

Summer 2021

Low-Overhead Techniques For Secure And Reliable Gpu Computing

Gurunath Kadam

William & Mary - Arts & Sciences, gurunath.kadam@gmail.com

Follow this and additional works at: <https://scholarworks.wm.edu/etd>



Part of the [Computer Sciences Commons](#)

Recommended Citation

Kadam, Gurunath, "Low-Overhead Techniques For Secure And Reliable Gpu Computing" (2021).
Dissertations, Theses, and Masters Projects. Paper 1627047873.
<http://dx.doi.org/10.21220/s2-jvrg-8s06>

This Dissertation is brought to you for free and open access by the Theses, Dissertations, & Master Projects at W&M ScholarWorks. It has been accepted for inclusion in Dissertations, Theses, and Masters Projects by an authorized administrator of W&M ScholarWorks. For more information, please contact scholarworks@wm.edu.

Low-Overhead Techniques for Secure and Reliable GPU Computing

Gurunath Kadam

Mumbai, India

Bachelor of Engineering, University of Mumbai, 2006
Master of Science, Technical University of Darmstadt, 2012

A Dissertation presented to the Graduate Faculty of
The College of William & Mary in Candidacy for the Degree of
Doctor of Philosophy

Department of Computer Science

College of William & Mary
May 2021

APPROVAL PAGE

This Dissertation is submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy



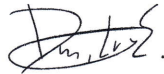
Gurunath Kadam

Approved by the Committee, May 2021

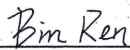


Committee Chair

Adwait Jog, Assistant Professor, Computer Science
College of William & Mary



Dmitry Evt'yushkin, Assistant Professor, Computer Science
College of William & Mary



Bin Ren, Assistant Professor, Computer Science
College of William & Mary



Evgenia Smirni, Professor, Computer Science
College of William & Mary



Ashutosh Pattnaik, Senior GPU Architect
ARM Ltd.

ABSTRACT

In recent years, Graphics Processing Units (GPUs) have become a de facto choice to accelerate the computations in various domains such as machine learning, security, financial and scientific computing. GPUs leverage the inherent data parallelism in the target applications to provide high throughput at superior energy efficiency. Due to the rising usage of GPUs for a large number of applications, they are facing new challenges, especially in the security and reliability domains. From the security side, recently several microarchitectural attacks targeting GPUs have been demonstrated. These attacks leak the secret information stored on GPUs, for example, the parameters of a neural network (NN) model and the private user information. From the reliability side, the innovations to improve GPU memory systems are making them more susceptible to errors. My dissertation research focuses on addressing these security and reliability challenges in GPUs while minimizing the associated overhead of the proposed protection mechanisms.

To improve GPU security, we focus on the previously demonstrated correlation timing attack. Such an attack exploits the deterministic nature of the coalescing mechanism in GPUs to correlate the execution time and the number of accesses. Consequently, an attacker can recover the encryption keys stored on GPUs. Therefore, to counter the correlation timing attack, we first introduce a randomized coalescing defense scheme (RCoal). RCoal randomizes the coalescing logic such that the attacker fails to correlate the execution time and the number of accesses. As a result, RCoal thwarts the correlation timing attack. Next, we propose a bucketing based coalescing defense scheme, BCoal, which minimizes the variation in the number of memory accesses by generating a predetermined number (called buckets) of memory accesses. With low variation in the number of memory accesses, the attacker cannot correlate the application execution time and the secret information, thus failing the correlation timing attack. BCoal generates less memory traffic than RCoal and, therefore, is performance efficient.

To improve GPU reliability, we address the data memory faults in GPU caches and DRAM. Existing reliability mechanisms of redundancy and check-pointing fail to scale with the increasing memory/computational demands on GPUs and quickly become impractical. To address this problem, we study a wide range of applications to find that a very small fraction of the data memory is most vulnerable to faults. This small fraction of the data is not only highly accessed but also highly shared across GPU threads. Consequently, we propose and develop two reliability schemes to detect-only and to detect/correct faults in this most vulnerable data while incurring low overhead. The focus of on-going and future work is to improve the reliability of machine learning applications.

TABLE OF CONTENTS

Acknowledgments	vi
Dedication	vii
List of Tables	viii
List of Figures	ix
1 Introduction	2
1.1 Low-Overhead Hardware Techniques for Improving GPU Security . .	3
1.1.1 Problem Statement	3
1.1.2 Contributions	4
1.2 Low-Overhead Hardware Techniques for Reliable GPU Computing . .	5
1.2.1 Problem Statement	5
1.2.2 Contributions	5
1.3 Dissertation Organization	7
2 A General Background on Graphics Processing Units (GPUs)	8
2.1 Baseline GPU Architecture	8
2.2 GPU Memory Access Optimization Techniques	9
2.3 Data Memory Faults in GPUs	11
2.3.1 Data Memory Faults and Their Impact in GPUs	11
2.3.2 Importance of Data Memory Faults in GPUs	12

3	RCoal: Mitigating GPU Timing Attack via Subwarp-based Randomized	
	Coalescing Techniques	13
3.1	Introduction	13
3.2	Background	16
3.2.1	Baseline GPU Architecture	16
3.2.2	AES Encryption	18
3.2.3	Baseline Timing Attack	20
3.3	Motivation and Goals	22
3.4	Subwarp based Defense Mechanisms	25
3.4.1	Fixed-sized Subwarps (FSS)	25
3.4.2	Random-sized Subwarp (RSS)	28
3.4.3	Random-threaded Subwarp (RTS)	29
3.4.4	Implementation Details	30
3.4.5	Corresponding Attacks	31
3.5	Theoretical Security Analysis	32
3.5.1	Analytical Model	32
3.5.2	Analysis of Defense Mechanisms	33
3.5.2.1	FSS	35
3.5.2.2	FSS+RTS	35
3.5.2.3	RSS+RTS	36
3.5.3	Results	36
3.6	Experimental Analysis of Security and Performance	37
3.6.1	Effect on Security	38
3.6.2	Effect on Performance and Data Movement	42
3.6.3	Evaluating the Trade-off Between Security and Performance	43
3.6.4	Case Study: Plaintext with 1024 Lines	43
3.7	Discussion and Future Work	45

3.8	Related Work	45
3.9	Conclusions	46
4	BCoal: Bucketing-based Memory Coalescing for Efficient and Secure GPUs	48
4.1	Introduction	48
4.2	Background	51
4.2.1	Basics of GPU Architecture	51
4.2.2	Bandwidth Conserving Mechanisms	52
4.2.3	AES Encryption	54
4.2.4	Baseline Attack and Defense Mechanism	55
4.3	Motivation and Analysis	58
4.3.1	Performance Overhead Analysis of RCoal	58
4.3.2	Effect of MSHRs and Caches on Security with RCoal	60
4.3.3	Our Proposal and Goals	62
4.4	Anatomy of Bucketing in GPUs	62
4.4.1	Bucket Features	63
4.4.2	Estimation of Number of Padded Accesses	64
4.4.3	Design Challenges in Generating Padded Accesses	64
4.4.4	BCoal: A Secure Bucketing Scheme	66
4.5	Hardware/Software Overhead	67
4.6	Analysis of Security & Performance	68
4.6.1	Experimental Analysis of Security	69
4.6.2	Theoretical Analysis of Security	71
4.6.3	Experimental Analysis of Performance	74
4.6.4	Evaluating BCoal on Other Applications	74
4.7	Related Work	77
4.8	Conclusions	78

5 Data-centric Reliability Management in GPUs	80
5.1 Introduction	80
5.2 Background	82
5.2.1 Baseline GPU Architecture	82
5.2.2 Data Memory Faults in GPUs	84
5.2.3 Fault Injection Setup	84
5.2.3.1 Fault Model	84
5.2.3.2 Error Metric Selection	85
5.3 Motivation and Workload Analysis	86
5.3.1 Problem Definition and Goals of This Work	86
5.3.2 Application Access Pattern Analysis	89
5.3.3 Impact of Faults in Data Memory	91
5.4 Data-centric Reliability Management: Analysis, Design, and Imple- mentation	93
5.4.1 Application Source Code Analysis	93
5.4.2 Detection and Correction Resilience Schemes	98
5.4.2.1 Multi-bit Fault Detection	98
5.4.2.2 Multi-bit Fault Detection-and-Correction	99
5.4.3 Implementation Overhead	100
5.5 Experimental Results	101
5.5.1 Performance Evaluation	101
5.5.1.1 Detection-Only	101
5.5.1.2 Detection-and-Correction	103
5.5.2 Reliability Evaluation	103
5.5.3 Reliability and Performance Tradeoff	106
5.6 Related Works	106
5.7 Conclusions	108

6	Exploration of the Reliability of ML Models	110
6.1	Advantages of Weight Quantization in ML Models	110
6.2	Impact of Memory Faults on the Quantized Models	111
6.2.1	Fault Model	111
6.2.2	Impact of Memory Faults	111
7	Conclusion and Future Research Directions	115
7.1	Summary of Dissertation Contributions	115
7.2	Future Research Directions	117
	Bibliography	118
	Vita	138

ACKNOWLEDGMENTS

The following thesis and research are the culmination of guidance, encouragement, and moral support from many people throughout my graduate studies. First, I sincerely thank my advisor, Adwait Jog, for his thoughtful, patient, and diligent mentoring. Adwait taught me not only how to conduct successful research, but also how to overcome failures. He ensured that as a graduate student I maintained a work-life balance. I hope to carry these lessons with me in future work.

During my PhD, I was fortunate to collaborate with Danfeng Zhang and Evgenia Smirni. Danfeng and Evgenia have broadened my knowledge in their respective domains. I thank them for their guidance, support, and time.

I extend my gratitude to my dissertation committee members, Bin Ren, Dmitry Evtvyushkin, Evgenia Smirni, and Ashutosh Pattnaik. Their generous support and attentive feedback improved this dissertation.

I thank Carlos Rozas for giving me an opportunity to intern at Intel Labs. Patrick Koeberl and Andrea Miele introduced me to new challenges in FPGA security and made my time at Intel a great learning experience.

I thank the College of William & Mary for supporting my graduate studies through a fellowship and for allowing me to conduct research through the College computing facilities. I thank Eric Walter, Laura Hild, and Jay Kanukurthy for their assistance in using the computing facilities. I also thank the Computer Science administrative staff – Vanessa Godwin, Jacquelyn Johson, and Dale Hayes – for their help to speedily resolve any administrative issues.

My time at William & Mary has been truly memorable thanks to my lab-mates Haonan Wang, Mohamed Ibrahim, and Hongyuan Liu. Haonan, Mohamed, and Hongyuan offered invaluable help and companionship throughout my Ph.D. life.

Finally, I would like to thank my wife, Mary Elizabeth, who firmly supported me through my graduate course work, submission deadlines, conference travels, and research. I thank all my family members for their care and encouragement without which I would not be what I am today.

To my parents.

LIST OF TABLES

3.1	Key configuration parameters of the simulated GPU configuration. ©	
	2018 IEEE.	16
3.2	Security analysis results with $N = 32$ and $R = 16$, where N is the	
	number of threads and R is the number of memory blocks. Here, M	
	is the number of subwarps and S is the number of samples normalized	
	to FSS with $M = 1$ case. © 2018 IEEE.	36
4.1	Key configuration parameters of the simulated GPU. © 2020 IEEE.	52
4.2	Security Analysis. S denotes the normalized number of samples re-	
	quired to successfully recover an AES key byte [49]. © 2020 IEEE.	
		73
5.1	Key configuration parameters of the simulated GPU. © 2021 IEEE.	83
5.2	Output Error Metrics for Applications. © 2021 IEEE.	86
5.3	Input data objects to the GPU applications. Data objects are sorted	
	based on the number of accesses incurred (Highest to Lowest). The	
	emboldened data objects are classified as hot data objects (highly	
	accessed and shared). © 2021 IEEE.	97
6.1	Comparison between 32-bit floating-point and weight quantized	
	(INT8) ML models.	111

LIST OF FIGURES

2.1 Overview of GPU Architecture.	9
2.2 Memory access coalescing in GPUs. (© 2018 IEEE).	10
2.3 L2 Cache size trend for Nvidia and AMD GPUs.	12
3.1 Overview of Baseline GPU Architecture. (© 2018 IEEE).	17
3.2 Effect of subwarps on memory coalescing. (© 2018 IEEE).	19
3.3 Last round execution of AES-128 algorithm. The t_i in $T4[t_i]$ represents the index of the table lookup operation. k_j and c_j represent the j th byte of the last round key and ciphertext, respectively. tid is the thread id within a warp. (© 2018 IEEE).	20
3.4 Overview of the process of guessing one of the correct last round key byte (k_j). $A_j^{m,n}$ is the number of memory requests for m^{th} guess of the j^{th} last round key byte using n^{th} plaintext. n varies between 1 to N , where N is the number of plaintext samples. m varies from 0 to 255 and j varies from 1 to 16. (© 2018 IEEE).	21
3.5 Relationship between Last Round and Total Execution Time. (© 2018 IEEE).	22
3.6 Effect of Coalescing on the Recovery of 0^{th} Last Round Key Byte (k_0): a) Recovery is Successful when Coalescing is Enabled, b) Recovery is Unsuccessful when Coalescing is Disabled. (© 2018 IEEE).	23

3.7 Performance of FSS enabled AES with respect to number of subwarps:	
a) Execution Time and Total Memory Accesses per plaintext with increasing number of subwarps, b) Average of correlations between the last round execution time and the last round memory accesses for all key bytes. The last round memory accesses are calculated assuming correct values of a key byte and the number of subwarps for coalescing to be one. © 2018 IEEE.	26
3.8 Fixed Size Subwarp (FSS) mechanism against FSS attack. © 2018 IEEE.	28
3.9 Subwarp size distribution of RSS for $num_subwarp = 4$. © 2018 IEEE.	29
3.10 Effects of different defense mechanisms on coalescing for $num_subwarp = 2$: a) FSS+RTS and b) RSS+RTS. <i>sid</i> represents subwarp id and <i>tid</i> represents thread id. © 2018 IEEE.	30
3.11 Modified Coalescing Unit to realize FSS, RSS, and RTS defense mechanisms. The additional hardware required is the field to store subwarp-id (<i>sid</i>) for each thread. © 2018 IEEE.	32
3.12 FSS+RTS defense mechanism against FSS+RTS attack. © 2018 IEEE.	39
3.13 RSS defense mechanism against RSS attack.	40
3.14 RSS+RTS defense mechanism against RSS+RTS attack. © 2018 IEEE.	40
3.15 Comparison between the security offered by FSS, FSS+RTS, RSS and RSS+RTS based on the average of correlations between the last round coalesced accesses for all key bytes and the last round execution time observed during the encryption. The last round coalesced accesses are calculated using the <u>corresponding attacks</u> . © 2018 IEEE.	41
3.16 Performance and Data Movement Comparisons between FSS, FSS+RTS, RSS, and RSS+RTS. © 2018 IEEE.	42

3.17 Comparison between the FSS, FSS+RTS, RSS and RSS+RTS defense mechanisms based on the RCoal score against the corresponding attacks: a) Security-oriented system with $a = 1$ and $b = 1$, b) Performance-oriented system with $a = 1$ and $b = 20$. © 2018 IEEE.	43
3.18 Effects of the defense mechanisms on security against the corresponding attacks and performance with respect to the number of subwarps for plaintext with 1024 lines: a) Average of correlations between the last round coalesced accesses from the attack and the execution for all key bytes, b) Execution time (normalized to the case when $num_subwarp=1$) with respect to the number of subwarps. © 2018 IEEE.	44
4.1 Overview of Baseline GPU Architecture. © 2020 IEEE.	52
4.2 Memory access coalescing in GPUs. © 2020 IEEE.	53
4.3 Baseline Attack. © 2020 IEEE.	56
4.4 Effect of different RCoal coalescing schemes on the recovery of one of the last round key byte (shown in red circle). In RCoal, the caches and MSHRs are disabled for security reasons (refer Section 4.3-B). © 2020 IEEE.	57
4.5 Illustrating the overhead of RCoal defense scheme for different sizes of plaintext. The results are normalized to a baseline GPU with MSHRs and caches. © 2020 IEEE.	58
4.6 Histogram of the number of coalesced accesses generated across a warp for 1000 plaintext samples each with 32 lines. © 2020 IEEE.	59
4.7 Effect of MSHRs on the cache-missed coalesced accesses in RCoal scheme. © 2020 IEEE.	61

4.8	The presence of MSHRs and caches leads to successful recovery of one of the last round key bytes in RCoal(32) and RCoal(4). Plaintext has 32 lines. (c) 2020 IEEE.	61
4.9	Evaluation of security offered by the bucketing scheme employing unique access padding mechanisms with one bucket of size 16. Plaintext has 32 lines. (c) 2020 IEEE.	65
4.10	BCoal defense scheme against correlation attack for plaintext with 32 lines. (c) 2020 IEEE.	69
4.11	BCoal defense scheme against correlation attack for plaintext with 64 lines. (c) 2020 IEEE.	70
4.12	Performance of BCoal for different plaintext sizes. All results are normalized to the baseline GPU. (c) 2020 IEEE.	75
4.13	Performance Evaluation of BCoal on GPGPU applications with MSHRs/caches enabled. Results are normalized to the baseline GPU. (c) 2020 IEEE.	76
5.1	A Schematic of the Baseline GPU Architecture. (c) 2021 IEEE.	83
5.2	L2 Cache size trends for NVIDIA and AMD GPUs. (c) 2021 IEEE.	87
5.3	Normalized number of accesses to data memory blocks. For (a)-(f), we note that very few memory blocks experience a very high number of RD accesses compared to other blocks. (c) 2021 IEEE.	88
5.4	Percentage of active warps accessing the data memory blocks. (c) 2021 IEEE.	90
5.5	Fault injection methodology to evaluate application vulnerability of hot memory blocks to the memory faults. The data memory blocks are sorted based on the total number of accesses to each block. (c) 2021 IEEE.	91

5.6	Effect of faults in the hot (highly accessed/shared) memory blocks versus the rest of the memory blocks on the application output. © 2021 IEEE.	92
5.7	Performance overhead of Detection-only (dark bar) and Detection-and-Correction (white bar) resilience schemes. All numbers are normalized to the baseline case (no reliability protection, 1.0). The hot data objects reside in hot memory blocks. © 2021 IEEE.	102
5.8	Fault injection for evaluating fault detection-and-correction: the probability of a memory block selection depends on the number of its L1-missed accesses (since the proposed schemes address faults in L2-caches and DRAM). © 2021 IEEE.	104
5.9	Silent data corruption due to faults in L2-cache and DRAM: The x-axis represents the number of protected data objects cumulatively increasing, starting from the baseline case (no data objects are protected). The y-axis shows the number of SDC outputs out of 1000 runs for each error injection configuration. The detection-only/detection-and-correction schemes stop the multi-bit data memory errors caused by the faults from propagating to the output. © 2021 IEEE.	105
6.1	Impact of faults in a weight element on the classification accuracy in FP32 and quantized (INT8) AlexNet model.	112
6.2	Contribution of faults in weights of different convolution layers of float 32 (FP32) and quantized (INT8) ML models on the classification accuracy.	113

Low-Overhead Techniques for Secure and Reliable GPU Computing

Chapter 1

Introduction

Graphics Processing Units (GPUs) provide significant performance and energy efficiency advantages over CPUs as the former exploits the data-level parallelism in the applications [60, 5, 69, 118, 59, 61, 106, 3, 2, 12, 54, 122]. As a result, GPUs are largely deployed to accelerate applications in various fields, such as high-performance computing (HPC), artificial intelligence (AI), finance, virtual/augmented reality, genomics, and autonomous vehicle workloads [23, 109, 122, 93, 116, 98, 91, 96, 103, 94].

