IPC rates. Flushing memory requests based on tracking traffic on off-chip memory partitions has a more accelerating effect on performance than tracking on on-chip caches. For example, lowering the off-chip memory workload has doubled the performance according to the performance results with *higgsTwitterRetweet* and *higgsTwitterMention* data. We deduce that regional memory traffics within the memory partitions dramatically starves SMs due to the long latency memory operations, and ApproxTrackers can resolve them at runtime. Since ApproxTrackers reach the number of flushable accesses quickly during execution for big and sparse data, they do not touch most of the execution of SPMV with *socLiveJournal*. As a result, ApproxTrackerL1D with 2% and 1% flush configurations accelerates experiments on the SPMV algorithm and various data by an average of 1.52× and 1.47× times, while ApproxTrackerL2 2% and 1% 1.47× and 1.44×, respectively. ApproxTrackers with 1% flush configurations can efficiently identify the memory requests to be issued to off-chip memory at critical bottleneck moments, and doubling the flushable requests does not drastically further improves the performance.

Figure 5.9 represents results for the average runtime power consumption (on the left) and overall energy consumption (on the right) observed through the experiments with naive and ApproxTrackers simulators. As seen in the left figure, neither ApproxTrackerL1D nor ApproxTrackerL2 methods have a considerable impact on the runtime power, apart from the slight decreases observed in some datasets such as *higgsTwitterRetweet*. Implementing ApproxTrackers does not directly manipulate the main contributors to the runtime power of GPUs, which are the functional units and register files on the SMs. In this manner, ApproxTrackers affects the overall energy proportional to the performance variations without considering the negligible overhead of ApproxTrackers. ApproxTrackerL1D and ApproxTrackerL2, which speed up the GPU performance, lower the overall energy consumption by 30%, 26.8%, 26.8%, and 24.1% through the experiments with 2% and 1% flush configurations, respectively.

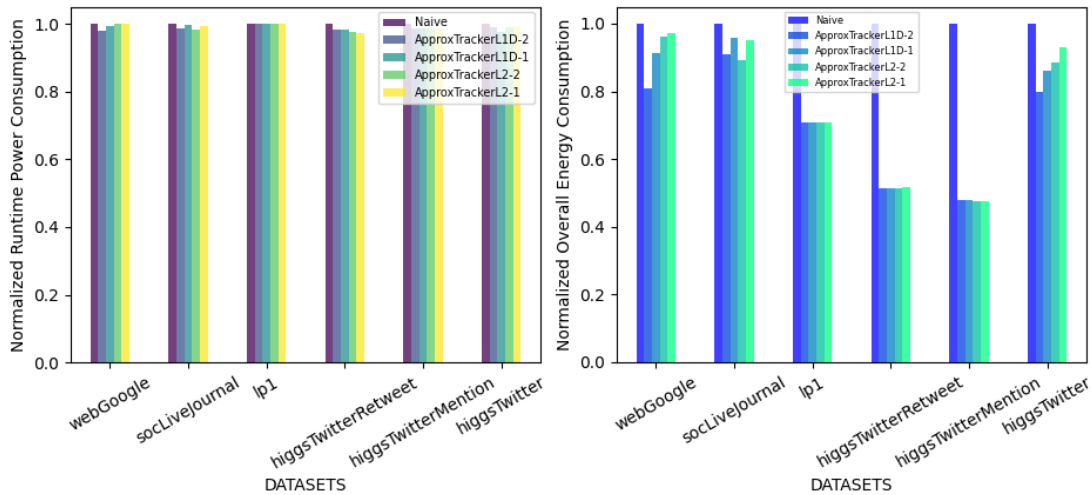


Figure 5.9. Average driven-power and energy consumption with ApproxTrackerL1D/L2 for various data types.