With the increasing demand for GPUs in the computational and graphical workloads, the challenges faced by the GPU architecture are also increasing, especially in the security and reliability domains. For example, some of the GPU-run applications, such as DNA and financial computing, process private user data. Furthermore, while the deep learning workloads benefit from the computational power of GPUs, the neural network (NN) models are under attack to steal the confidential model parameters and the private user data processed on the GPUs [37, 81]. Also, cryptographic applications, such as AES encryption handling sensitive data, are known to achieve significant performance benefits

IEEE Copyright Note: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of the College of William and Mary's products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink. If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

from GPUs [44, 128, 72, 17, 83, 42]. As a result, the secure computations on GPUs are a primary focus to ensure that the confidential and private data processed is not leaked by an attacker.

Next, emerging computing needs have fueled the growth of GPU architectures. Specifically, the growing data sizes in the deep learning and AI workloads have lead to a significant increase in the GPU storage structures [99]. Consequently, several research efforts have been made to operate these storage structures efficiently. However, the effect of these innovations on GPU reliability is not yet well-understood. For example, low voltage cache designs (i.e., AMD Killi [27] or IBM Dante [11]) for managing power consumption of large last-level caches in GPUs can increase the likelihood of multi-bit faults. Furthermore, advanced DRAM architectures make multi-bit faults more common [120, 125]. The increasing multi-bit faults in the GPU memory structures may adversely affect the output of the applications leading to less reliable GPU computations. The reduced reliable GPU computations, in turn, may lead to catastrophic failures, such as accidents of autonomous vehicles [105, 16, 65, 43]. Consequently, the data memory faults in the GPU memory structures must be addressed to ensure reliable GPU computations.

1.1 Low-Overhead Hardware Techniques for Improving GPU Security

1.1.1 Problem Statement

As noted above, GPUs outperform CPUs in terms of performance and power efficiency when executing cryptographic applications, such as AES encryption [42, 72, 17, 83]. However, several new correlation timing and covert channel attacks have been demonstrated on GPUs to leak confidential and private user data [44, 80, 135, 45]. Our research focuses on the recent correlation timing attack on GPU, which exploited the correlation between the execution time and the number of coalesced accesses to memory to recover the AES

encryption keys [44]. Specifically, the attacker exploits the relationship between the coalesced accesses and AES encryption keys to reveal the encryption key through an offline correlation analysis with the help of the recorded encryption execution time and encrypted (cipher-) text.

1.1.2 Contributions

We introduce two hardware-based defense mechanisms for GPUs to mitigate the correlation timing attack and prevent the leakage of security-sensitive data. The first defense mechanism, RCoal, randomizes the memory access coalescing mechanism to generate additional accesses to thwart the correlation timing attack [49]. For the second defense mechanism, we propose a bucketing-based coalescing mechanism, BCoal [50]. BCoal generates additional memory accesses whenever necessary to match the total number of accesses to a set of pre-determined numbers (called buckets). Consequently, BCoal reduces the correlation between the number of accesses and private data to mitigate the correlation timing attack. Furthermore, since BCoal generates fewer memory accesses compared to RCoal, BCoal proves to be a performance efficient defense mechanism.

Our research on secure GPU computing through hardware techniques makes the following contribution:

- We analyze the correlation timing attack on GPUs to show that the regularity and deterministic nature of the memory access coalescing is a major security vulnerability.
- We propose two novel hardware-based defense mechanisms to mitigate the correlation timing attack leveraging the memory access coalescing. The first defense mechanism, RCoal, randomizes the memory coalescing logic to eliminate the relationship between the number of memory accesses and the execution time to thwart the correlation timing attack.
- Our second defense mechanism, BCoal, implements a bucketing-based coalescing mechanism to always issue pre-determined numbers (chosen from a small set, called buckets) of coalesced accesses by padding additional accesses to the real accesses whenever necessary. As a result, the correlation between the number of accesses and the execution

time reduces thus mitigating the correlation timing attack.

- We demonstrate through the theoretical and empirical analysis that both of our defense mechanisms offer improved security against the correlation timing attack at a low performance overhead. Furthermore, both defense mechanisms offer a tradeoff between the security and performance that can be set by a user.

- We show that both, RCoal and BCoal, defense mechanisms incur a very low hardware overhead.

1.2 Low-Overhead Hardware Techniques for Reliable GPU Computing

1.2.1 Problem Statement

The current generation of GPUs employs an error checking and correction (ECC) mechanism called single error correction and double error detection (SECCDED) to detect and correct the data memory faults [58]. SECCDED-ECC corrects single-bit fault and detects up to two-bit faults. However, as noted earlier, with the growing error rate in the GPU memory structures, the number of multi-bit (more than 2-bits) faults is increasing [120, 125, 27, 11]. SECCDED-ECC is not capable of addressing these rising multi-bit faults and, therefore, the GPU output reliability cannot be guaranteed. A stronger ECC, such as Chipkill, can address the multi-bit faults. However, Chipkill is not feasible to implement on GPUs [58]. As a result, a performance efficient reliability mechanism to address the rising multi-bit faults is needed.

1.2.2 Contributions

To devise a performance efficient reliability scheme, we adopt a data-centric approach [48]. We begin with the GPU application profiling to study how the data memory is accessed by an application during runtime. We noted that a small fraction of the data memory is

highly accessed and shared by the GPU threads as compared to the rest of the memory. We call this highly accessed and shared memory as *hot* memory. Next, we simulate the fault injections to study their impact on this small fraction of hot memory on GPU application output. We note that the faults in the hot memory spread widely across GPU warp thread instructions leading to catastrophic failures, for example, misclassifications in the neural networks. Consequently, we conclude that the hot memory should be prioritized to be covered under the reliability mechanism. Based on our observations, we develop a detection-only and detection-and-correction reliability schemes to reduce GPU output corruption.

Our research work on performance efficient reliable GPU computing makes the following contribution:

- We perform a detailed application-level analysis to show that a small fraction of data used by a large number of GPGPU application threads can dramatically increase their vulnerability to multi-bit faults. This data is usually read-only and can be profiled with low-overhead.
- We develop both detection and correction mechanisms that prioritize the reliability of this identified critical data. Our mechanisms leverage the critical data information obtained from the software for the partial data replication and execute checks only for this small fraction of data.
- Our selective detection and correction mechanisms based on partial data replication exhibit very limited overhead due to the fact that overhead of additional checks (and associated memory accesses) can be hidden thanks to latency tolerance property of GPUs. Quantitatively, we significantly improve GPU reliability, an average 98.97% drop in the number of execution runs with corrupt output while incurring a low average performance overhead of 1.2%.

1.3 Dissertation Organization

The rest of this dissertation is organized as follows. Chapter 2 provides an overview of GPU architecture, along with the memory access optimization techniques and data memory faults and their impact. Chapter 3 introduces our first defense scheme against correlation timing attacks which employs randomized coalescing. Chapter 4 details our bucketing-based coalescing scheme which secures the GPU against correlation timing attacks. In Chapter 5, we present our data-centric reliability mechanism to address the data memory faults in GPUs. We present the initial findings of our ongoing exploration of the reliability of the machine learning (ML) models in Chapter 6. Finally, in Chapter 7, we conclude this dissertation and provide details of possible future research directions.

Chapter 2

A General Background on Graphics Processing Units (GPUs)

In this chapter, we provide a general background on Graphics Processing Units (GPUs). Specifically, we focus on the GPU memory access optimization techniques and application execution on GPUs. We also introduce the data memory faults and their impact on GPU applications.

2.1 Baseline GPU Architecture

Figure [2.1](#) shows a baseline GPU architecture. A typical GPU is comprised of a set of processing cores known as Streaming Microprocessors (SMs) in Nvidia terminology. Each SM is comprised of several processing elements (PEs), usually 32. To leverage the parallelism in applications, GPUs employ a single instruction multiple thread (SIMT) programming paradigm to launch applications on SMs. In SIMT, GPUs execute the same set of instructions with different data on multiple threads in lockstep. These threads are grouped in concurrent thread array (CTA) blocks and are launched on SMs. The SMs split each CTA block into warps (usually 32 threads/warp) and process each warp individually on the PEs. By launching multiple warps on each SM, GPUs hide memory access latency

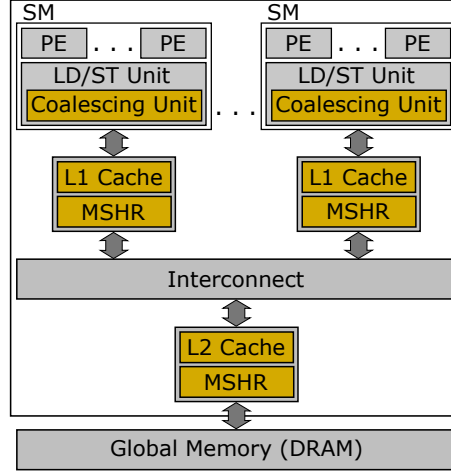


Figure 2.1: Overview of GPU Architecture.

by executing warps in a multiplexed manner to improve the efficiency of SMs.

To address the high bandwidth requirement of the GPU applications, GPUs employ high-bandwidth global memory (DRAM) along with multi-level caches. Each SM on GPUs has a private L1 cache shared across the corresponding PEs. Next, GPUs have multiple L2 cache banks shared between all SMs connected through an interconnection network. Finally, the L2 cache banks are connected to the off-chip DRAM through separate memory channels.

2.2 GPU Memory Access Optimization Techniques

Memory bandwidth is one of the most performance-critical shared resources in GPUs [47, 46]. GPUs adopt several memory bandwidth optimization techniques, such as memory access coalescing, caching and merging to reduce the number of accesses to the global memory. In this sub-section, we provide a brief overview of these optimizations.

Access Coalescing. In GPUs, threads within a warp execute the instructions in lock-step. For a global memory load instruction, all 32 threads within a warp execute 32 load instructions. The coalescing unit in the LD/ST unit merges multiple memory requests from different threads of the same warp (*intra-warp coalescing*) into as few cache line-

sized coalesced memory accesses as possible. The intra-warp coalescing happens at the sub-warp granularity, where the coalescing unit of the SM determines the coalesced accesses of the warp by examining a group of threads belonging to the same sub-warp. If the threads of a sub-warp access data within a contiguous memory block, their requests are coalesced together to reduce memory bandwidth consumption. The size and number of sub-warps are typically fixed and remain the same throughout the application execution. However, to achieve security, the coalescing mechanisms can be randomized (RCoal [49]) so that the coalesced accesses are no longer predictable to the attacker.

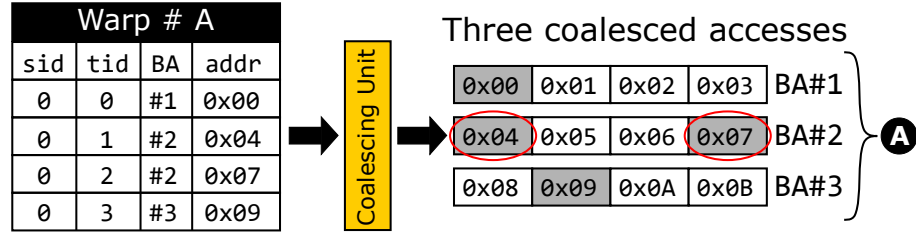


Figure 2.2: Memory access coalescing in GPUs. © 2018 IEEE.

Figure 2.2 illustrates the coalescing in baseline GPU and previously proposed randomized coalescing techniques. Assume a single warp with four threads. The per-thread addresses and the requested block addresses (BA) are shown with corresponding thread-ids (tid) and sub-warp id (sid). In the baseline GPU, we assume a single sub-warp (sid = 0 for all threads) and hence all threads participate together in the coalescing. Since the requests from tid 1 and 2 map to the same cache block, only three accesses are generated (A) to conserve the memory bandwidth.

Caching. GPUs further conserve the memory bandwidth by exploiting the temporal and spatial locality in memory accesses across and within warps with the help of hardware caches. Current GPUs employ two levels of caches, L1-cache (shared by the warps executing on the same SM) and L2-cache (shared by the warps executing on different SMs).

Access Merging. The coalesced memory accesses from a warp are sent to the L1-cache. Upon cache misses, the memory accesses are logged in the miss-status holding

registers (MSHRs). Multiple cache-missed coalesced accesses to the same cache block from different warps on the same SM are merged (*inter-warp merging*) in MSHRs. Note that as independent loads from the same warp can be issued to improve memory-level parallelism, MSHRs also help in merging redundant accesses from the same warp (*intra-warp merging*) if they are issued at different times. Another source of inter-warp merging is via MSHRs at L2-cache, where the redundant L2-cache misses (across different SMs) can be merged together.

2.3 Data Memory Faults in GPUs

2.3.1 Data Memory Faults and Their Impact in GPUs

GPUs are susceptible to a variety of faults due to the manufacturing process variations, on-chip cross-talk between interconnects, alpha/neutron particle strikes, temperature variations, power supply noise, etc. [77, 125, 85, 86, 120, 121, 27]. In our research work, we focus on memory faults in GPUs. The memory faults may cause the bits of the stored data to flip to a different value than the original. This changed value of the data may lead to the silent data corruption (SDC) of the GPU output, where the application is successfully executed but results in erroneous output.

The effect of the data memory faults can be well understood when studied with respect to application usage. GPUs are increasingly used for the machine learning applications, such as self-driving cars, medical imaging, or computer virus detection. A data memory fault may cause a mis-classification in the respective machine learning application. The impact of such mis-classification itself depends upon the purpose of an application. In the case of a self-driving car, a mis-classification due to the data memory error may lead to a crash endangering lives and property. In the case of medical imaging, a mis-classification may lead to misdiagnosis leading to delay in critical care of a patient.

The faults in the hardware are addressed through an on-chip error checking and correction (ECC) mechanism, which mitigates the effect of faults on the application output.

The current generation of GPUs employs a single-bit error correction and double-bit error detection (SECCDED) ECC mechanism to address the data memory faults in the caches and DRAM [58]. The multi-bit (more than two-bit) faults cannot be addressed in the current generation GPUs.

2.3.2 Importance of Data Memory Faults in GPUs

Due to the increasing workload sizes, the sizes and the operating speeds of the GPU memory are increasing rapidly. Figure 2.3 shows that the L2 cache sizes are increasing rapidly over the generations of GPUs from the major vendors. Consequently, significant research efforts have been made to operate the L2 cache at low voltage to achieve power efficiency [28, 27]. However, when operated at a low voltage, the fault rate of the L2 caches increases as well [28].

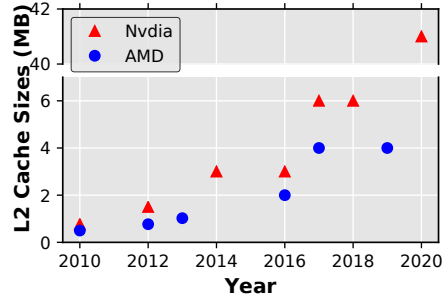


Figure 2.3: L2 Cache size trend for Nvidia and AMD GPUs.

Next, several field studies have demonstrated that the DRAMs in GPUs are more susceptible to multi-bit faults compared to the DRAMs in CPUs as the former uses a weaker ECC mechanism, SECCDED, while the latter uses a stronger ECC mechanism, Chipkill [120, 121, 77]. However, the implementation of Chipkill [19] on GPUs is currently not feasible [58].

Chapter 3

RCoal: Mitigating GPU Timing Attack via Subwarp-based Randomized Coalescing Techniques

3.1 Introduction

Graphics Processing Units (GPUs) are becoming an inevitable part of every computing system because of their ability to provide fast and energy-efficient computation. Given such ability, GPUs are also now being used to accelerate a variety of cryptographic algorithms. For example, the popular Advanced Encryption Standard (AES) algorithm [78] is known to achieve significant speedups on GPUs compared to CPUs [34, 41, 89, 66] as the AES algorithm exposes abundant thread-level parallelism to leverage high bandwidth and compute throughput of GPUs. With such increasing popularity of GPUs to accelerate security-sensitive applications, it is imperative to keep GPUs secure against a variety of side-channel attacks and other security vulnerabilities.

In this paper, we specifically focus on the correlation-based timing attacks on GPUs.

In general, a correlation-based timing attack exploits the relationship between the secret data and its impact on the processing time of an application: the attacker sends a large number of data samples to calculate the correlation between the actual processing time and the secret data. Among the *guessed* values for the secret data, the one leading to the highest correlation is the actual secret data. Notably, the recent work from Jiang et al. [44] demonstrated a correlation-based timing attack on a remote GPU server. They exploited two observations. First, the number of coalesced memory accesses in the last round can be deterministically calculated based on the last round private key byte and the encrypted text. Second, the number of coalesced accesses in the last round is correlated with the total execution time. With these two observations, an attacker can recover each key byte by picking the value that best correlates with the recorded total execution time from the remote GPU server¹.

The goal of this paper is to design low-overhead defense mechanisms to thwart timing attacks that exploit the memory coalescing in GPUs. To this end, a straightforward solution is to eliminate the correlation between the number of coalesced accesses and the total execution time by disabling the memory access coalescing mechanism completely. However, since the memory access coalescing is one of the key features in GPUs that optimizes the memory bandwidth consumption, the disabling of coalescing will incur a heavy performance penalty due to increase in the number of memory accesses [61, 55, 44, 112]. To provide a better trade-off between security and performance, we propose *RCoal*, a series of three tunable coalescing mechanisms to guard against correlation-based timing attacks.

The first mechanism focuses on tuning the granularity at which threads are coalesced together, thereby increasing the number of coalesced accesses at a finer granularity. We call this technique as fixed-sized subwarp (FSS) defense mechanism, where the size of subwarp determines the coalescing granularity. FSS mechanism helps to reduce the correlation between the coalesced accesses and total execution time by reducing the variance in the

¹Section 3.2 presents more details on the attack.

coalesced accesses. Building on the first mechanism, the second mechanism focuses on randomly changing the size of each subwarp. We call this technique as random-sized subwarp (RSS) defense mechanism where the size of each subwarp affects the attacker’s ability to correctly determine the number of coalesced accesses. The final mechanism focuses on randomly changing the thread elements of each subwarp. We call this technique as random-threaded subwarp (RTS) defense mechanism as the coalescer picks random thread elements to form a subwarp. RTS can be applied to both FSS and RSS to further hinder the attacker’s ability to determine the number of coalesced accesses correctly.

To the best of our knowledge, this is the first work to thwart timing attacks in GPUs via randomized coalescing techniques. In summary, this paper makes the following contributions:

- We generalize the correlation-based timing attack on GPUs and show that the regularity and determinism in memory access coalescing is a major security vulnerability.
- We propose three novel coalescing mechanisms to mitigate the timing attacks arising from memory access coalescing. These mechanisms revolve around carefully changing the size, number, and thread elements of a subwarp to reduce the correlation between the number of coalesced accesses and the total execution time.
- We present a detailed information-theoretical analysis to show that our randomized coalescing mechanisms can improve the GPU security by 24 to 961 times. Our extensive simulation results confirm the theoretical results and demonstrate that the improved security can be achieved at a performance loss of 5 to 28%.
- We propose a new metric called *RCoal_Score* that provides an opportunity for hardware engineers to tune the security and performance trade-off as per their requirements. We discuss two such security-performance trade-off designs and conclude that RSS and RTS mechanisms provide significant advantages towards performance and security, respectively.

3.2 Background

In this section, we briefly introduce a) the baseline GPU architecture and the process of memory access coalescing, b) the anatomy of AES encryption, and c) the baseline timing attack assumed in this paper.

3.2.1 Baseline GPU Architecture

Overview. Figure 3.1 shows a high-level schematic of the GPU architecture. A typical GPU consists of multiple cores, called as streaming multiprocessors (SMs) in NVIDIA terminology. Each SM takes advantage of the Single Instruction, Multiple Threads (SIMT) programming paradigm [54] to schedule multiple threads on its processing elements (PEs). These threads are scheduled at the granularity of a *warp*, which is essentially a collection of (usually 32) individual threads that execute a single instruction on the PEs in a lock step manner. Each SM can execute multiple warps concurrently in a multiplexed manner to hide the long global memory latencies and improve the utilization of core resources (e.g., register file, scratchpad memory). All SMs are connected to global memory partitions via an on-chip interconnect. In this paper, we evaluate the proposed techniques on a GPU architecture simulated using a cycle accurate GPU simulator – GPGPU-Sim [6]. More details on the simulated architecture are given in Table 3.1

Table 3.1: Key configuration parameters of the simulated GPU configuration. © 2018 IEEE.