## 5.4. Summary

To conclude, ApproxTrackers aim to detect the memory bottlenecks on GPU hardware during runtime by receiving local feedback from each memory unit and selectively discarding memory requests to improve performance and reduce energy consumption. ApproxTrackerL1D tracks the utilization of each L1D cache within the SMs through additional tracker hardware and flushes memory requests that could cause a bottleneck in the memory hierarchy. Similarly, ApproxTrackerL2 tracks the memory exploitation on off-chip sub-memory partitions individually and deletes the memory requests from the memory pipeline before transmitting them from SMs to off-chip partitions at runtime. While ApproxTrackerL1D targets solving on-chip memory utilization problems directly, ApproxTrackerL2 manages regional workloads within the memory partitions. Furthermore, Each ApproxTracker writes zero to the operands of the flushed memory requests.

ApproxTrackers' performance improvements vary depending on the application's runtime memory behavior. While the proposed approaches enhance the execution performance of memory-bounded applications significantly, they can slightly affect the performance of the applications that utilize the hardware in a balanced manner. The average performance advancement and energy consumption reduction among all experiments with ApproxTrackers are 18.6% and 14.8%, respectively. To investigate how memory utilization affects the overall execution performance and ApproxTrackers changes it, we test ApproxTrackers with the SPMV algorithm where SPMV processes sparse data causing low memory exploitation and various datasets whose graph sizes and number of vertices

and edges change. ApproxTrackerL1D, configured with 2% and 1% flush settings, speeds up the SPMV algorithm processing various datasets by an average of 1.52× times and 1.47×, respectively. Likewise, ApproxTrackerL2 accelerates the same algorithm 1.47× and 1.44× with the same flush configurations. Additionally, ApproxTrackerL1D/L2 with 2% and 1% flush assignments reduce overall energy consumption by 30%, 26.8%, 26.8%, and 24.1% for the same experimental configurations.



## CHAPTER 6

### CONCLUSION

In this thesis, we present a detailed investigation of the impacts of micro-architectural metrics on error resiliency which we handle separately as soft error vulnerability and approximate computing, and performance in the GPU domain. We first propose a comprehensive runtime GPU monitoring tool, *GPPRMon*, proposing a systematic runtime metric collection for performance and power consumption and visualizing execution statistics in multi-perspective by considering the literature’s deficiency for such a tool. We conduct a performance bottleneck analysis by utilizing *GPPRMon* to evaluate execution at warp instruction granularity at runtime and help explicit observations of the execution. We believe that *GPPRMon* will allow conducting baseline analysis for the literature concerning GPU performance and power dissipation and eliminate the need for additional in-house efforts for real-time monitoring and profiling support. Many contributions from or on top of *GPPRMon* might emerge as future work. For illustrate, Micro-architectural metrics collected from other environments can be displayed by exploiting the *GPPRMon*’s visualizer. Furthermore, metric collection and visualization frameworks may be expanded to monitor the execution on GPUs in detail.

With *GPPRMon*, we acquire the knowledge for interpreting the interaction between applications and hardware, which enables evaluating error vulnerability for the error resiliency domain. In soft error vulnerability prediction work, we estimate the occurrence rates of SDCs, crashes, and masked faults employing a comprehensive set of ML frameworks trained with micro-architectural metrics. We exploit classification and regression methods to predict SDC and crash, and masked fault rates, respectively, because SDCs and crashes occur less frequently than masked faults. We exploit the simulator and profiler to collect metrics deployed for training ML models. The experiments utilizing simulator features, which are selected among metrics according to relevance with faults, yield more accurate predictions for SDCs and crashes. Conversely, profiler features result in better results for estimating masked faults. Furthermore, an intriguing point for future research is to apply the same approach using different GPU devices. In addition to providing insights about the error resiliency of various GPUs, a comprehensive analysis, including the execution of contemporary tasks such as DNN or graph applications, brings a generic baseline related to GPU reliability. Moreover, it would be interesting to explore

the reconfiguration of fault injections by introducing faults into diverse hardware regions under different physical conditions, such as extreme temperature or radiation conditions, instead of relying solely on modeling. This physical configuration would bring a more comprehensive fault tolerance analysis, ranging from macro-scale to micro-scale, and facilitate prediction analysis by re-evaluating the profiling results within the corresponding architectural context.