Core Features	1400MHz core clock, SIMT width = 32 (16×2)
Resources / Core	32KB shared memory, 32KB register file, 15 SMs 32 threads/warp, one subwarp per coalescing unit
Features	immediate post dominator based branch divergence handling
Memory Model	6 GDDR5 Memory Controllers (MCs), FR-FCFS scheduling 16 DRAM-banks, 4 bank-groups/MC, 924 MHz memory clock Global linear address space is interleaved among partitions in chunks of 256 bytes Hynix GDDR5 Timing [39], $t_{CL} = 12$, $t_{RP} = 12$, $t_{RC} = 40$, $t_{RAS} = 28$, $t_{CCD} = 2$, $t_{RCD} = 12$, $t_{RRD} = 6$
Interconnect	1 crossbar/direction, 1400MHz interconnect clock, islip VC and switch allocators

Memory Access Coalescing. One of the effective ways to improve the collective perfor-

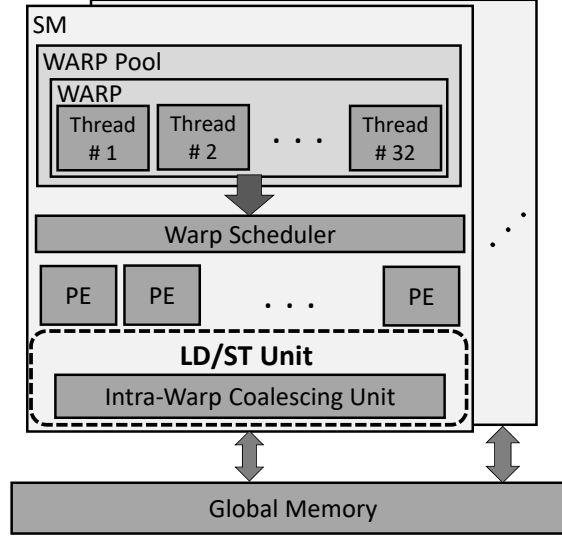


Figure 3.1: Overview of Baseline GPU Architecture. © 2018 IEEE.

mance of the concurrently executing threads on GPUs is to optimize the global memory bandwidth. To this end, several techniques such as intra-warp memory access coalescing, inter- and intra-warp request merging via miss status handling registers (MSHRs), sectoring [112], and L1/L2 caching have been proposed for GPUs. In this work, we focus on intra-warp memory access coalescing technique, which merges multiple memory requests from different threads of the same warp in to as few cache line sized coalesced memory accesses as possible.

The coalescing unit (part of LD/ST unit of the SM) performs the agglomeration of memory requests from the threads in a warp at a *subwarp* level, where the number of subwarps is an architectural parameter. If the threads of a particular subwarp request nearby data within a contiguous block of the memory, their requests are coalesced together to avoid redundant accesses. Therefore, if the memory access size, subwarp size, and thread-data pattern (e.g., if/when thread to table index mapping is known) are known, the number of memory accesses can be calculated accurately. As per CUDA programming guide [95], the scalar threads from the same warp can be coalesced together (subwarp size of 1), at a half-warp basis (subwarp size of 2) or at a quarter-warp basis (subwarp size of

4). The subwarp size is decided based on the size of the memory request from each thread. The generated coalesced accesses are serviced at the rate that matches with the underlying cache/memory bandwidth. To correctly simulate the number of coalesced accesses as that of in the baseline attack model (explained later in the section), we assume subwarp size to be 1 in our baseline architecture.

To understand the effect of subwarps on coalescing, consider an example with warp comprising of four threads under two different cases employing the number of subwarps (*num-subwarp*) as 1 and 2, respectively, as shown in Figure 3.2. We assume that four threads generate four accesses and if perfectly coalesced will generate one coalesced access (memory block). When all the threads are considered together for coalescing (i.e., Case 1: *num-subwarp* is 1), only three coalesced accesses are generated as the requests from the second and third thread are coalesced into one request. When *num-subwarp* is 2 (Case 2), the coalescing is performed independently for each subwarp. Consequently, two coalesced accesses per subwarp (in total four) are generated.

3.2.2 AES Encryption

Basics. The Advanced Encryption Standard (AES) [78] is a widely used symmetric-key algorithm. The AES standard specifies 128, 192, and 256 bits as the standard key lengths. Without losing generality, we focus on AES-128, which employs a 128-bit key to encrypt the plaintext. AES-128 algorithm consists of 10 rounds each with its own round key of 16 bytes, which is generated from the encryption key. In each round, `subBytes()` transformation (details of other transformation can be found in prior works on AES [34, 41, 89, 66]) performs a table look-up operation on the substitution (S-box) table. In the last round, a table look-up operation is performed on the T_4 S-box table followed by bitwise XOR operation with the last round key. This operation is expressed by Equation 3.1 for the j^{th} byte of output ciphertext (c_j) and i^{th} input state of the last round (t_i , table lookup index) [34, 44]. $T_4 []$ represents the last round S-box table look-up operation whose result

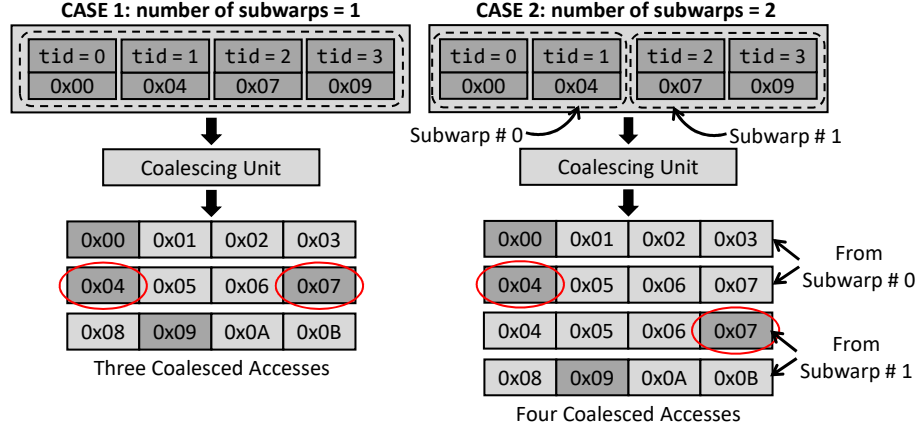


Figure 3.2: Effect of subwarps on memory coalescing. © 2018 IEEE.

is XORed with j^{th} byte of the last round key (k_j).

$$c_j = T_4[t_i] \oplus k_j \quad (3.1)$$

GPU Implementation of AES Encryption. A CUDA implementation of AES divides the plaintext across multiple parallel threads to improve GPU throughput. Each thread performs encryption on one line (block) of the plaintext. Therefore, each warp consists of 32 threads performing 32 different encryptions. The line to thread mapping is sequential and deterministic in the baseline implementation. If the size of the plaintext exceeds 32 lines, then it is divided sequentially among several warps. For example, a plaintext with 1024 lines will employ 32 warps each executing 32 lines of the plaintext. Figure 3.3 shows the encryption process for the last round on 32 threads of a single warp. Each thread performs encryption of a byte (p_j) of the input text, where j varies from 1 to 16. All threads of the warp work in a lock-step manner and perform the same table look up operation ($T_4[t_i]$) with different values of t_i . The accesses are coalesced together by the coalescing unit, and when the replies come back, all threads use the same last round key (k_j) to generate one column of the ciphertext c_j as per Equation 3.2. In Equation 3.2, tid is the thread index.

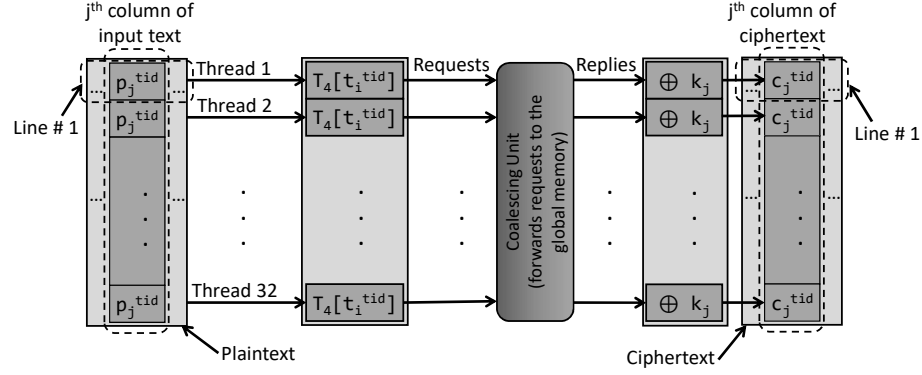


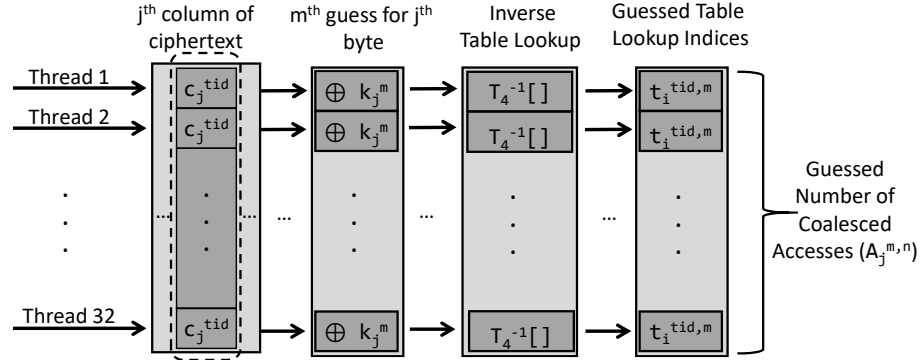
Figure 3.3: Last round execution of AES-128 algorithm. The t_i in $T_4[t_i]$ represents the index of the table lookup operation. k_j and c_j represent the j th byte of the last round key and ciphertext, respectively. tid is the thread id within a warp. © 2018 IEEE.

$$c_j^{tid} = T_4 \left[t_i^{tid} \right] \oplus k_j \quad (3.2)$$

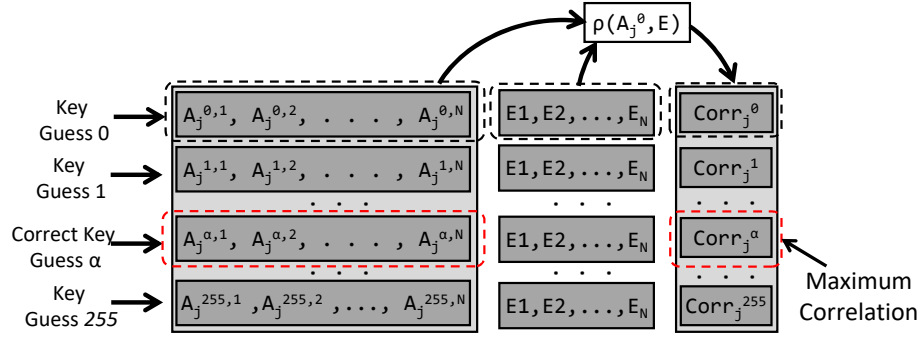
3.2.3 Baseline Timing Attack

In this paper, we use the correlation timing attack proposed by Jiang et al. [44] as the baseline attack. The attack model assumes that the attacker sends a large number of plaintexts to a remote GPU AES encryption server. The attacker collects the ciphertexts and records the total execution time for each plaintext. The goal is to correctly find all 16 last round key bytes by exploiting a key observation that there is a high correlation between the number of memory accesses and the total execution time on GPU. The baseline attack targets the last round key since it is the most vulnerable round and key expansion is invertible (i.e., it is possible to derive the original private key from any round key) [82]. The observation is that each table lookup index in the last round can be computed from a byte of the last round key (k_j) and the corresponding byte of ciphertext (c_j), independent of other ciphertext bytes (as shown in Equation 3.3). Thus, the attacker is able to observe the security leakage separately at per-byte level.

$$t_i = T_4^{-1} [c_j \oplus k_j] \quad (3.3)$$



(a) Guessing the Coalesced Accesses from the Table Look up Indices



(b) Guessing the Last Round Key

Figure 3.4: Overview of the process of guessing one of the correct last round key byte (k_j). $A_j^{m,n}$ is the number of memory requests for m^{th} guess of the j^{th} last round key byte using n^{th} plaintext. n varies between 1 to N , where N is the number of plaintext samples. m varies from 0 to 255 and j varies from 1 to 16. © 2018 IEEE.

Figure 3.4 shows the attack process for recovering the j^{th} last round key byte (k_j). The attack process has two major steps. The first step involves a *guessed* key value k_j^m where m ranges from 0 to 255. According to Equation 3.3, the table lookup index of each thread ($t_i^{tid,m}$) can be computed, as shown in Figure 3.4a. Once the indices are obtained for all threads, the attacker can calculate the expected number of coalesced accesses ($A_j^{m,n}$) for the n^{th} plaintext with the known and deterministic behavior of coalescing (in our configuration, 16 consecutive table elements are mapped sequentially to the same memory block). This particular attack assumes *num-subwarp* to be 1 (i.e., all threads in the warp are processed together for coalescing). This first step is repeated for all possible 256 key byte guesses for the j^{th} byte and for N plaintext samples. As a result, a memory access

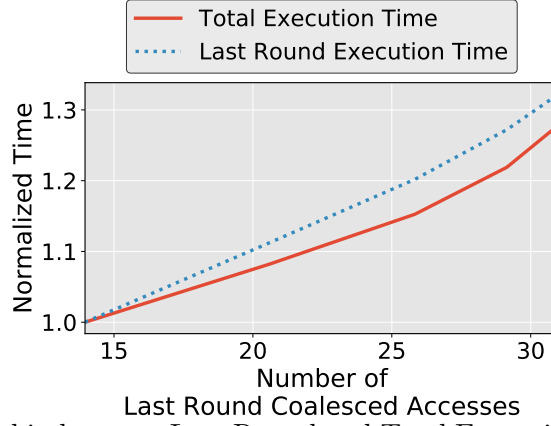


Figure 3.5: Relationship between Last Round and Total Execution Time. © 2018 IEEE.

matrix is generated as shown in Figure 3.4b. Each row of the matrix corresponds to the number of guessed memory accesses for a particular key guess (m) across N plaintext samples (A_j^m).

The second step involves calculating the correlation between each row (key guess) of the memory access matrix with the last round execution time (E) to encrypt each plaintext (collected by the attacker). Since both the total and last round execution time correlate with last round coalesced accesses (shown in Figure 3.5), the guessed key value (α) is correct for k_j if it has the maximum correlation value ($corr_j^\alpha$) with E . For the rest of the paper, we assume a stronger attack with the capability of accessing last round execution time as compared to the realistic attack, which is weaker due to the noise in the total execution time.

3.3 Motivation and Goals

The primary reason behind the success of the baseline correlation timing attack is the deterministic behavior of memory access coalescing that allows accurate calculation of the coalesced accesses generated. To verify this on our GPGPU-Sim based simulation environment, we plot the correlation values ($corr_j^m$) of all 256 possible values of m for 0^{th} key byte ($k_0, j=0$). We calculate this correlation value between the coalesced accesses from the attack and the execution time of the last round of AES-128. From Figure 3.6a,

we observe that the correlation value is the highest (highlighted in red and encircled) for the correct value of the 0^{th} key byte among all other guess values. We observe this trend for all 16 last round key bytes indicating that we can successfully guess all of them.

As a first step towards defending against the baseline attack, we aim to eliminate the relationship between the number of coalesced accesses and the last round execution time by disabling the coalescing mechanism. As a result, the number of coalesced accesses will always be 32 (i.e., the worst case scenario) from a warp with 32 threads. We executed the same baseline attack with coalescing disabled to find that there is no correlation between the number of coalesced accesses and the last round execution time. Consequently, we could not successfully guess any of the key byte. Figure 3.6b shows the plot of correlation values against the possible values of the 0^{th} key byte. The correlation of the correct key byte is very close to zero, so as that of other key guesses.

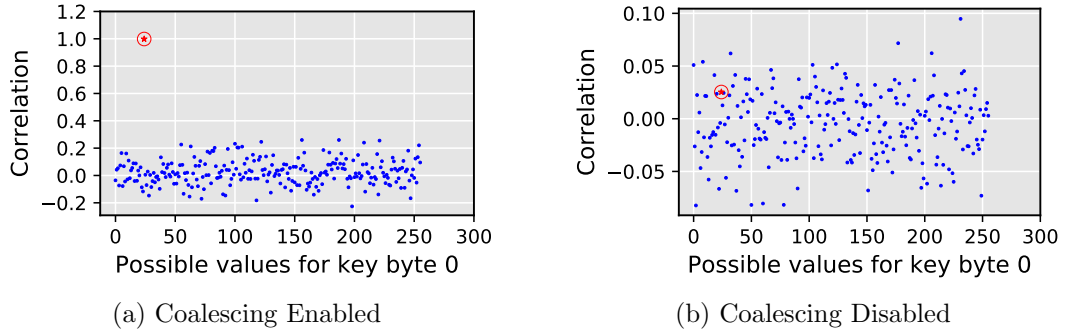


Figure 3.6: Effect of Coalescing on the Recovery of 0^{th} Last Round Key Byte (k_0): a) Recovery is Successful when Coalescing is Enabled, b) Recovery is Unsuccessful when Coalescing is Disabled. © 2018 IEEE.

Although disabling the coalescing is an effective technique to defend against the baseline correlation timing attack, absence of memory access coalescing degrades the GPU performance and energy efficiency significantly [61, 55, 44, 112]. Our own experiments show that the performance degrades by up to 178% for AES-128 algorithm encrypting plaintext of 1024 lines. Also, the data movement (i.e., the number of memory accesses) increases by $2.7\times$. Therefore, disabling the coalescing is not an attractive solution from

the perspective of GPU efficiency.

In this paper, our goal is to design randomized coalescing techniques to carefully balance the security and performance trade-offs. Our techniques exploit two primary shortcomings of the GPU AES implementation that lead to the successful correlation timing (baseline) attack. First, all threads of a warp are grouped in a single subwarp for coalescing. As a result, the calculation of number of coalesced accesses becomes straightforward: a) determine the requested table look up indices, and then b) given that the table elements are sequentially mapped to the memory blocks and the size of each block is known, determine the number of memory blocks (coalesced accesses) required. Second, because all threads of the warp were considered together for coalescing, the order in which the threads are grouped together had no impact on the coalescing. However, if coalescing is performed at a subwarp-level (with number of subwarp being more than one), the order of grouping the threads would affect the total number of coalesced accesses depending on which threads fall into the same subwarp. To address these two shortcomings, we focus on the following three randomized coalescing aspects to weaken the correlation between the coalesced accesses calculated by the baseline attack and the execution time from the encryption.

- **Number of Subwarps:** We choose the number of subwarps that is unknown to the baseline attacker. The benefit of using subwarps is that the attacker may not be able to correctly estimate the number of coalesced accesses. Further, with a large number of subwarps, the variance in the number of coalesced accesses decreases, entailing more number of plaintext samples to establish a weak correlation. This weak correlation reduces the information leakage over the timing channels. We call this defense mechanism as *Fixed Subwarp Size (FSS)*, as the size of subwarp chosen by the defense mechanism is fixed.

- **Size of Subwarps:** In case the attacker knows the number of subwarps (or calculates it based on the timing information), we aim to increase the strength of the defense mechanism by randomizing the number of threads per subwarp such that the total number of threads per warp still remains 32. This randomness makes the number of coalesced

accesses harder to estimate (same reasoning as FSS) even if the number of subwarps is known to the attacker. We call this defense mechanism as *Random Subwarp Size (RSS)* as the size of each subwarp is chosen randomly.

- **Thread Elements of Subwarps:** Our last mechanism is focused on further enhancing the GPU security by randomizing the thread elements of each subwarp (Random-threaded Subwarp (RTS)). It introduces additional randomness in the number of coalesced accesses generated. Note that RTS can be combined with both FSS and RSS defense mechanisms.

3.4 Subwarp based Defense Mechanisms

In this section, we discuss a series of subwarp-based defense mechanisms that are designed to weaken the deterministic memory coalescing logic in GPUs. By doing so, the baseline attack that leverages the knowledge of memory coalescing logic will find it difficult to correctly guess the last round key bytes, thereby improving the security of the GPU-based systems.

3.4.1 Fixed-sized Subwarps (FSS)

In the baseline attack, the attacker assumes that the number of subwarps (*num-subwarp*) is 1, and hence, all threads are processed together for coalescing. In our first defense mechanism, fixed size subwarps (FSS), we break this assumption by choosing a value of *num-subwarp* that is unknown to the attacker. In order to understand the impact of *num-subwarp* on performance, consider Figure 3.7a. We find that the total execution time increases with increase in the value of *num-subwarp*. It is because a large *num-subwarp* leaves few threads for being coalesced together thereby reducing coalescing possibilities across the threads within a warp. This leads to increased number of coalesced accesses resulting in the performance loss.

Advantages of FSS. Although FSS has disadvantage in terms of performance, we find

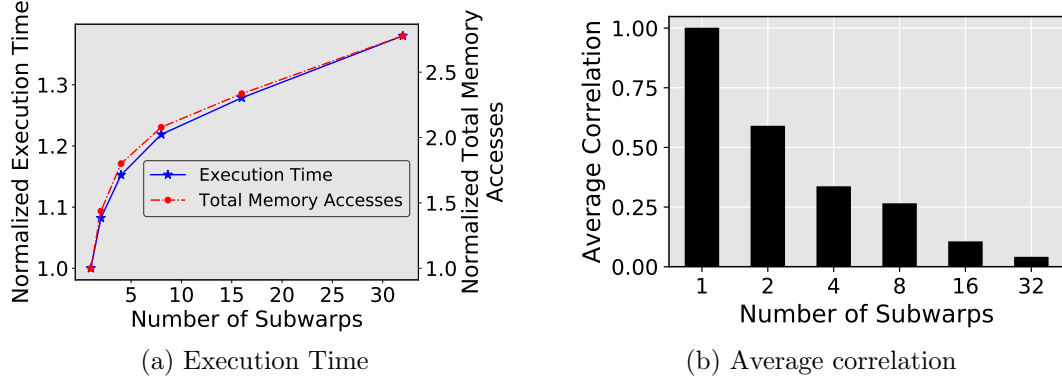


Figure 3.7: Performance of FSS enabled AES with respect to number of subwarps: a) Execution Time and Total Memory Accesses per plaintext with increasing number of subwarps, b) Average of correlations between the last round execution time and the last round memory accesses for all key bytes. The last round memory accesses are calculated assuming correct values of a key byte and the number of subwarps for coalescing to be one. © 2018 IEEE.

that such a mechanism can improve the GPU security against the baseline attack. It is because a value of *num-subwarp* other than 1 will generate different number of coalesced accesses than the baseline attack, which assumes *num-subwarp* to be 1. Therefore, the attacker will find it hard to guess the correct key byte as the correlation between the estimated number of coalesced accesses and the execution time reduces. To understand this further, we evaluate FSS-enabled GPU under the baseline attack. Figure 3.7b shows the average correlations for the correct guesses of all 16 key bytes of the last round key. As expected, we observe that the correlation between the last round execution time and coalesced accesses calculated from the attack reduces with the increase in the value of *num-subwarp*. Therefore, a high number of samples would be required to correctly guess the last round keys depending on the *num-subwarp* value.

Limitations of FSS. We evaluate the security of FSS mechanism when the attacker knows or correctly calculates the value of *num-subwarp*. For example, the calculation can be done based on the significant execution time differences across *num-subwarp* values (Figure 3.7). By repeatably measuring the execution time for encryption for a plaintext, an attacker can determine which *num-subwarp* is used by the remote GPU server. We call

this new attack as “FSS Attack”, where the attacker first calculates the number of last round coalesced accesses generated per subwarp. Next, since the last round execution time correlates with the last round coalesced accesses across a complete warp, the attacker sums up the last round coalesced accesses across all subwarps in a warp. Algorithm 1 illustrates the steps to calculate the number of last round coalesced accesses per warp.

Algorithm 1 Algorithm for FSS attack to calculate the number of last round coalesced accesses for a given key byte guess while considering *num-subwarp*.

```

 $k_j \leftarrow \text{guess\_value}$ 
 $\text{last\_round\_mem\_accesses} \leftarrow 0$ 
for  $i = 0 \rightarrow \text{num\_subwarp}$  do
     $\text{mem\_accesses\_subwarp}[i] \leftarrow 0$ 
for  $\text{grp} = 0 \rightarrow \text{num\_subwarp}$  do
    for  $i = 0 \rightarrow \frac{32}{\text{num\_subwarp}}$  do
         $\text{holder}[i] \leftarrow 0$ 
    % comment: line represents plaintext line
    % comment: LEN represents the total number of lines in the plaintext
    for  $\text{line} = \frac{\text{grp} * \text{LEN}}{\text{num\_subwarp}} \rightarrow \frac{(\text{grp} + 1) * \text{LEN}}{\text{num\_subwarp}}$  do
         $\text{holder}[T4^{-1}[\text{cipher}[\text{line}][j] \oplus k_j] >> 4] ++$ 
    for  $i = 0 \rightarrow \frac{32}{\text{num\_subwarp}}$  do
        if  $\text{holder}[i] \neq 0$  then
             $\text{mem\_accesses\_subwarp}[\text{grp}] ++$ 
for  $i = 1 \rightarrow \text{num\_subwarp}$  do
    if  $\text{mem\_accesses\_subwarp}[i] \neq 0$  then
         $\text{last\_round\_mem\_accesses} \leftarrow \text{last\_round\_mem\_accesses} + \text{mem\_accesses\_subwarp}[i]$ 

```

We evaluate the effectiveness of FSS-enabled GPU under the FSS-attack in Figure 3.8, which illustrates that the attacker is able to establish a high correlation between the number of coalesced accesses and the last round execution time using the FSS attack. Using Algorithm 1, the attacker can calculate the last round memory accesses across the whole warp as observed during the encryption. Therefore, the attacker can establish a high correlation between the calculated number of last round coalesced accesses and the observed last round execution time to successfully recover the last round key. For $\text{num_subwarp} = 32$ (not shown), the variation in the numbers of last round coalesced accesses generated across all plaintexts drops to 0. Subsequently, the correlation between the number of last round coalesced accesses from Algorithm 1 and the observed last round execution time also drops to 0. Therefore, FSS enabled GPU is immune to the correlation timing attacks only when $\text{num_subwarp} = 32$ but at the cost of performance.

In summary, we conclude that the stand alone FSS-enabled GPU cannot provide adequate security against the generalized correlation timing attacks. Therefore, improved defense mechanisms are required.

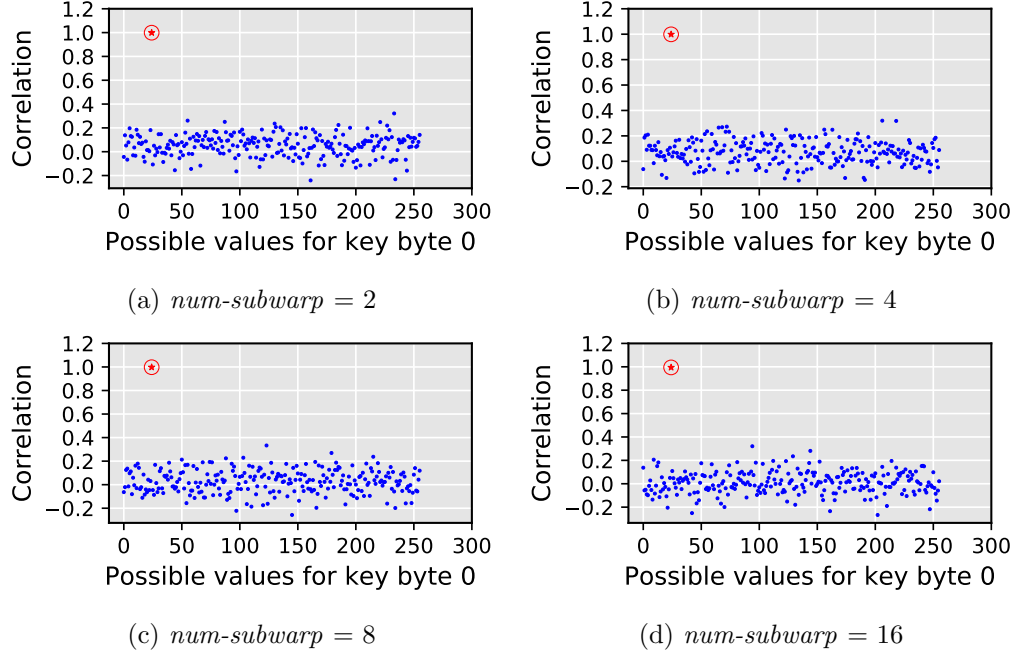


Figure 3.8: Fixed Size Subwarp (FSS) mechanism against FSS attack. © 2018 IEEE.

3.4.2 Random-sized Subwarp (RSS)

In Random-sized Subwarp (RSS) defense mechanism, the size of each subwarp is randomly chosen by the hardware. It implies that the coalescing unit considers different numbers of threads per subwarp for coalescing together. This results in increased randomness in the number of last round coalesced accesses generated per warp leading to reduction in correlation. We consider two distributions to generate sizes of subwarps: normal and skewed. Figure 3.9 shows these distributions for 1000 plaintexts and with the assumption of $num\text{-}subwarp = 4$. In the normal distribution case, the mean of the distribution is close to that of the FSS scenario ($32/num\text{-}subwarp$). According to empirical results (not shown), this implies that security and performance of RSS with normal distribution is

similar to that of FSS.

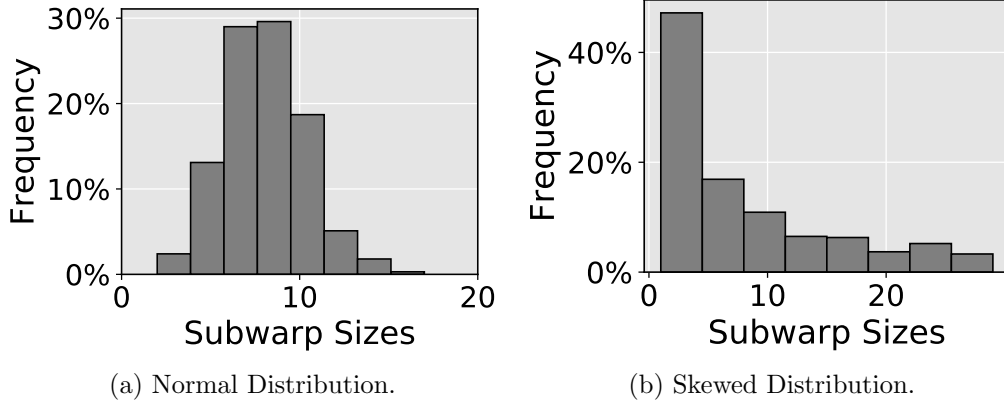


Figure 3.9: Subwarp size distribution of RSS for $num-subwarp = 4$. © 2018 IEEE.

In order to improve security and performance over FSS, we consider skewed size distribution for the RSS mechanism that leads to significant differences in the subwarp sizes. This has two benefits. First, due to the mismatch in the subwarp sizes, the attacker will find it hard to correctly calculate the last round coalesced accesses using Algorithm 1. Second, the skewed distribution also results in an improved performance, since the opportunities for coalescing increases with the subwarp size. Further, we ensure that the skewed distribution considers all possible subwarp size combinations equally likely and no subwarp is empty (a formalization can be found in Section 3.5.2.3). In summary, we use skewed distribution for RSS to improve security and performance.

3.4.3 Random-threaded Subwarp (RTS)

In addition to the size and number of subwarps, we consider an additional level of randomness that comes from the choice of threads that form a particular subwarp. By random allocation of the threads to different subwarps, we eliminate the in-order mapping of threads to the subwarp. We find that such random formation of subwarps significantly changes the number of expected coalesced accesses as the threads processed for coalescing in a subwarp are chosen randomly. We define this technique as Random-threaded Subwarp

(RTS).

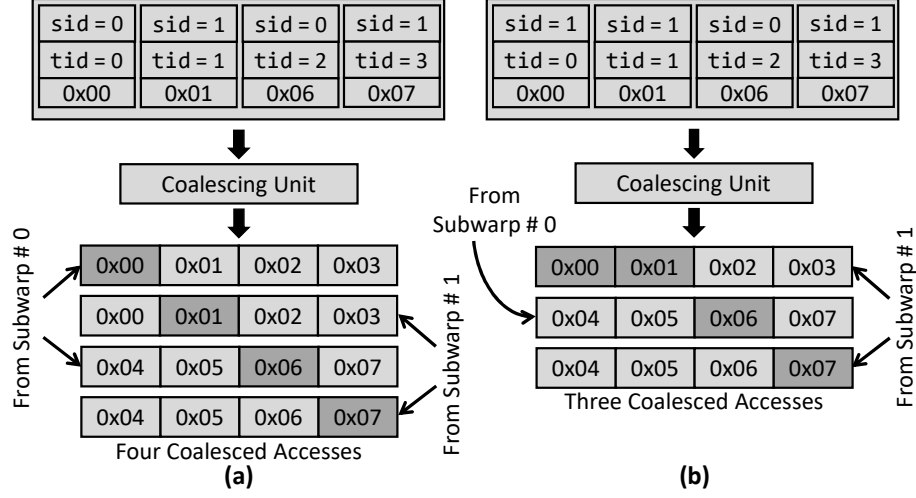


Figure 3.10: Effects of different defense mechanisms on coalescing for $num_subwarp = 2$: a) FSS+RTS and b) RSS+RTS. sid represents subwarp id and tid represents thread id. © 2018 IEEE.

RTS can be applied on top of both FSS and RSS, called as FSS+RTS and RSS+RTS, respectively. Extending the example used in Section 3.2.1 to study the impact of subwarps on coalescing, Figures 3.10a and 3.10b illustrate examples of FSS+RTS and RSS+RTS with 4 threads and 2 subwarps, respectively. In the case of FSS+RTS, the size of both subwarps is 2 but threads are not mapped in order. For example, subwarp 0 ($sid = 0$) has two threads 0 and 2 ($tid = 0$ and 2) instead of threads 0 and 1. Therefore, four coalesced accesses are generated. In the case of RSS+RTS, sizes of the subwarp are different: 1 and 3. Consequently, the mapping of one of threads is changed (i.e., $tid = 0$ is now mapped to $sid = 1$) leading to total three coalesced accesses. In summary, we find that RSS can help in reducing the number of coalesced accesses while providing randomness (along with RTS) for better security.

3.4.4 Implementation Details

In order to implement the proposed subwarp based defense mechanisms, we modify the coalescing unit to allow flexibility in processing of threads for memory access coalescing.

Figure 3.11 shows a schematic of the memory coalescing unit (MCU) of GPU (the additional hardware logic for security is shaded). As described by Leng et al. [61], each MCU contains a multi-entry pending request table (PRT). Each entry in the PRT table stores the thread index (*tid*), the base and offset addresses of the memory requests from the threads, and their sizes. An entry is logged when a memory request is issued from a thread. We add an additional subwarp-id (*sid*) field to identify which threads should be coalesced together. The subwarp-id and thread-id mapping is set by the hardware logic at the beginning of the application execution and does not change during the execution. The logic is dependent on the adopted defense mechanism. In case of FSS and RSS, the bits are set based on the chosen value of *num-subwarp* and the sizing mechanism. The subwarp-ids are allotted in order, that is, first group of threads will belong to the first subwarp with *sid* set to 0 and so on. For RTS, the available *sids* are allotted randomly to the threads in a warp. The additional hardware overhead of our mechanisms is related to the addition of subwarp-id field to each PRT entry. The number of concurrent warp scheduler per SM in our case is two. Therefore, for each SM, the nominal overhead would be $32 \times 2 \times 5$ bits (to represent 32 maximum possible values of *sid*) = 320 bits.

3.4.5 Corresponding Attacks

Similar to the FSS attack, which generalized our baseline attack, we assumed that the attacker is aware of the details of our defense mechanisms implemented on GPU. Therefore, for each defense mechanism, we modified Algorithm 1 to mimic the respective defense mechanism on the attacker's side. For example, against the RSS+RTS enabled GPU, the corresponding attack algorithm simulates RSS-like subwarp size distribution along with random allocation of threads to subwarps within a warp as in RTS. We assume corresponding attacks in the rest of the paper.

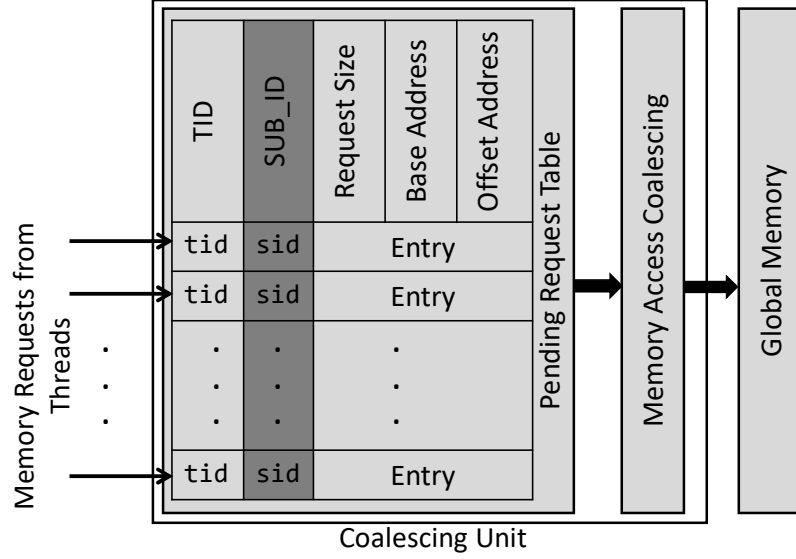


Figure 3.11: Modified Coalescing Unit to realize FSS, RSS, and RTS defense mechanisms. The additional hardware required is the field to store subwarp-id (*sid*) for each thread. © 2018 IEEE.

3.5 Theoretical Security Analysis

3.5.1 Analytical Model

To measure the security strength of the defense mechanisms introduced in Section 3.4, we inspect a natural metric of the (expected) number of samples needed to successfully launch the correlation timing attack.

To estimate that, we use T to represent the *measurement vector*, a vector of the encryption times for a sample set using the actual key. For the j^{th} last round key byte k_j , we use $\hat{U}_{k_j^m}$ to represent the *estimation vector* of k_j^m , a vector of the coalesced accesses for the same sample set if $0 \leq m \leq 255$ were the actual value of k_j . The correlation attack essentially tries to find the value \hat{m} that maximizes the correlation with the measure vector:

$$\hat{m} = \arg \max_m (\rho(T, \hat{U}_{k_j^m}))$$

We follow the derivation in [76, 124] to estimate the number of needed samples, S , for

a successful attack as follows:

$$S = 3 + 8 \times \left(\frac{Z_\alpha}{\ln \left(\frac{1+\rho(T, \hat{U})}{1-\rho(T, \hat{U})} \right)} \right)^2 \approx \frac{2 \times Z_\alpha^2}{\rho^2(T, \hat{U})} \quad (3.4)$$

where \hat{U} is a short hand for $\hat{U}_{k_j^{\hat{m}}}$, ρ represents the correlation and Z_α is the quantile of the standard normal distribution for α , the desired success rate of an attack. With $\alpha = 0.99$, $2 \times Z_\alpha^2$ is approximately 11. Z_α is proportional to α . So the smaller α is, the smaller S is (i.e., fewer samples are needed).