Lastly, we evaluate the error resiliency based on the approximate computing idea, which can improve performance and diminishes energy consumption, by configurable hardware-based two approximator mechanisms, ApproxTrackerL1D/L2. While ApproxTrackerL1D tracks each L1D cache on SMs' memory performances, ApproxTrackerL2 follows the data locality exploitation of L2 caches on off-chip sub-memory partitions at runtime, and each selectively flushes memory requests from the pipeline to resolve performance bottlenecks. ApproxTrackers enhance the performance of algorithms from various domains by 18.6% and reduce overall energy consumption by 14.8%. Among the experiments for a memory-bounded application with various datasets, ApproxTrackers improves the execution performance by 1.49 $\times$  and lowers the energy dissipation by 28.4%. We found that irregular access behavior causes significant performance degradation by causing long latency operations, and handling off-chip memory workload without causing bottleneck during execution significantly affects performance and energy consumption. Moreover, alternative approximation approaches by incorporating feedback from both L2 and L1D caches or considering additional other runtime tracks such as SM occupancy to improve execution efficiency may be possible feature works. Additionally, investigating the impact of varying threshold values to decide to flush a memory request and analyzing their effects on performance would be another future study. These may reveal optimal flush thresholds and approximation methods applicable generically to all GPU workloads.

## BIBLIOGRAPHY

- Aamodt, T. M., W. W. L. Fung, and T. G. Rogers (2018). General-purpose graphics processor architectures. *Synthesis Lectures on Computer Architecture* 13(2), 1–140.
- Abdi, H. and L. J. Williams (2010). Principal component analysis. *Wiley interdisciplinary reviews: computational statistics* 2(4), 433–459.
- Aktılav, B. and I. Öz (2022). Performance and accuracy predictions of approximation methods for shortest-path algorithms on gpus. *Parallel Computing* 112, 102942.
- Anghel, L., M. Benabdenbi, A. Bosio, M. Traiola, and E. I. Vatajelu (2018). Test and reliability in approximate computing. *Journal of Electronic Testing* 34(4), 375–387.
- Appleyard, J. and S. Yokim (2017, October). Programming tensor cores in cuda 9.
- Ariel, A., W. W. L. Fung, A. E. Turner, and T. M. Aamodt (2010). Visualizing complex dynamics in many-core accelerator architectures. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pp. 164–174.
- Borkar, S. (2005). Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *IEEE Micro* 25(6), 10–16.
- Candel, F., S. Petit, J. Sahuquillo, and J. Duato (2015). Accurately modeling the gpu memory subsystem. In *2015 International Conference on High Performance Computing & Simulation (HPCS)*, pp. 179–186.
- Chatterjee, N., M. O’Connor, G. H. Loh, N. Jayasena, and R. Balasubramonia (2014). Managing dram latency divergence in irregular gpgpu applications. In *SC ’14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 128–139.
- Che, S., M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron (2009). Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 44–54.

- Chen, X., L.-W. Chang, C. I. Rodrigues, J. Lv, Z. Wang, and W.-M. Hwu (2014). Adaptive cache management for energy-efficient gpu computing. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 343–355.
- Choi, J., H.-J. Lee, and C. E. Rhee (2022). Adc-pim: Accelerating convolution on the gpu via in-memory approximate data comparison. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 12(2), 458–471.
- Dimitrov, M., M. Mantor, and H. Zhou (2009). Understanding software approaches for gpgpu reliability. In *Workshop on General Purpose Processing on Graphics Processing Units*.
- Dimitrov, M. and H. Zhou (2007). Unified architectural support for soft-error protection or software bug detection. In *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, pp. 73–82.
- Domenico, M., A. Lima, P. Mougel, and M. Musolesi (2013). The anatomy of a scientific rumor. *Scientific reports* 3(1), 1–9.
- Du, B., J. E. R. Condia, and M. S. Reorda (2019). An extended model to support detailed gpgpu reliability analysis. In *2019 14th International Conference on Design Technology of Integrated Systems In Nanoscale Era (DTIS)*, pp. 1–6.
- Esmailzadeh, H., A. Sampson, L. Ceze, and D. Burger (2012, mar). Architecture support for disciplined approximate programming. *SIGARCH Comput. Archit. News* 40(1), 301–312.
- Fang, B., K. Pattabiraman, M. Ripeanu, and S. Gurusurthi (2016). A systematic methodology for evaluating the error resilience of gpgpu applications. *IEEE Transactions on Parallel and Distributed Systems* 27(12), 3397–3411.
- Gao, J., X. Chu, X. Wu, J. Wang, and G. He (2022). Parallel dynamic sparse approximate inverse preconditioning algorithm on gpu. *IEEE Transactions on Parallel and Distributed Systems* 33(12), 4723–4737.
- Garcia, R., F. Asgarinejad, B. Khaleghi, T. Rosing, and M. Imani (2021). Trulook: A framework for configurable gpu approximation. In *2021 Design, Automation & Test*

*in Europe Conference & Exhibition (DATE)*, pp. 487–490.

- Giménez, A., T. Gamblin, I. Jusufi, A. Bhatele, M. Schulz, P.-T. Bremer, and B. Hamann (2018). Memaxes: Visualization and analytics for characterizing complex memory performance behaviors. *IEEE Transactions on Visualization and Computer Graphics* 24(7), 2180–2193.
- Goloubeva, O., M. Rebaudengo, M. Sonza Reorda, and M. Violante (2003). Soft-error detection using control flow assertions. In *Proceedings 18th IEEE Symposium on Defect and Fault Tolerance in VLSI Systems*, pp. 581–588.
- Grauer-Gray, S., L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos (2012). Auto-tuning a high-level language targeted to gpu codes. In *2012 Innovative Parallel Computing (InPar)*, pp. 1–10.
- Guerreiro, J., A. Ilic, N. Roma, and P. Tomás (2019). Dvfs-aware application classification to improve gpgpus energy efficiency. *Parallel Computing* 83, 93–117.
- Guo, L., D. Li, and I. Laguna (2021). Paris: Predicting application resilience using machine learning. *Journal of Parallel and Distributed Computing* 152, 111–124.
- Hari, S., T. Tsai, M. Stephenson, S. Keckler, and J. Emer (2017, 04). Sassifi: An architecture-level fault injection tool for gpu application resilience evaluation. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 249–258.
- Hong, J., S. Cho, and G. Kim (2022). Overcoming memory capacity wall of gpus with heterogeneous memory stack. *IEEE Computer Architecture Letters* 21(2), 61–64.
- Hoshino, T., A. Ida, T. Hanawa, and K. Nakajima (2018). Load-balancing-aware parallel algorithms of h-matrices with adaptive cross approximation for gpus. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 35–45.
- Imani, M., D. Peroni, and T. Rosing (2018). Nvalt: Nonvolatile approximate lookup table for gpu acceleration. *IEEE Embedded Systems Letters* 10(1), 14–17.
- Imani, M., A. Sokolova, R. Garcia, A. Huang, F. Wu, B. Aksanli, and T. Rosing (2019).

- Approxlp: Approximate multiplication with linearization and iterative error control. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6.
- Islam, T., A. Ayala, Q. Jensen, and K. Ibrahim (2019). Toward a programmable analysis and visualization framework for interactive performance analytics. In *2019 IEEE/ACM International Workshop on Programming and Performance Visualization Tools (ProTools)*, pp. 70–77.
- Jain, P., A. Jain, A. Nrusimha, A. Gholami, P. Abbeel, J. Gonzalez, K. Keutzer, and I. Stoica (2019). Checkmate: Breaking the memory wall with optimal tensor rematerialization. *CoRR abs/1910.02653*, 497–511.
- Jauk, D., D. Yang, and M. Schulz (2019). Predicting faults in high performance computing systems: An in-depth survey of the state-of-the-practice. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '19*.
- Jiao, Y., H. Lin, P. Balaji, and W. Feng (2010). Power and performance characterization of computational kernels on the gpu. In *2010 IEEE/ACM Intel Conference on Green Computing and Communications & Intel Conference on Cyber, Physical and Social Computing*, pp. 221–228.
- Jog, A., O. Kayiran, N. C. Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. R. Iyer, and C. R. Das (2013). OWL: cooperative thread array aware scheduling techniques for improving GPGPU performance. In V. Sarkar and R. Bodík (Eds.), *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013*, pp. 395–406. ACM.
- Kalra, C., F. Previlon, X. Li, N. Rubin, and D. Kaeli (2018). Prism: Predicting resilience of gpu applications using statistical methods. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC '18*.
- Kandiah, V., S. Peverelle, M. Khairy, J. Pan, A. Manjunath, T. G. Rogers, T. M. Aamodt, and N. Hardavellas (2021). Accelwattch: A power modeling framework for modern gpus. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microar-*