To estimate $\rho^2(T, \hat{U})$, we observe that (as shown in Figure 3.5) the total execution time of AES is proportional to the number of last-round coalesced accesses. Hence, we can draw on the latter in the analytical model². Hence, let us assume that U is the actual vector of number of coalesced accesses from the lookup of table T_4 with respect to the key byte k_j (Equation 3.1 in Section 3.2.2). We can rewrite Equation 3.4 as

$$S \propto \frac{1}{\rho^2(U, \hat{U})} = \left(\frac{\mu(U \times \hat{U}) - \mu(U)\mu(\hat{U})}{\sigma(U)\sigma(\hat{U})} \right)^{-2} = \left(\frac{\mu(U \times \hat{U}) - \mu^2(U)}{\sigma^2(U)} \right)^{-2} \quad (3.5)$$

where μ and σ , as standard, respectively represent the mean and standard deviation of a random variable. The last equation is true since U and \hat{U} are identically distributed.

3.5.2 Analysis of Defense Mechanisms

To make the analysis general, we assume there are in total M subwarps and N threads. Moreover, we assume that each lookup table may map to R memory blocks. As discussed in Section 3.2, our configuration has $N = 32$ and $R = 16$.

We first define three useful definitions.

Definition 1 *Given m threads, if each thread accesses one of n memory blocks in a uni-*

²We note that using the number of coalesced accesses rather than the execution time assure a lower bound on the number of samples since the later is noisier than the former.

form way, then the number of coalesced accesses, $\mathfrak{N}_{m,n}$, obeys the following distribution:

$$P(\mathfrak{N}_{m,n} = i) = \frac{1}{n^N} \frac{n!}{(n-i)!} \left\{ \begin{matrix} m \\ i \end{matrix} \right\}$$

where $\left\{ \begin{matrix} m \\ i \end{matrix} \right\}$ denotes the Stirling number of the second kind.

Here, $\left\{ \begin{matrix} m \\ i \end{matrix} \right\}$ represents the ways of partitioning m threads into i non-empty subsets; $\frac{n!}{(n-i)!}$, i -permutations of n , represents the ways of forming i non-empty subsets from n memory blocks.

It is infeasible to compute Equation (3.5) by enumerating all possible mappings from threads to memory blocks since there are in total R^N possibilities ($16^{32} = 2^{128}$ when $N = 32$ and $R = 16$). However, we note that with RTS, the number of coalesced accesses only depends on the *frequency* of the R memory blocks, which is defined as follows.

Definition 2 For R memory blocks and n threads, we define a frequency set \mathcal{F} as

$$\{(f_1, \dots, f_R) \mid f_1 + \dots + f_R = n\}$$

where $f_i \in \mathcal{F}$ represents the frequency of accessing the i -th memory block among the n threads.

Given a frequency vector $F \in \mathcal{F}$, we note that the “contribution” of each memory block to the number of last-round coalesced accesses U is independent. Hence,

Definition 3 Given a frequency sequence $F \in \mathcal{F}$ and a vector $C = \{c_1, \dots, c_m\}$ that specifies the capacity of each subwarp, if each thread uniformly accesses one of the $|F|$ memory blocks, then the number of coalesced accesses, written as $\mathfrak{M}_{F,C}$, satisfies

$$\mu(\mathfrak{M}_{F,C}) = \sum_{f_i \in F} \sum_{c_j \in C} (1 - C_{f_i}^{S-c_j} / C_{f_i}^S)$$

where C_n^m denotes the binomial coefficient and $S = \sum_{1 \leq j \leq n} c_j$.

Here, $C_{f_i}^{S-c_j}/C_{f_i}^S$ is the probability that the j -th subwarp is empty and $\mu(\mathfrak{M}_{F,R})$ is the sum of the expectations for each subwarp and each memory block.

Next, we derive the (normalized) samples needed for a successful attack for each defense mechanism. We skip the theoretical analysis for the RSS mechanism since it requires enumerating all possible mappings from threads to memory blocks rather than the frequency set, making it infeasible for the calculation. Instead, we provide the empirical results for the RSS mechanism in Section [3.6](#).

3.5.2.1 FSS

With sufficiently random plaintexts, the probability that one thread accesses one of the R memory blocks is $1/R$. Hence, for each subwarp with size N/M , the number of coalesced accesses is $\mathfrak{N}_{N/M,R}$. Since each subwarp is independent, we have

$$\mu(U) = M \times \mu(\mathfrak{N}_{N/M,R}) \quad \sigma(U) = M \times \sigma(\mathfrak{N}_{N/M,R})$$

For $\mu_{U \times \hat{U}}$, we note that given any sequence of memory blocks being accessed by threads, U is identical to \hat{U} . Hence, $\mu(U \times \hat{U}) = \mu(U^2) = \sigma^2(U) + \mu^2(U)$.

3.5.2.2 FSS+RTS

The random permutation does not affect $\mu(U)$ and $\sigma(U)$. For $\mu(U \times \hat{U})$, $(U|F)$ and $(\hat{U}|F)$ are independent and identical for any $F \in \mathcal{F}$. Hence, the term is equivalent to

$$\sum_{F \in \mathcal{F}} P(F) \mu^2(U|F) \tag{3.6}$$

Here, $P(F)$ is the probability of seeing the frequency vector F . Among all R^N combinations of N memory accesses, $C_N^{f_1} C_{N-f_1}^{f_2} \cdots C_{N-\sum_{1 \leq j \leq R-1} f_j}^{f_R} = \frac{(N)!}{\prod_{f_i \in \mathcal{F}} f_i!}$ match F . Hence, we have $P(F) = \frac{(N)!}{\prod_{f_i \in \mathcal{F}} f_i!} \times \frac{1}{R^N}$. Moreover, $\mu(U|F)$ is the same as $\mu(\mathfrak{M}_{F,\{N/M, \dots, N/M\}})$ since each subwarp has size N/M .

3.5.2.3 RSS+RTS

We use U_i to represent the coalesced accesses of the i -th subwarp. With RSS, U_i and U_j are not independent. Hence, we cannot compute $\sigma(U)$ as for FSS.

However, given the size of each subwarp, U_i and U_j are independent for any $1 \leq i, j \leq M$. We use $\mathcal{W} = \{(w_1, \dots, w_M) \mid \sum_{1 \leq i \leq M} w_i = N \wedge \forall_{1 \leq i \leq M} w_i \neq 0\}$ to denote all possible non-empty sizes of subwarps under RSS. Due to uniformity, $P(W) = \frac{1}{|\mathcal{W}|}$ for any $W \in \mathcal{W}$.

For $\mu(U)$, we have $\mu(U) = \sum_{W \in \mathcal{W}} P(W) \mu(U|W) = \sum_{W \in \mathcal{W}} P(W) \sum_{w_i \in W} \mu(U_i|w_i)$ where $\mu(U_i|w_i)$ is the same as $\mu(\mathfrak{N}_{w_i,R})$. For $\sigma(U)$, we know $\sigma^2(U) = \mu(U^2) - \mu^2(U)$ and

$$\begin{aligned} \mu(U^2) &= \sum_{W \in \mathcal{W}} P(W) \mu(U^2|W) \\ &= \sum_{W \in \mathcal{W}} P(W) \left(\sum_{1 \leq i \leq M} \sigma^2(U_i|w_i) + \mu^2(U|W) \right) \end{aligned}$$

Here, $\sigma^2(U_i|w_i) = \mu(U_i^2|w_i) - \mu^2(U_i|w_i)$ and $\mu(U|W) = \sum_{1 \leq i \leq M} \mu(U_i|w_i)$ due to independence. We note that $(U_i|w_i)$ is $\mathfrak{N}_{w_i,R}$ and $(U_i^2|w_i)$ is $(\mathfrak{N}_{w_i,R})^2$. So these terms can be computed via Definition 1.

For $\mu(U \times \widehat{U})$, we can reuse Equation 3.6 since with RTS, $(U|F)$ and $(\widehat{U}|F)$ are independent and identical. Similar to FSS+RTS, $\mu(U|F) = \sum_{W \in \mathcal{W}} P(W) \mu(\mathfrak{M}_{F,W})$ in this case.

3.5.3 Results

Table 3.2: Security analysis results with $N = 32$ and $R = 16$, where N is the number of threads and R is the number of memory blocks. Here, M is the number of subwarps and S is the number of samples normalized to FSS with $M = 1$ case. © 2018 IEEE.

	ρ			S (normalized)		
M	FSS	FSS+RTS	RSS+RTS	FSS	FSS+RTS	RSS+RTS
1	1.00	1.00	1.00	1	1	1
2	1.00	0.41	0.20	1	6	25
4	1.00	0.20	0.15	1	24	42
8	1.00	0.09	0.11	1	115	78
16	1.00	0.03	0.05	1	961	349
32	0.00	0.00	0.00	∞	∞	∞

We use a Python script to compute the correlation and normalized sample size for a successful attack. The results are summarized in Table 3.2.

As expected, when $M = 32$, we have $\rho = 0$ and $S = \infty$ because in this case, each thread is mapped to one subwarp and hence, $U = 32$ regardless of the last-round key. Otherwise, FSS is the least secure ($\rho = 1, S = 1$), where the key can be revealed easily (as shown in Figure 3.8). For both FSS+RTS and RSS+RTS, increasing the number of subwarps reduces ρ and increases S . We note that FSS+RTS is more secure than RSS+RTS for $M = 8$ and 16 though the latter adds randomness to the subwarp size. We hypothesize the reason for this improved security is that one of the subwarps has large size under RSS+RTS most of the times (see, Figure 3.9). In this case, the correlation between measurement vector and estimation vector is higher than that under FSS+RTS. Moreover, the empirical results are consistent with the evaluation (Section 3.6).

Since, in practice, the attacker may observe only the noisy total execution time rather than the last-round coalesced accesses as assumed for the theoretical analysis, the absolute value of needed samples for a successful attack is very large. We note that the FSS mechanism with $M = 1$ is the same as the (baseline) architecture used in [44]. As reported in [44], one million timing samples are needed (if the timing data measured is clean) in this case, and the samples can be collected within 30 minutes. Hence, we estimate that under FSS+RTS with $M = 16$, around one billion samples (refer Table 3.2) are needed for a successful attack. Although such an attack is theoretically possible, it is not practical since collecting timing samples alone may take $(30 \text{ minutes} * 961 \approx) 20$ days.

3.6 Experimental Analysis of Security and Performance

In this section, we present empirical results to support the theoretical results discussed in Section 3.5. We first analyze the security of each mechanism by assessing the key recovery ability using the scatter plots and by inspecting the reduction in the correlation values. Subsequently, we discuss the effects of the proposed mechanisms on performance

and data movement. All the results are collected on a GPU architectural simulator, GPGPU-Sim [6]. Note that it is impractical to execute the attack experiments with a large number of plaintexts on a simulator. However, because of the less noisy environment in the simulators compared to the real hardware, we were able to demonstrate the baseline attack with 100 plaintext samples (each with 32 lines) in Section 3.3. Therefore, for a fair comparison, we use the same number of samples to demonstrate the effectiveness of our defense mechanisms.

3.6.1 Effect on Security

FSS+RTS Attack on FSS+RTS enabled GPU. Figure 3.12 shows four scatter plots each with a different value of *num-subwarp*. Each scatter plot shows the correlation values between the last round execution time and the number of last round coalesced accesses calculated from the FSS+RTS attack algorithm for all the guessed values of the key byte 0. We notice that as *num-subwarp* increases the last round key byte recovery gets difficult as opposed to the standalone FSS defense mechanism. This enhancement in the security is due to the random noise added by the RTS mechanism. Although the FSS+RTS attack implements the random thread allocation in the attack algorithm, it is hard to correctly match the thread allocation order to the one used during the encryption. We conclude that the randomization in the thread allocations allows FSS+RTS to improve GPU security.

RSS Attack on RSS enabled GPU. Figure 3.13 shows four scatter plots each with a different value of *num-subwarp*. Each scatter plot shows the correlation values between the last round execution time and the number of last round coalesced accesses calculated from the RSS attack algorithm for all the guessed values of the key byte 0. For *num-subwarp* greater than 2, we observe that the key byte recovery is difficult as the correlation value for the correct guess is no longer the highest. The drop in the correlation value against the RSS attack is due to the random nature of the subwarp sizing employed in RSS defense mechanism. This random subwarp sizing is changed between the plaintexts and is hard to mimic during the correlation timing attack. Therefore, with the random sizing of the

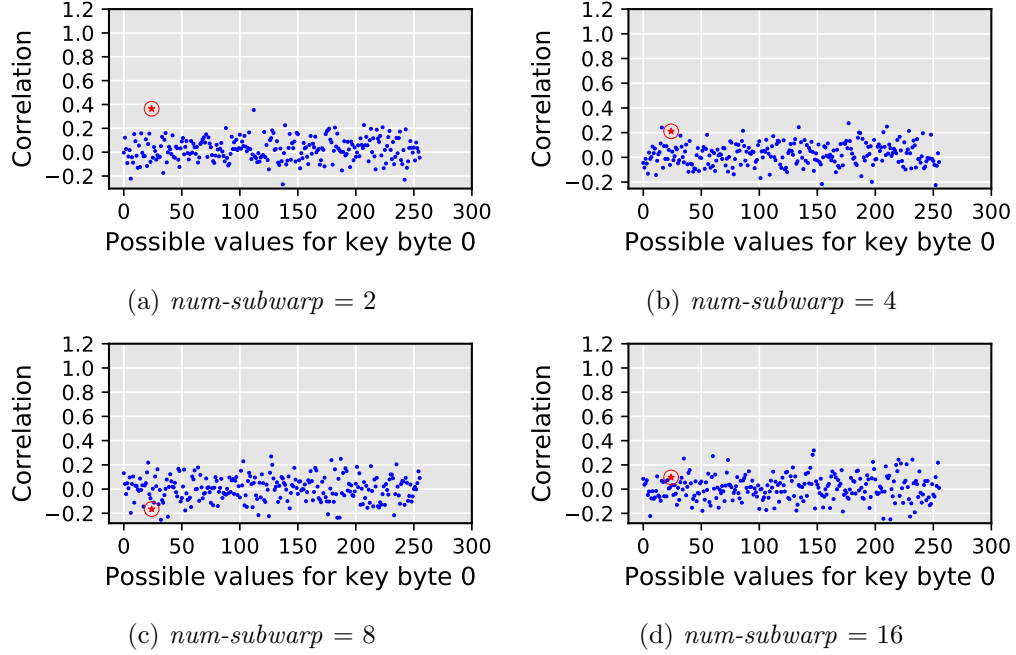


Figure 3.12: FSS+RTS defense mechanism against FSS+RTS attack. © 2018 IEEE.

subwarps, the RSS defense mechanism offers improved security as compared to the FSS defense mechanism.

RSS+RTS Attack on RSS+RTS enabled GPU. Figure 3.14 shows four scatter plots each with a different value of $num-subwarp$. Each scatter plot shows the correlation values between the last round execution time and the number of last round coalesced accesses calculated from the RSS+RTS attack algorithm for all the guessed values of the key byte 0. Similar to FSS+RTS and RSS defense mechanisms, we notice that the recovery of the correct value of the key byte is difficult with the RSS+RTS defense mechanism for $num-subwarp$ greater than 2. The RSS+RTS leverages the randomness in the subwarp sizing and in the thread allocation to the subwarps, which is very difficult to replicate in the RSS+RTS attack. We conclude that RSS+RTS offers security benefits over the FSS defense mechanism.

Security Comparisons. Figure 3.15 compares the security offered by the different defense mechanisms proposed in this work using the average correlation. As noted in Section

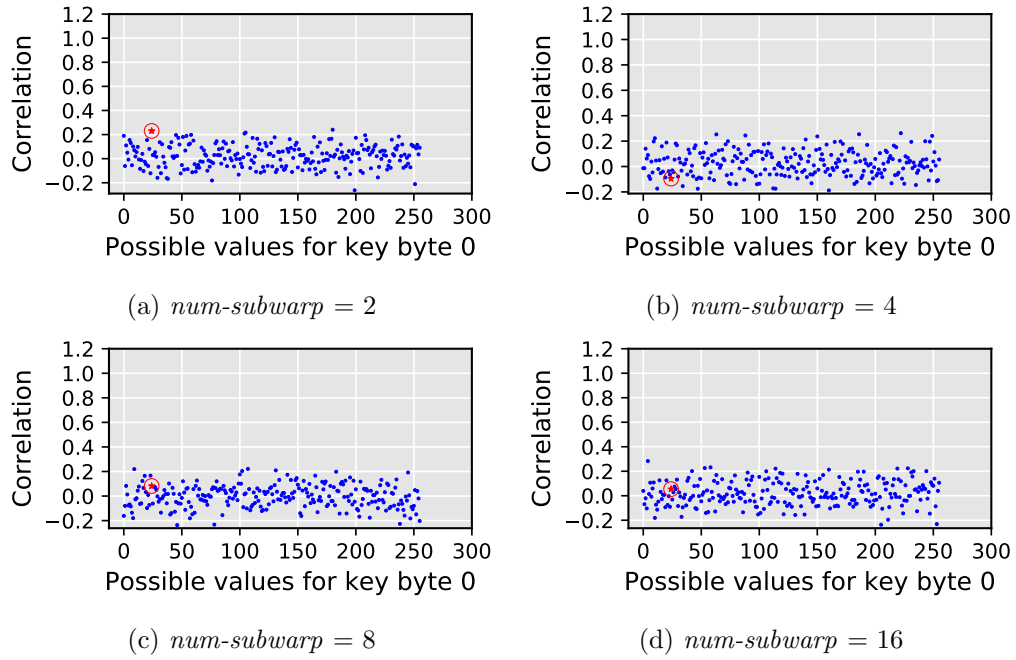


Figure 3.13: RSS defense mechanism against RSS attack.

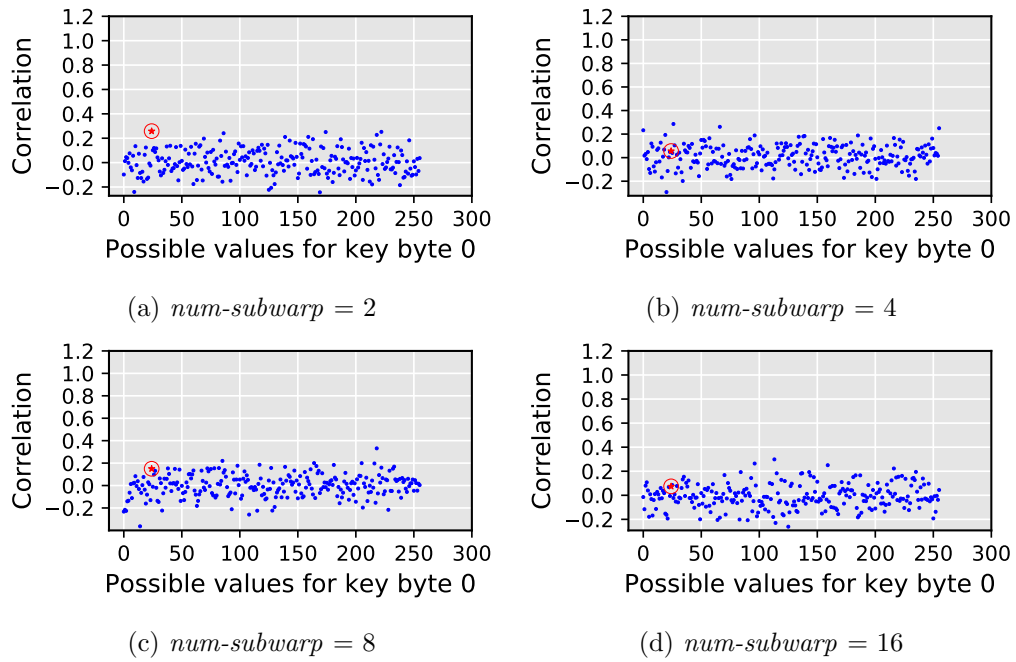


Figure 3.14: RSS+RTS defense mechanism against RSS+RTS attack. © 2018 IEEE.