chitecture, MICRO '21, New York, NY, USA, pp. 738–753. Association for Computing Machinery.

Karki, A., C. Palangotu Keshava, S. Mysore Shivakumar, J. Skow, G. Madhukeshwar Hegde, and H. Jeon (2019). Tango: A deep neural network benchmark suite for various accelerators. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 137–138.

Khairy, M., Z. Shen, T. M. Aamodt, and T. G. Rogers (2020). Accel-sim: An extensible simulation framework for validated gpu modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 473–486.

Koo, G., Y. Oh, W. W. Ro, and M. Annavaram (2017). Access pattern-aware cache management for improving data utilization in gpu. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pp. 307–319.

Kosmidis, L., I. Rodriguez Ferrandez, A. Jover-Alvarez, S. Alcaide, J. Lachaize, J. Abella, O. Notebaert, F. Cazorla, and D. Steenari (2020, 06). Gpu4s: Embedded gpus in space - latest project updates. *Microprocessors and Microsystems* 77, 103143.

Krzywaniak, A., P. Czarnul, and J. Proficz (2022). Gpu power capping for energy-performance trade-offs in training of deep convolutional neural networks for image recognition. In *Computational Science – ICCS 2022*, Cham, pp. 667–681. Springer International Publishing.

Laguna, I., M. Schulz, D. F. Richards, J. Calhoun, and L. Olson (2016). Ipas: Intelligent protection against silent output corruption in scientific applications. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*.

Leskovec, J., K. J. Lang, A. Dasgupta, and M. W. Mahoney (2009a). Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. *Internet Mathematics* 6(1), 29 – 123.

Leskovec, J., K. J. Lang, A. Dasgupta, and M. W. Mahoney (2009b). Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics* 6(1), 29–123.

- Leveugle, R., A. Calvez, P. Maistri, and P. Vanhauwaert (2009). Statistical fault injection: Quantified error and confidence. In *2009 Design, Automation & Test in Europe Conference & Exhibition, Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*.
- Lew, J., D. A. Shah, S. Pati, S. Cattell, M. Zhang, A. Sandhupatla, C. Ng, N. Goli, M. D. Sinclair, T. G. Rogers, and T. M. Aamodt (2019). Analyzing machine learning workloads using a detailed gpu simulator. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 151–152.
- Li, M.-L., P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou (2008, mar). Understanding the propagation of hard errors to software and implications for resilient system design. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XIII*, New York, NY, USA, pp. 265–276. Association for Computing Machinery.
- Li, S., J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi (2009). Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 469–480.
- Lu, Q., K. Pattabiraman, M. S. Gupta, and J. A. Rivers (2014). Sdctune: A model for predicting the sdc proneness of an application for configurable protection. In *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*.
- Mahmoud, A., S. K. S. Hari, M. B. Sullivan, T. Tsai, and S. W. Keckler (2018). Optimizing software-directed instruction replication for gpu error detection. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC '18*.
- Maier, D., B. Cosenza, and B. Juurlink (2018). Local memory-aware kernel perforation. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization, CGO 2018*, New York, NY, USA, pp. 278–287. Association for Computing Machinery.