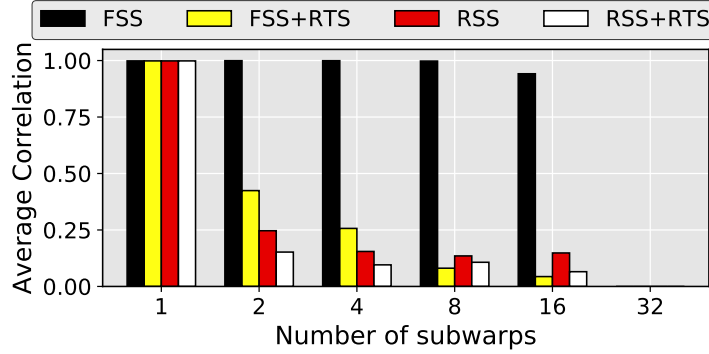


Figure 3.15: Comparison between the security offered by FSS, FSS+RTS, RSS and RSS+RTS based on the average of correlations between the last round coalesced accesses for all key bytes and the last round execution time observed during the encryption. The last round coalesced accesses are calculated using the corresponding attacks. © 2018 IEEE.

[3.4](#), the FSS defense mechanism fails to reduce the correlation as the FSS attack can correctly calculate the last round coalesced accesses. For FSS+RTS, RSS and RSS+RTS defense mechanisms, we observe a decrease in correlation for $num-subwarp = 2$ and 4. We observe slight fluctuations in the respective correlations for RSS and RSS+RTS defense mechanisms for $num-subwarp = 8$ and 16 due to increased randomness in the coalescing. This randomness affects the incorrect guesses of the key bytes as well and results in an overall improved security. Also, we notice that RSS+RTS outperforms all other defense mechanisms for $num-subwarp = 2$ and 4, while FSS+RTS outperforms rest of the defense mechanisms for $num-subwarp = 8$ and 16. For $num-subwarp = 2$ and 4, the RSS+RTS introduces randomness in the coalescing at subwarp sizing as well as at thread to subwarp allocation level. Therefore, the correlation values decreases more in RSS+RTS than in FSS+RTS. However, for $num-subwarp = 8$ and 16, the variance in the last round coalesced accesses is lower in the case of FSS+RTS compared to RSS+RTS. It is because FSS+RTS has more subwarps with the same size compared to RSS+RTS. These findings are corroborated by the theoretical analysis (Table [3.2](#)).

3.6.2 Effect on Performance and Data Movement

Figure 3.16 shows the execution time and the total number of memory accesses with respect to *num-subwarp* for each defense mechanism. In Figure 3.16a, we notice an increase in the total memory accesses with respect to *num-subwarp*. This increase in the memory accesses is attributed to the subwarp based defense mechanisms – FSS and RSS – which reduce the possibility of memory accesses coalescing by dividing the threads of a warp into different subwarps. Therefore, we observe an increase in the execution time as the *num-subwarp* increases (Figure 3.16b). We make two more observations. First, the RTS mechanism does not affect the performance. Although the order of the thread allocation to the subwarps dictates the number of coalesced accesses in a subwarp and hence across the entire warp, the overall effect on performance averages itself out over a large number of plaintexts.

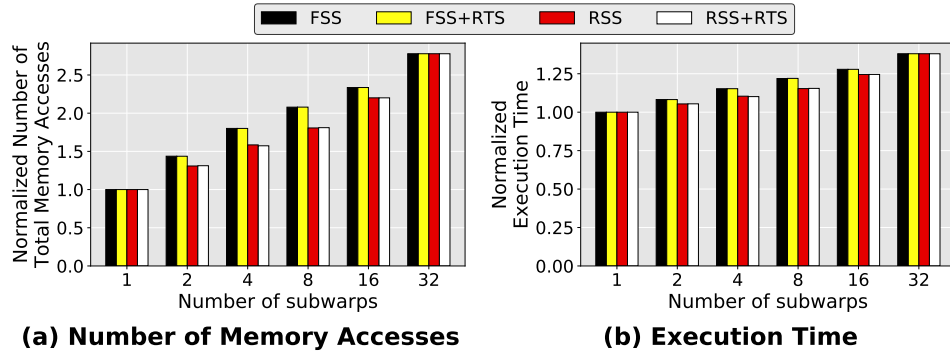


Figure 3.16: Performance and Data Movement Comparisons between FSS, FSS+RTS, RSS, and RSS+RTS. © 2018 IEEE.

Second, the RSS-based mechanisms (RSS and RSS+RTS) show a slightly lower increase in the memory accesses compared to the FSS-based mechanisms (FSS and FSS+RTS). This is because the skewed distribution of subwarp sizes in the RSS-based mechanisms (Section 3.4.2) increases the possibility of a few subwarps to be larger than others. Therefore, on average, the RSS and RSS+RTS defense mechanisms perform better than the FSS and FSS+RTS defense mechanisms.

3.6.3 Evaluating the Trade-off Between Security and Performance

We define the *RCoal_Score* metric, as per Equation 3.7, to allow hardware engineers to achieve a trade-off between the security and performance as per design requirements.

$$RCoal_Score = \frac{S^a}{execution_time^b} \quad (3.7)$$

In the above equation, S is the square of the inverse of the average correlation values calculated from the attack as shown in Figure 3.15. The parameters (a and b) can be set by the hardware engineer to put an appropriate emphasis on either security or performance. For example, Figure 3.17a shows the *RCoal_Score* values for a security-oriented system with $a = 1$ and $b = 1$. We note that FSS+RTS with *num-subwarp* = 8 and 16 is best suited for improving GPU security, albeit with a considerable loss in the performance. For a performance-oriented system, we set $a = 1$ and $b = 20$, as shown in Figure 3.17b. In this case, for *num-subwarp* = 8 and 16, RSS+RTS scores higher than FSS+RTS since it offers an improvement in the performance at a moderate loss in security.

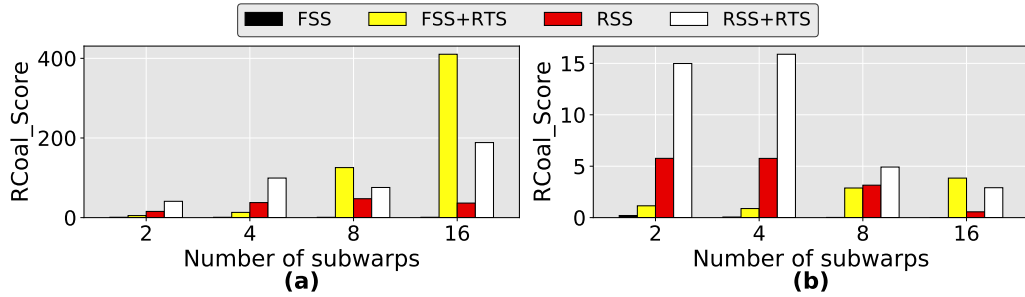


Figure 3.17: Comparison between the FSS, FSS+RTS, RSS and RSS+RTS defense mechanisms based on the RCoal score against the corresponding attacks: a) Security-oriented system with $a = 1$ and $b = 1$, b) Performance-oriented system with $a = 1$ and $b = 20$. © 2018 IEEE.

3.6.4 Case Study: Plaintext with 1024 Lines

We evaluate the scalability of the subwarp based defense mechanisms by increasing the plaintext size to 1024 lines. To negate the ill-effects of the warp scheduling noise during the security evaluation of the defense mechanisms, we correlate the last round coalesced

accesses calculated from the corresponding attacks with the last round coalesced accesses observed during the encryption. It is evident that if the attacker is able to correctly estimate the last round coalesced accesses during the attack, then the correlation will be highest for the correct guess of the key byte leading to a successful recovery of the key. We discuss the security and the performance of each mechanism with respect to *num-subwarp*.

Security. Figure 3.18a shows the average correlation for all key bytes of the last round key for each defense mechanism. As expected, we notice that the average correlation decreases for FSS+RTS, RSS and RSS+RTS mechanisms for *num-subwarp* greater than 1. We conclude that our defense mechanisms improve security on GPUs encrypting large plaintexts as well.

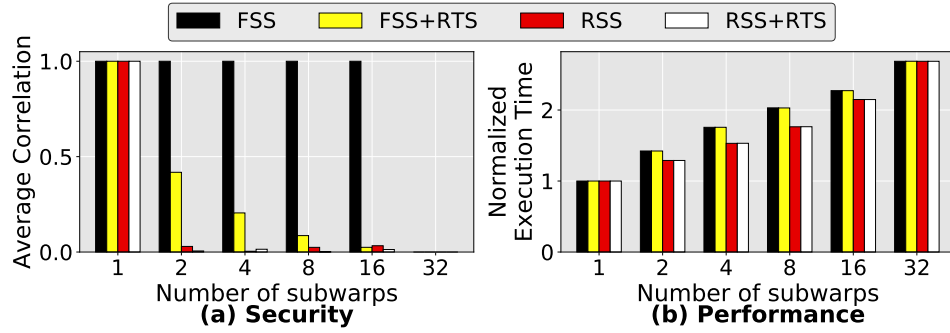


Figure 3.18: Effects of the defense mechanisms on security against the corresponding attacks and performance with respect to the number of subwarps for plaintext with 1024 lines: a) Average of correlations between the last round coalesced accesses from the attack and the execution for all key bytes, b) Execution time (normalized to the case when *num-subwarp*=1) with respect to the number of subwarps. © 2018 IEEE.

Performance. Figure 3.18b shows the execution time for each mechanism normalized to the baseline case of *num-subwarp* set to 1. As in the case of plaintext with 32 lines, we note that the RTS mechanism does not affect the execution time. Also, the the execution time increases with *num-subwarp*. Additionally, as earlier, RSS-based mechanisms increase the coalescing possibilities and deliver better performance than the FSS-based mechanisms. In conclusion, we observe that RSS+RTS mechanism offers an improved security with performance degradation in the range of 29 to 76% for *num-subwarp* = 2, 4, and 8. This

indicates that the defense mechanisms presented in this work scale well with the plaintext size.

3.7 Discussion and Future Work

In the context of RCoal, we discuss the following two future research directions.

- The current implementation of RCoal spans over the entire execution of the AES and assumes that all rounds are equally vulnerable [101]. The advantage of such an implementation is that it does not require software support to identify the vulnerable portions (rounds) of the code. To enhance the performance further, RCoal can be limited only to the vulnerable part of the code. However, that would require software support to correctly identify the vulnerable portions of the code and hardware support to frequently turn coalescing on and off based on which warps are executing the vulnerable code at a given time. We leave the development of such hardware/software support as a part of the future work.

- We presented a series of defense mechanisms that focused only on the intra-warp coalescing techniques. Therefore, we disabled other bandwidth conserving optimizations in GPUs (e.g., MSHRs and caches). However, we believe our proposed intra-warp coalescing will be more effective if randomization is employed at all levels of the memory hierarchy. We leave the development of these randomization techniques as a part of the future work.

3.8 Related Work

To the best of our knowledge, this is the first work that proposes randomized coalescing mechanisms to thwart timing attacks in GPUs. In this section, we list works relevant to ours.

Timing attacks. Cryptographic algorithms implemented on CPUs have been the major targets of timing attacks. Those attacks exploit the fact that key-dependent memory accesses, such as table-lookups in AES, affect the memory access patterns and hence, the

status of data cache. Hence, an attacker may infer private keys by observing the execution time of either a cryptographic algorithm (e.g., [102, 9, 8, 30]), or his own application if the data cache is shared (e.g., [30, 142, 40, 145]).

Pietro et al. [107] identified that the memory leaks are possible at various levels of GPU memory hierarchy, especially at software-managed scratchpad memory and register file. A recent work [45] exploits a new fine-grained timing channel caused by bank conflicts in a GPU’s shared memory. A complete AES key recovery timing attack was first demonstrated on a commercial GPU architecture by Jiang et al. [44]. We have already extensively discussed this attack and proposed defense mechanisms that trade-off performance for security.

Timing channel mitigation. Several hardware-based timing attacks have been proposed in the context of CPUs [102, 67, 144, 132, 133, 70, 136]. Among those works, more related are mechanisms based on randomization [132, 133, 70, 136]. Most of these works randomize the memory-to-cache mapping or the cache replacement policy, while our work proposes to randomize the coalescing behavior.

Coalescing and Bandwidth Saving Techniques in GPUs. Kloosterman et al. [55] proposed warp-pool, an enhanced inter-warp sharing mechanism to reduce global memory accesses. Rhu et al. [112] proposed cache sectoring mechanism to reduce unnecessary data fetches from global memory. A series of warp scheduling techniques [47, 51, 114, 113] have been proposed to reduce cache misses and improve memory bandwidth utilization. None of these works focused on hardware security issues, as we do in this paper.

3.9 Conclusions

Our findings confirm that the deterministic nature of the coalescing logic is a major cause of security vulnerability in GPUs. To address this vulnerability, we propose a series of defense mechanisms that allow the coalescing logic to randomly change the number of coalesced accesses. Specifically, we propose to randomize: a) the granularity at which

intra-warp coalescing is performed in the baseline architecture, and b) allocation of the thread elements per subwarp. Our theoretical and empirical results show that our randomized coalescing defense mechanisms significantly improve the GPU security at a modest performance loss.

Chapter 4

BCoal: Bucketing-based Memory Coalescing for Efficient and Secure GPUs

4.1 Introduction

Graphics Processing Units (GPUs) provide orders of magnitude higher throughput compared to CPUs thanks to a large number of computational units attached with high bandwidth memory. GPUs have traditionally accelerated a wide-range of arguably security insensitive applications ranging from gaming to high-performance computing. However, many applications that benefit from GPUs nowadays process or contain security/privacy-sensitive information. For example, DNA and financial computing applications that heavily process private data are taking advantage of GPUs [94, 92]. The deep learning community has significantly benefited from the computational power of GPUs but now is also concerned about the privacy of their models and vendors; they are interested in protecting them from motivated attackers [37, 81]. Cryptographic and other computations that handle sensitive data are also known to achieve significant performance benefits from GPUs [49, 44, 128, 72, 17, 83, 42].

With the growing need for secure GPU computation, it is important to protect GPUs from a variety of possible side-channel attacks. For example, several attacks (especially, cache-based side-channel attacks [131, 132, 133, 70, 71, 111, 141]) on the CPU side have exploited the fact that critical information can be leaked if it affects the latency (or total execution time). In the same vein, new correlation timing attacks and covert channels [44, 80, 135, 45] are being exposed in GPUs – a recent attack [44] showed that AES private keys can be recovered by exploiting the correlation between the number of coalesced accesses and execution time. Specifically, an attacker exploits the relationship between the private keys and the number of coalesced accesses to reveal the entire private key by performing off-line correlation analysis with the help of recorded execution time and encrypted (cipher) text information.¹

Kadam et al. [49] presented the first work to address the aforementioned correlation timing attack. They showed that by randomizing the logic of coalescing unit (RCoal), additional accesses can be generated such that the correlation between the baseline (real) accesses and the execution time is reduced. Consequently, the attacker finds it hard to recover the private keys. However, we find that RCoal has two major drawbacks. First, the performance loss for security gain is very high due to the randomization of coalescing logic, especially for large plain texts. Second, RCoal provides sub-optimal security in the presence of other memory bandwidth conserving mechanisms such as miss-status holding registers (MSHRs) and caches. As we further demonstrate in Section 4.3, the additional duplicate accesses generated during randomization are merged back in MSHRs to render RCoal ineffective. Therefore, RCoal turned-off caches and MSHRs for security reasons, leading to even more significant performance overheads.

To efficiently address the limitations of RCoal, we propose a new bucketing-based coalescing technique – BCoal. It always generates the number of coalesced accesses equal to one of the pre-determined values (known as buckets), irrespective of program secrets. This implies BCoal would generate additional memory accesses (if necessary) along with

¹More details on the attack are provided in Section 4.2

the real accesses to match the bucket requirements. As the number of accesses is always equal to the pre-determined values, the variance in the number of accesses drops. As a result, BCoal reduces the correlation to mitigate the timing attack.

To reduce the performance overhead of additional accesses, we select optimal bucket features by analyzing the application-level coalescing profile. The goal of profiling is to select the bucket features such that overall fewer additional accesses are generated. Further, we observe that the generation of additional accesses is non-trivial because we need to ensure that they affect the execution time at the same rate as the real accesses, otherwise their effect on the execution time can be filtered out (i.e., noise can be filtered out from signal). To address this issue, we generate *unique* additional accesses to the same memory space as that of the real accesses. We find that this helps in reducing the disparity between caching/merging probabilities of real accesses and additional accesses, thereby making their individual effects on execution time also similar. Consequently, our bucketing-based coalescing technique provides security even in the presence of MSHRs and caches.

To the best of our knowledge, this is the first work that proposes a bucketing-based coalescing technique for GPUs to achieve better security compared to the state-of-the-art scheme while incurring low overhead. In summary, this paper makes the following contributions:

- We perform a detailed analysis to show that the state-of-the-art defense schemes against the coalescing-based correlation timing attack are inefficient. They incur a significant performance and data movement overhead as they work only when the bandwidth conserving hardware such as caches and MSHRs are not employed.
- We propose a new bucketing-based coalescing mechanism (BCoal) that always issues pre-determined numbers (chosen from a small set, called buckets) of coalesced accesses by padding additional accesses to the real accesses, if necessary.
- Our analysis shows that the generation of padded accesses is non-trivial and the effect of MSHRs and caches should be considered to ensure security. BCoal implements a

homogeneous padding mechanism to ensure that the real and padded accesses affect the execution time *similarly* even in the presence of MSHRs and caches. Therefore, an attacker fails to separate the timing effect of padded accesses thereby improving the security.

- Our theoretical and experimental analysis shows that BCoal significantly improves the security (i.e., drops the correlation by up to 100%) at a modest performance overhead ranging from 5% to 15%. We also evaluate BCoal across a large set of GPGPU applications and show that coalescing with three equally-spaced buckets provides an excellent performance-security trade-off that can be leveraged to secure the GPUs.

4.2 Background

This section briefly introduces: a) the baseline GPU architecture, b) bandwidth conserving mechanisms, c) the AES encryption on GPU, and d) the baseline correlation timing attack and the state-of-the-art defense mechanism against it.

4.2.1 Basics of GPU Architecture

We consider a baseline GPU architecture with multiple cores, known as streaming multiprocessors (SMs) in NVIDIA terminology. The SMs are connected to memory partitions via an interconnect as shown in Figure 4.1. GPUs achieve high throughput by executing a large number of threads concurrently. To facilitate this, GPUs are supported by a large register file (for fast context switching across threads) and high bandwidth memories (for fast data access to a large number of concurrent threads). Each SM executes the threads assigned to it at the granularity of a *warp*, which is essentially a collection of (usually 32) individual threads that execute a single instruction on the processing elements (PEs) of the SM in a lock-step. The warps hide long memory latencies to improve the utilization/throughput of the SM via executing in a pipelined and multiplexed manner. Throughout the paper, we evaluate the proposed techniques on a cycle-level GPU simulator – GPGPU-Sim [6]. Table 4.1 provides details of the simulated architecture.