- Maier, D., N. Mammeri, B. Cosenza, and B. Juurlink (2019). Approximating memory-bound applications on mobile gpus. In *2019 International Conference on High Performance Computing & Simulation (HPCS)*, pp. 329–335.
- Mittal, S. (2016, mar). A survey of techniques for approximate computing. *ACM Comput. Surv.* 48(4), 1–33.
- Mittal, S. and J. S. Vetter (2016). A survey of techniques for modeling and improving reliability of computing systems. *IEEE Transactions on Parallel and Distributed Systems* 27(4), 1226–1238.
- Mukherjee, S. (2008). *Architecture Design for Soft Errors*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.
- Nie, B., J. Xue, S. Gupta, T. Patel, C. Engelmann, E. Smirni, and D. Tiwari (2018). Machine learning models for gpu error prediction in a large scale hpc system. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 95–106.
- NVIDIA (2015). Ptx and sass assembly debugging.
- NVIDIA (2018a). Data sheet: Nvidia quadro p4000.
- NVIDIA (2018b). Data sheet: Quadro gv100.
- NVIDIA (2018c, March). Volta architecture white paper.
- NVIDIA (2019, June). Jetson agx xavier and the new era of autonomous machines.
- NVIDIA (2022a, Aug). Kernel profiling guide.
- NVIDIA (2022b). Nvidia, cuda-gdb.
- NVIDIA (2022c). Nvidia quadro p4000.
- NVIDIA (2022d, May). Profiler user’s guide.

- NVIDIA (2023a). Cuda toolkit documentation.
- NVIDIA (2023b). Nvidia cuda compiler driver nvcc.
- NVIDIA (2023c). Nvidia management library (nvml).
- NVIDIA (2023d). Parallel thread execution isa version 8.0.
- Oliveira, D., F. B. Moreira, P. Rech, and P. Navaux (2018). Predicting the reliability behavior of hpc applications. In *International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*.
- O’Neil, M. A. and M. Burtscher (2014). Microarchitectural performance characterization of irregular gpu kernels. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 130–139.
- Öz, I. and Ö. F. Karadaş (2022). Regional soft error vulnerability and error propagation analysis for gpgpu applications. *The Journal of Supercomputing* 78(3), 4095–4130.
- Pattnaik, A., X. Tang, O. Kayiran, A. Jog, A. Mishra, M. T. Kandemir, A. Sivasubramanian, and C. R. Das (2019). Opportunistic computing in gpu architectures. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pp. 210–223.
- Pedregosa, F., G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research* 12, 2825–2830.
- Pentecost, L., U. Gupta, E. Ngan, J. Beyer, G.-Y. Wei, D. Brooks, and M. Behrisch (2019). Champvis: Comparative hierarchical analysis of microarchitectural performance. In *2019 IEEE/ACM International Workshop on Programming and Performance Visualization Tools (ProTools)*, pp. 55–61.
- Peroni, D., M. Imani, H. Nejatollahi, N. Dutt, and T. Rosing (2020). Data reuse for accelerated approximate warps. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39(12), 4623–4634.

- Power, J., J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood (2015). gem5-gpu: A heterogeneous cpu-gpu simulator. *IEEE Computer Architecture Letters* 14(1), 34–36.
- Rahimi, A., L. Benini, and R. K. Gupta (2013). Spatial memoization: Concurrent instruction reuse to correct timing errors in simd architectures. *IEEE Transactions on Circuits and Systems II: Express Briefs* 60(12), 847–851.
- Rahimi, A., L. Benini, and R. K. Gupta (2016). Circa-gpus: Increasing instruction reuse through inexact computing in gp-gpus. *IEEE Design & Test* 33(6), 85–92.
- Rhu, M., M. Sullivan, J. Leng, and M. Erez (2013). A locality-aware memory hierarchy for energy-efficient gpu architectures. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, New York, NY, USA, pp. 86–98. Association for Computing Machinery.
- Rogers, T. G., M. O’Connor, and T. M. Aamodt (2013). Divergence-aware warp scheduling. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, New York, NY, USA, pp. 99–110. Association for Computing Machinery.
- Rusch, M. and E. Hart (2022, October). Improve shader performance and in-game frame rates with shader execution reordering.
- Sabena, D., L. Sterpone, L. Carro, and P. Rech (2014). Reliability evaluation of embedded gpgpus for safety critical applications. *IEEE Transactions on Nuclear Science* 61(6), 3123–3129.
- Sampson, A., J. Nelson, K. Strauss, and L. Ceze (2014, sep). Approximate storage in solid-state memories. *ACM Trans. Comput. Syst.* 32(3), 1–23.
- Shende, S. S. and A. D. Malony (2006, may). The tau parallel performance system. *Int. J. High Perform. Comput. Appl.* 20(2), 287–311.
- Sidiroglou-Douskos, S., S. Misailovic, H. Hoffmann, and M. Rinard (2011). Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE ’11, New York, NY, USA, pp. 124–134. Association for

Computing Machinery.

Singh, S. and R. Nasre (2018). Scalable and performant graph processing on gpus using approximate computing. *IEEE Transactions on Multi-Scale Computing Systems* 4(3), 190–203.

Singh, S. and R. Nasre (2019). Optimizing graph processing on gpus using approximate computing: Poster. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming, PPOPP '19*, New York, NY, USA, pp. 395–396. Association for Computing Machinery.

Singh, S. and R. Nasre (2020). Graffix: Efficient graph processing with a tinge of gpu-specific approximations. In *Proceedings of the 49th International Conference on Parallel Processing, ICPP '20*, New York, NY, USA. Association for Computing Machinery.

Stine, D., C. Musterle, and A. Rink (2021, September). Nvidia dlss and enscape: Introducing the latest technology in real-time visualization.

Sun, Y., S. Mukherjee, T. Baruah, S. Dong, J. Gutierrez, P. Mohan, and D. Kaeli (2018). Evaluating performance tradeoffs on the radeon open compute platform. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 209–218.

Sun, Y., Y. Zhang, A. Mosallaei, M. D. Shah, C. Dunne, and D. R. Kaeli (2021). Daisen: A framework for visualizing detailed GPU execution. *Comput. Graph. Forum* 40(3), 239–250.

TechPowerUp (2020). Nvidia geforce rtx 2060.

TOP500 (2023). List Statistics | TOP500 — top500.org.

Topçu, B. and I. Öz (2022). Predicting the soft error vulnerability of gpgpu applications. In *2022 30th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pp. 108–115.

Ubal, R., B. Jang, P. Mistry, D. Schaa, and D. Kaeli (2012). Multi2sim: A simulation

- framework for cpu-gpu computing. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 335–344.
- Venkatagiri, R., A. Mahmoud, S. K. S. Hari, and S. V. Adve (2016). Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–14.
- Vijaykumar, N., E. Ebrahimi, K. Hsieh, P. B. Gibbons, and O. Mutlu (2018). The locality descriptor: A holistic cross-layer abstraction to express data locality in gpus. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 829–842.
- Wang, Z., S. H. Nelaturu, and S. Amarasinghe (2019). Accelerated cnn training through gradient approximation. In *2019 2nd Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2)*, pp. 31–35.
- Wei, X., R. Zhang, Y. Liu, H. Yue, and J. Tan (2019). Evaluating the soft error resilience of instructions for gpu applications. In *2019 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC)*, pp. 459–464.
- Wong, D., N. S. Kim, and M. Annavaram (2016). Approximating warps with intra-warp operand value similarity. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 176–187.
- Xu, Z., X. Chen, J. Shen, Y. Zhang, C. Chen, and C. Yang (2019, jan). Gardenia: A graph processing benchmark suite for next-generation accelerators. *ACM Journal on Emerging Technologies in Computing Systems* 15(1), 1280–1293.
- Zhao, W., S. Tan, and P. Li (2020). Song: Approximate nearest neighbor search on gpu. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pp. 1033–1044.
- Zhao, X., A. Adileh, Z. Yu, Z. Wang, A. Jaleel, and L. Eeckhout (2019). Adaptive memory-side last-level gpu caching. In *2019 ACM/IEEE 46th Annual International*

*Symposium on Computer Architecture (ISCA)*, pp. 411–423.

Öz, I. and S. Arslan (2021). Predicting the soft error vulnerability of parallel applications using machine learning. *Int. J. Parallel Program.* 49(3), 410–439.