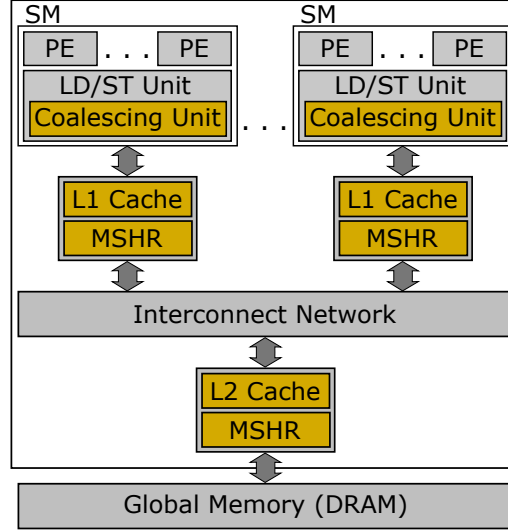


Figure 4.1: Overview of Baseline GPU Architecture. © 2020 IEEE.

Table 4.1: Key configuration parameters of the simulated GPU. © 2020 IEEE.

Core Features	1400MHz core clock, SIMT width = 32 (16×2)
Resources / Core	32KB shared memory, 32KB register file, 15 SMs
L1 Caches / Core	16KB 4-way L1 data cache, 2KB 4-way I-cache 128B cache block size
L2 Caches	16-way 256 KB/memory channel (1536 KB in total), 128B cache block size
Features	Inter-warp merging enabled
Memory Model	6 GDDR5 Memory Controllers, FR-FCFS scheduling 16 DRAM-banks, 924 MHz memory clock
Interconnect	1400MHz interconnect clock

4.2.2 Bandwidth Conserving Mechanisms

Memory bandwidth is one of the most performance-critical shared resources in GPUs [47, 46]. GPUs adopt several memory bandwidth optimization techniques, such as memory access coalescing, caching and merging to reduce the number of accesses to the global memory. In this sub-section, we provide a brief overview of these optimizations.

Access Coalescing. In GPUs, threads within a warp execute the instructions in lock-step. For a global memory load instruction, all 32 threads within a warp execute 32 load instructions. The coalescing unit in the LD/ST unit merges multiple memory requests from different threads of the same warp (*intra-warp coalescing*) into as few cache line-sized coalesced memory accesses as possible. The intra-warp coalescing happens at the sub-warp granularity, where the coalescing unit of the SM determines the coalesced ac-

cesses of the warp by examining a group of threads belonging to the same sub-warp. If the threads of a sub-warp access data within a contiguous memory block, their requests are coalesced together to reduce memory bandwidth consumption. The size and number of sub-warps are typically fixed and remain the same throughout the application execution. However, to achieve security, the coalescing mechanisms can be randomized (RCoal [49]) so that the coalesced accesses are no longer predictable to the attacker.

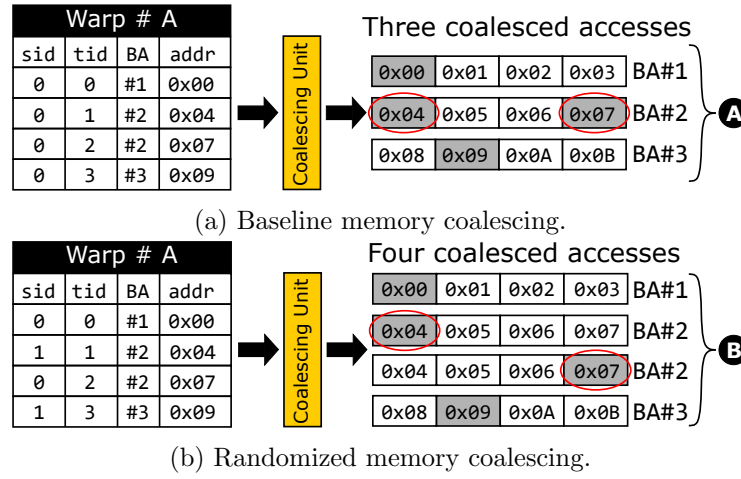


Figure 4.2: Memory access coalescing in GPUs. © 2020 IEEE.

Figure 4.2 illustrates the coalescing in baseline GPU and previously proposed randomized coalescing techniques. Assume a single warp with four threads. The per-thread addresses and the requested block addresses (BA) are shown with corresponding thread-ids (tid) and sub-warp id (sid). In the baseline GPU, we assume a single sub-warp (sid = 0 for all threads) and hence all threads participate together in the coalescing. Since the requests from tid 1 and 2 map to the same cache block, only three accesses are generated (A) to conserve the memory bandwidth. With randomized coalescing, the threads are randomly assigned to subwarps and hence lead to unpredictable effects on coalescing. In Figure 4.2(b), we observe that four accesses are generated (B) due to different sub-warp ids assigned to the random groups of thread. More details on the randomized coalescing techniques are discussed in Section 4.2.4

Caching. GPUs further conserve the memory bandwidth by exploiting the temporal

and spatial locality in memory accesses across and within warps with the help of hardware caches. Current GPUs employ two levels of caches, L1-cache (shared by the warps executing on the same SM) and L2-cache (shared by the warps executing on different SMs).

Access Merging. The coalesced memory accesses from a warp are sent to the L1-cache. Upon cache misses, the memory accesses are logged in the miss-status holding registers (MSHRs). Multiple cache-missed coalesced accesses to the same cache block from different warps on the same SM are merged (*inter-warp merging*) in MSHRs. Note that as independent loads from the same warp can be issued to improve memory-level parallelism, MSHRs also help in merging redundant accesses from the same warp (*intra-warp merging*) if they are issued at different times. Another source of inter-warp merging is via MSHRs at L2-cache, where the redundant L2-cache misses (across different SMs) can be merged together.

4.2.3 AES Encryption

To demonstrate the GPU timing attack exploiting the vulnerability due to memory coalescing, we consider the widely used symmetric-key algorithm, Advanced Encryption Standard (AES) [78, 34, 41, 89, 66] with a key length of 128 bits, to encrypt the plaintext. AES-128 algorithm consists of 10 rounds, each with a 16-bytes round key generated from the encryption key. We focus on the last round of the AES, which is shown to be the most vulnerable to side-channel attacks [44]. The last round involves a table (for the S-box table T_4) look-up operation followed by bitwise XOR operation with the last round key.

Our AES implementation on GPU is from Jiang et al. [44, 42], which was used in the original attack [44] and a known defense [49]. We used the same implementation for a fair comparison. The AES implementation on GPU involves dividing the plaintext across multiple parallel threads to achieve high throughput. Each thread encrypts a line of the plaintext independent of other threads. Therefore, a warp consisting of 32 threads can perform 32 different encryptions concurrently. In general, the line to thread mapping

is sequential and deterministic. If the size of the plaintext exceeds 32 lines, then it is divided sequentially among several warps. For example, a plaintext with 1024 lines will employ 32 warps each executing 32 lines of the plaintext. To ensure a stronger baseline for comparison, the AES implementation used in this paper performs random mapping of threads to the warps (known as input blinding) to gain additional security [49].

4.2.4 Baseline Attack and Defense Mechanism

Baseline Attack. In this work, we use the same attack model as designed by Jiang et al. [44]. It assumes that the attacker can send a large number of plaintexts to a remote GPU-based AES [78, 34, 41, 89, 66] encryption server and collect the ciphertext. The attacker also records the total execution time required to complete each encryption. The attack was also shown to be very effective in noisy environments [44].

Given that the GPU coalescing procedure is deterministic [61] and the last round of AES is invertible [44], the attacker can calculate the number of coalesced accesses with the help of ciphertext and a last round key guess. As the number of coalesced accesses is correlated with the execution time in the baseline system [44], the key guess that leads to the best correlation across a large number of encryptions is determined to be the correct key. This attack further assumes that the round tables are kept in GPU DRAM, which can be cached in L1/L2 caches based on the access patterns. For brevity, we skip the algorithmic details of the attack and refer readers to prior works [49, 44]. Also, the rest of the paper assumes a stronger attacker with the capability of accessing last round execution time as compared to the realistic attack, which is weaker due to the noise in the total execution time. Consequently, we assume the goal of the attacker is to correctly guess the last round AES encryption key [49], which can divulge all other round keys by reverting the fixed AES key generation schedule.

Figure 4.3 shows the scatter plots for the baseline correlation attack for the single-warp (plaintext with 32 lines) and multi-warp (plaintext with 64 lines) cases. Each scatter plot shows the correlation values for all 256 possible values for the 3^{rd} key byte of the last

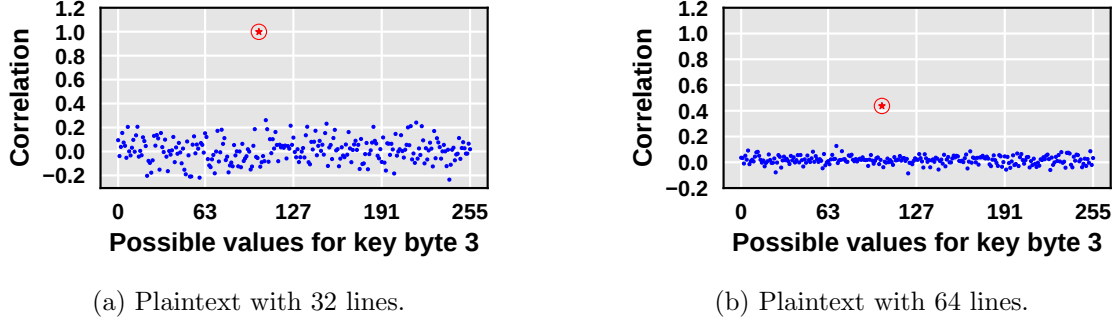


Figure 4.3: Baseline Attack. © 2020 IEEE.

round. Each point on the scatter plot corresponds to a correlation value between the number of coalesced accesses (on a per-warp basis) calculated by the attacker and the execution time of the last round of AES-128. In the multi-warp case, the maximum value of the number of coalesced accesses across all warps is used as the warp that generates the most number of coalesced accesses has shown to dominate the total execution time [44]. From Figure 4.3, we observe that the correlation value is the highest (highlighted in red and encircled) for the correct value of the 3rd key byte among all other guess values for the single- as well as the multi-warp case. Therefore, the correct value of the key byte 3 is recoverable. We observe this trend for all last round key bytes indicating successful recovery.

Baseline Defense. Kadam et al. [49] presented a series of randomized coalescing (RCoal) mechanisms to defend against the correlation timing attacks. They showed that randomizing the number of subwarps, the sizes of subwarps, and the thread elements of the subwarp can improve the GPU security, however at the cost of performance loss and increased data movement between SMs and memory. Based on these three parameters, three RCoal mechanisms were proposed: fixed-sized subwarp (FSS), random-sized subwarp (RSS), and random-threaded subwarp (RTS). They showed that the best performance-security trade-off can be achieved with an RCoal mechanism (RSS+RTS+4), which uses the number of subwarps to be 4, the sizes of warps are chosen based on a skewed distribution, and the thread elements are chosen randomly based on a uniform distribution. In rest of the

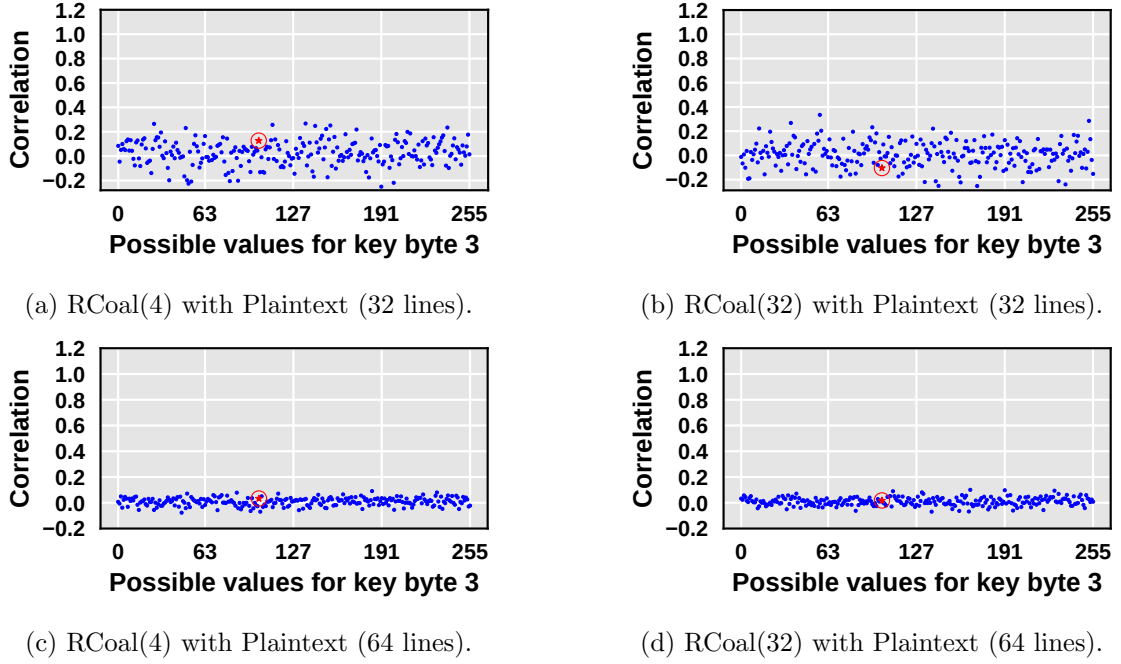


Figure 4.4: Effect of different RCoal coalescing schemes on the recovery of one of the last round key byte (shown in red circle). In RCoal, the caches and MSHRs are disabled for security reasons (refer Section 4.3-B). © 2020 IEEE.

paper, we denote this best of the RCoal scheme as RCoal(4). Note that if the number of subwarps is equal to the number of threads in a warp then it is equivalent to coalescing being disabled as all threads independently participate in the coalescing procedure. For example, with a warp size of 32, choosing the number of subwarps to be 32 is equivalent to disabling the coalescing. We denote this as RCoal(32). RCoal(32) was shown to be the most secure design as the number of coalesced access is always constant at 32 [49]. Due to security concerns, RCoal disabled caches and MSHRs (refer to Section 4.3.2 for more details).

Figure 4.4 shows the scatter plots for RCoal(32) (the most secure mechanism) and RCoal(4) (best of RCoal) using plaintext with 32 and 64 lines. In contrast to the baseline attack, for RCoal(32) and RCoal(4), the correlation between the number of coalesced accesses and execution time with the correct key (highlighted in red and encircled) dropped significantly. Consequently, this point is no more distinguishable among the other corre-

lation points ensuring successful defense against the attack. We observe this trend for all last round key bytes.

4.3 Motivation and Analysis

Although RCoal helps in improving the GPU security significantly, it also incurs a very high performance and data movement overhead. To substantiate the overhead of RCoal, Figure 4.5 shows the total execution time and number of DRAM accesses for two scenarios: a) RCoal(32) – the most secure design, and b) RCoal(4) – the best of RCoal. These results are shown for three different sizes of plaintexts (32, 64, and 1024) and are normalized to the baseline GPU. We observe that the overhead of RCoal(32) is very high – more than $27\times$ increase in the number of DRAM accesses leading to over $9.4\times$ increase in the execution time. Furthermore, the performance degradation increases rapidly with the size of plaintexts. The same trend is visible for RCoal(4) as well.

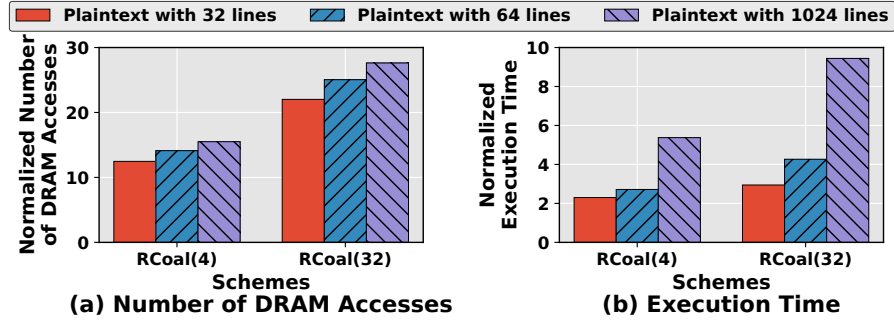


Figure 4.5: Illustrating the overhead of RCoal defense scheme for different sizes of plaintext. The results are normalized to a baseline GPU with MSHRs and caches. © 2020 IEEE.

4.3.1 Performance Overhead Analysis of RCoal

There are two major reasons behind the large performance and data movement overhead. First, RCoal introduces sub-optimal and randomized coalescing that causes additional memory traffic. To understand this, we analyze the number of coalesced accesses generated

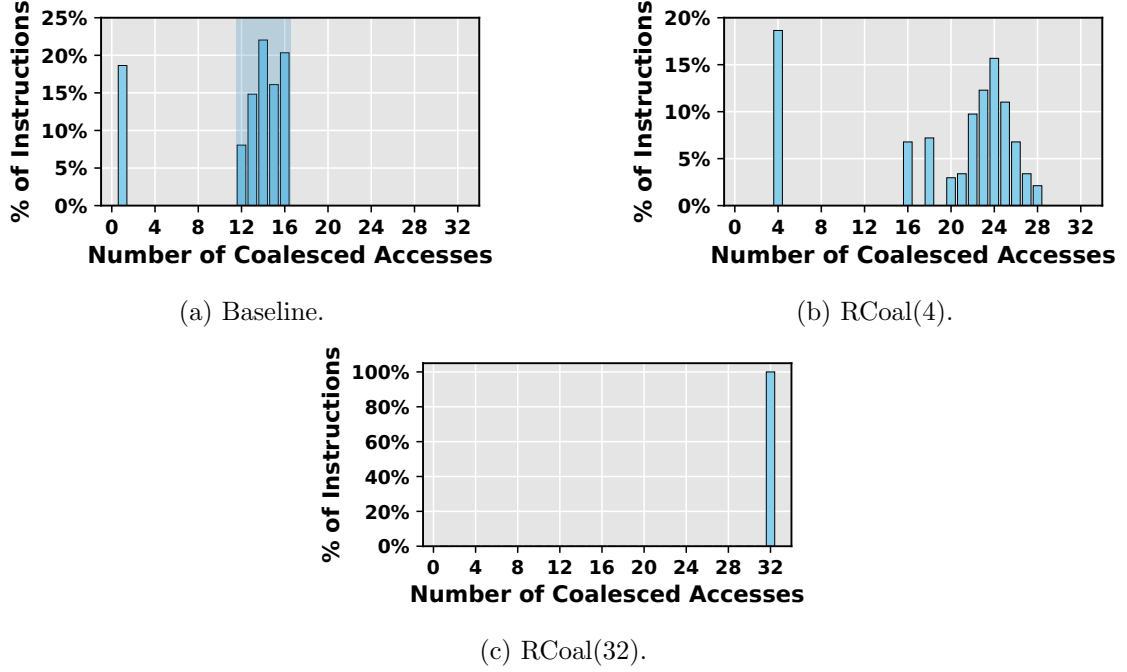


Figure 4.6: Histogram of the number of coalesced accesses generated across a warp for 1000 plaintext samples each with 32 lines. © 2020 IEEE.

in three different architecture options: baseline, RCoal(4), and RCoal(32). For these three options, Figure 4.6 shows the number of coalesced accesses with respect to the percentage of load instructions in the AES CUDA implementation. We observe a bimodal distribution in the baseline scenario (Figure 4.6(a)): the first peak occurs when only one coalesced cache line access is generated for roughly 20% instructions and the second peak occur between 12-16 coalesced cache line accesses for the remaining instructions. The first peak is observed due to the loads for the round keys and the second peak is due to the table lookup operations. With RCoal(32) (Figure 4.6(c)), the coalescing unit performs worst to always generate 32 coalesced accesses for all load instructions. As noted before, this is similar to the coalescing being disabled. Although it is the most secure option, the average number of coalesced accesses and the overall number of DRAM accesses increase significantly (Figure 4.5). In RCoal(4) (Figure 4.6(b)), we observe that the second peak has shifted to the right compared to Figure 4.5(a) due the obfuscation of the coalescing mechanism that generates additional memory traffic. Overall, RCoal(4) and

RCoal (32) generate additional memory traffic and incur performance penalties to reduce the correlation between the number of baseline coalesced accesses and the execution time. Importantly, RCoal ignores the application properties, especially the baseline coalescing profile to optimally generate the traffic while reducing the correlation.

Second, due to the security reasons, RCoal schemes were only shown to work in the absence of other bandwidth optimization techniques, such as caches and MSHRs. The absence of MSHRs and caches has a substantial impact on the performance and data movement, and is well-documented in GPU literature [6, 47, 117]. The combined effect of sub-optimal coalescing, and absence of MSHRs and caches leads to a sharp increase in the number of DRAM accesses resulting in high performance degradation.

4.3.2 Effect of MSHRs and Caches on Security with RCoal

Effect of MSHRs. In the presence of MSHRs, RCoal scheme becomes vulnerable to the correlation timing attacks. RCoal randomizes the access coalescing and generates redundant accesses to the same block addresses to reduce the correlation between the execution time and the number of baseline coalesced accesses. The MSHRs render RCoal scheme ineffective by merging the redundant accesses to the same block addresses leading to similar correlation as in the case of baseline GPU. The effect of MSHRs on RCoal scheme is prominent for the table lookup instructions experiencing a high cache-miss rate as the corresponding accesses are likely served through MSHRs leading to predictable access merging. This is especially true for the initial table lookup instructions of the last round because T_4 table elements are less likely cached. Figure 4.7 shows this merging-back phenomenon using the example from Figure 4.2. RCoal(4) generated 4 accesses (A), including one redundant access. However, MSHRs merged back the cache-missed accesses, leading to the same number of accesses (B) generated to the DRAM as that of in the baseline case. Consequently, it leads to the same correlation and information leakage as that of the baseline GPU.

Effect of caches. The security of RCoal depends on the cache hit rates. For example,

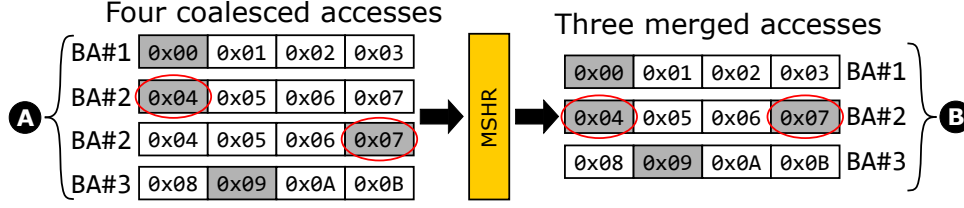


Figure 4.7: Effect of MSHRs on the cache-missed coalesced accesses in RCoal scheme. © 2020 IEEE.

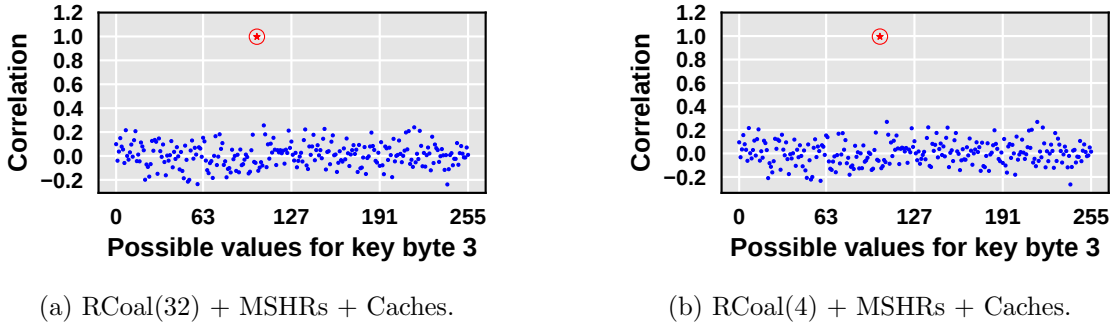


Figure 4.8: The presence of MSHRs and caches leads to successful recovery of one of the last round key bytes in RCoal(32) and RCoal(4). Plaintext has 32 lines. © 2020 IEEE.

in the case of RCoal(32), if all accesses of a table lookup instruction are *always* cached, then all 32 accesses from the coalescing unit are served by the cache. Therefore, if the execution time remains constant due to the constant number of accesses to the cache, the attacker cannot establish the correlation between the number of baseline coalesced accesses and the execution time to reveal the private key. However, a perfect cache hit rate cannot be guaranteed for all the table lookup instructions across a large number of plaintext samples. Therefore, if the accesses of a table lookup instruction miss in the cache, the key byte can still be recovered with RCoal due to the access merging in MSHR as discussed earlier. To illustrate this point, Figure 4.8 shows the scatter plots for the first table lookup instruction of the last round with MSHRs and caches enabled. We note that the private key byte 3 corresponding to the first table lookup instruction can easily be recovered in both the RCoal scenarios.

In summary, RCoal becomes vulnerable due to the access optimizations in MSHRs and caches.

4.3.3 Our Proposal and Goals

Our goal is to design a mechanism that reduces the performance overheads of RCoal while offering comparable security. To this end, we propose BCoal: a bucketing-based coalescing mechanism to address the primary performance-related shortcomings of RCoal discussed before. BCoal matches the number of coalesced accesses generated for a global memory load instruction per warp to one of the predetermined values (denoted as *buckets*). To match the number of accesses to one of the preset bucket sizes, we pad the real coalesced accesses from a warp with additional (padded) memory accesses. Since the total numbers of accesses always match one of the bucket sizes, their overall variance decreases. Furthermore, as observed earlier for RCoal, MSHRs adversely affect the security by merging the redundant accesses after randomized coalescing. Therefore, the padding mechanism in BCoal is devised such that MSHRs cannot merge the real and padded accesses, thereby maintaining a very low variance in the resulting number of accesses. Additionally, the padding mechanism ensures that the real and padded accesses follow similar access merging and caching pattern, such that they affect the execution time at the same rate. Subsequently, the individual effects of real and padded accesses on the execution time are indistinguishable. Therefore, in BCoal-enabled GPU, the attacker will not be able to correlate the number of real coalesced accesses with the observed execution time. Consequently, the security offered by BCoal scheme against the correlation timing attacks remains intact even in the presence of MSHRs and caches. In summary, BCoal scheme presented in this work not only offers improved security but also incurs minimal performance degradation as compared to RCoal.

4.4 Anatomy of Bucketing in GPUs

In this section, we first explain our general approach towards realizing a bucketing scheme and then explore the design challenges in meeting the bucketing requirements in the presence of MSHRs and caches. Finally, based on our analysis, we present our secure bucketing

scheme – BCoal.

4.4.1 Bucket Features

Let us assume a system with n buckets and sizes of buckets to be: $b_1, \dots, b_i, b_{i+1}, \dots, b_n$ where $\forall i : b_i < b_{i+1}$. A predetermined number of coalesced accesses are generated per table lookup (load) instruction as per the bucket size. If a load instruction generates n number of coalesced accesses, where $b_i < n \leq b_{i+1}$, then additional accesses are padded such that the total number of coalesced accesses is equal to b_{i+1} . The number of buckets is selected to achieve the desired reduction in the variance of the number of coalesced accesses. For example, with only one bucket, the number of accesses generated is always equal to the size of that bucket, thus, reducing the variance to zero. As the number of buckets increases, the variance in the number of coalesced accesses increases due to the increased number of distinct possible values for the coalesced accesses. This leads to higher information leakage, however, also reduces the total number of additional padded accesses.

We revisit Figure 4.6(a) to select the bucket features for AES. We observe that the number of coalesced accesses during the AES encryption on GPU never exceeds 16. Therefore, we select the size of the bucket to be 16 as one of the options and denote the scheme as BCoal(16). With only one bucket of size 16 in the coalescing unit, the AES encryption will always generate 16 number of coalesced accesses to reduce their variance to 0. Consequently, the correlation between the number of real coalesced accesses and the execution time drops as well. However, with only one bucket, each (security-sensitive and security-insensitive) load instruction sends 16 accesses, leading to performance degradation (Section 4.6).

The performance of BCoal scheme can be further improved by adding multiple buckets of intermediate sizes. We propose to add one more bucket with size 1 because of the bimodal distribution observed in Figure 4.6(a) and call this scheme BCoal(1, 16). The performance degradation in BCoal(1, 16) will be lower than in BCoal(16) because the coalesced accesses generated by instructions other than the table lookups (the first peak

in Figure 4.6(a)) now fit into the added bucket. Furthermore, in BCoal(1, 16), as the bucket with size 1 does not affect the table lookup instructions, its effect on the security is minimum. We quantify all performance and security results in Section 4.6.

4.4.2 Estimation of Number of Padded Accesses

To generate an optimal number of padded memory accesses for bucketing, we first need to determine the number of real memory accesses generated for a load instruction of a warp. The number of real coalesced accesses generated by the load instruction is stored as the pending request count (PRC) in the coalescing unit [61]. By reading PRC, we determine the number of real memory accesses. Next, we compare the number of real memory accesses generated with the preset bucket values. If the number of real memory accesses does not match, then we generate a number of padded memory accesses equal to the difference between the next larger bucket value and the number of real memory accesses. For example, in BCoal(1,16), if the number of original memory accesses is 12, then we need to generate 4 extra memory accesses.

4.4.3 Design Challenges in Generating Padded Accesses

We consider the effect of MSHRs and caches on RCoal scheme while designing the padding mechanism for BCoal. In RCoal, the redundant accesses to the same block addresses were merged in MSHRs eliminating the security offered by randomized coalescing of accesses. Therefore, to meet the bucketing requirement, we must generate padded accesses to the *unique* block addresses. Consequently, all memory accesses originating from a warp, real and padded, have *unique* block addresses. The unique accesses for padding are generated *randomly* from an address range that is accessible to the AES CUDA application. In our case, the block address range spans over the five tables used for table lookups and the round keys used for each round, all saved in the DRAM.

To evaluate the resulting bucketing scheme, we first determine the possibility of key byte recovery in the absence of MSHRs and caches. Figure 4.9a shows the scatter plots for

the bucketing scheme employing padding via unique accesses in the absence of MSHRs and caches. We note that the correct value of the key byte cannot be recovered as the attacker fails to establish a correlation between the real number of accesses and the execution time. The low correlation is attributed to the constant number of accesses generated per table lookup instruction across the plaintext samples leading to the low variance in them.

From the above padding mechanisms employed in the bucketing scheme, we make the following observation:

Observation I: For the secure bucketing scheme, the block addresses of the padded accesses should be random and unique (that is, exclusive of the block addresses of the real and other padded accesses of the corresponding table lookup instruction).

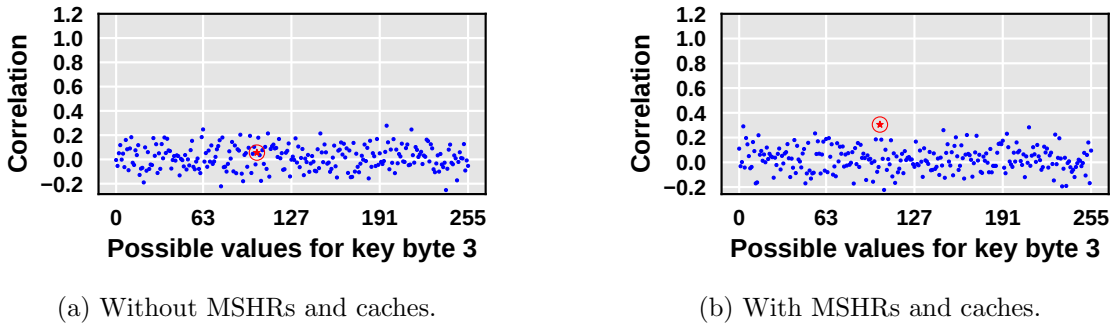


Figure 4.9: Evaluation of security offered by the bucketing scheme employing unique access padding mechanisms with one bucket of size 16. Plaintext has 32 lines. © 2020 IEEE.

Effect of MSHRs and caches. We evaluate the effect of MSHRs and caches on the security of above bucketing mechanism using the scatter plot in Figure 4.9b. We note that while the correlation and related key byte value leakage is low, the correct value of the key byte can still be recovered. The key byte value leakage is possible because the real and padded accesses affect the execution time at different rates due to their distinct merging and caching patterns.

The distinct access merging and caching patterns are caused because of the different block address ranges accessed by the real and padded accesses. In the above bucketing scheme, the padded accesses generated in each round access the same range of block ad-

addresses spread across the entire memory space of the AES CUDA application. In contrast to the padded accesses, in the last round, the real accesses target only the T_4 table elements. As the real accesses are confined to a narrower address space (only T_4 table elements) as compared to the padded accesses (entire application memory space), their respective merging and caching patterns are different. Therefore, the padded and real accesses affect the execution time at different rates. An attacker can then treat the effect of padded accesses on the execution time as noise and filter it out over a large number of plaintext samples to correlate the real accesses and the execution time to recover the private key. The effect of MSHRs and caches on the real and padded accesses leads to the following observation:

Observation II: The padded and real accesses should be homogeneous in terms of their respective probabilities of merging in MSHRs and caching.

4.4.4 BCoal: A Secure Bucketing Scheme

From the observations I and II recorded previously, we note that for a secure bucketing scheme to operate in the presence of MSHRs and caches, the padded accesses should have the following two characteristics: i) the block addresses of the padded accesses should be random and exclusive (unique) of the block addresses of the other accesses and ii) the padded accesses should follow the same merging and caching pattern as that of the real accesses.

Padding via Homogeneous Unique Accesses. The first property of the desired padding mechanism is met by ensuring that the block addresses of the padded accesses are random and unique across each security-sensitive load instruction. To enforce the second property, we recall the merging mechanism in MSHRs, where the accesses going to the same block addresses are merged together. Furthermore, the caching also works at the block address granularity. Therefore, to obtain similar merging and caching probabilities across all accesses, we restrict the block addresses of the padded accesses to the range of possible block addresses of the real accesses, thereby generating *homogeneous* unique

accesses²

During the AES execution, the table lookup instructions of the first nine rounds access first four tables, while for the last round only T_4 table is accessed. Therefore, to meet the bucketing requirements, the padding mechanism should restrict the block address range of the padded accesses to the block address range of the first four tables in DRAM during the first nine rounds, while to the block address range of T_4 table in DRAM during the last round. As the padding mechanism maintains similar merging and caching properties for the real and the padded accesses, the attacker cannot segregate their effects on the total execution time. Therefore, the attacker will fail to establish the correlation between the real number of coalesced accesses and the execution time, thereby failing to recover the key byte value. Furthermore, as all rounds of AES encryption are potentially vulnerable to timing attacks [100], BCoal is enabled for all ten rounds of AES.

In summary, we select the padding via homogeneous unique accesses for the BCoal bucketing scheme. We present the security and performance evaluation of the proposed BCoal scheme with MSHRs and caches enabled in Section 4.6.

4.5 Hardware/Software Overhead

In this section, we describe the implementation overhead of BCoal. We consider a generalized BCoal scheme, which targets a security-sensitive application with an arbitrary number of program sections. For example, the two program sections in AES are the first 9 rounds and the last round. The generated padded accesses have memory addresses that target respective program sections.

Storage overhead. The storage requirement is for keeping track of a) bucket sizes and b) the start/end addresses of the program sections. To store the buckets sizes, BCoal uses a 32-bit mask that covers all 32 possible number of coalesced accesses across a warp. The indices of the mask are set as per the BCoal configuration. For example, for BCoal(1,

²This heuristic may have to be tuned for different applications based on their memory access pattern.

16, 32), only 1st, 16th and 32nd bits are set. Next, BCoal maintains an address table – accessible by all SMs executing the security-sensitive application – to save the start and end 32-bit addresses of each program section. For an application with N program sections, the size of the table will be $(2N \times 32)$ bits. For AES with 2 program sections, the size of the table will be 128 bits and the total storage overhead is $128 + 32 = 160$ bits.

Address Generation. The generation of unique homogeneous accesses for padding follows three steps: a) determine the number of padded accesses needed, b) determine the unique homogeneous block addresses for the accesses, and c) generate the accesses. As noted in Section 4.4.2, the pending request count (PRC) in the memory coalescing unit (MCU) records the number of real accesses across a warp. Therefore, the number of padded accesses needed can be identified by comparing the size of a bucket with PRC. Since the maximum value of PRC (limited by the maximum possible number of coalesced accesses) and the maximum size of a bucket is 32, BCoal needs a 5-bit comparator.

The address range for each program section is known from the memory allocation and data copy operations executed at the start of a GPGPU application. This information can also be embedded in the load instructions. To generate padded accesses in the range of the program section under execution, BCoal uses a 32-bit random address generator.

4.6 Analysis of Security & Performance

In this section, we first analyze the security of our proposed bucketing-based coalescing mechanism, BCoal, via experimental and theoretical analysis. Subsequently, we discuss the effects of the proposed mechanism on performance and data movement. We also compare BCoal with RCoal in terms of security, performance and data movement. Finally, we generalize our mechanism across a wide range of GPGPU applications.

All the results are collected on a cycle-level GPU simulator – GPGPU-Sim [6]. We assume the same number of samples as that of in the attack scenario [49] for plaintext with 32 lines. For plaintext with 64 lines, we use 1000 samples, the same number as needed for

the successful attack, to evaluate the defense mechanism for a fair comparison.

4.6.1 Experimental Analysis of Security

For the security evaluation of BCoal scheme in the presence of MSHRs and caches, we consider two configurations: i) default with one bucket of size 16 denoted as BCoal(16) and ii) performance efficient with two buckets of sizes 1 and 16 denoted as BCoal(1, 16). For each BCoal configuration, we plot a scatter plot as explained in Section 4.2.4

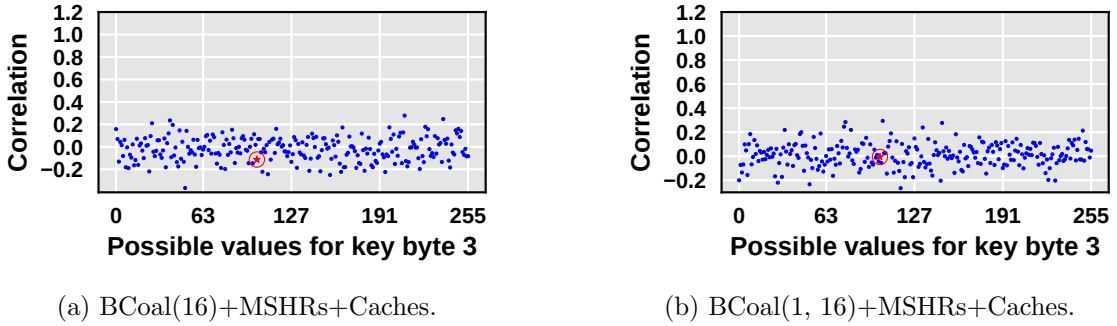


Figure 4.10: BCoal defense scheme against correlation attack for plaintext with 32 lines.
© 2020 IEEE.

Plaintext with 32 lines. Figure 4.10 shows the scatter plots for BCoal scheme using plaintext with 32 lines with MSHRs and caches enabled. We note that the key byte recovery is not possible because of the low correlation between the number of accesses and the execution time. The low correlation can be explained as follows. With BCoal scheme, three scenarios can occur for a table lookup instruction. First, all accesses – real and padded – for the instruction are cached. In this case, the instruction always generates 16 accesses to the cache. Second, no accesses are cached, therefore, generating 16 DRAM accesses to the unique block addresses which MSHRs cannot optimize. In both scenarios, the number of accesses to the cache or DRAM remains constant leading to reduced correlation with the execution time. In the third case, a partial set of accesses of the instruction are either cached or merged in MSHR. Here, since the real and the padded accesses target the same block address range, their merging and caching probabilities are

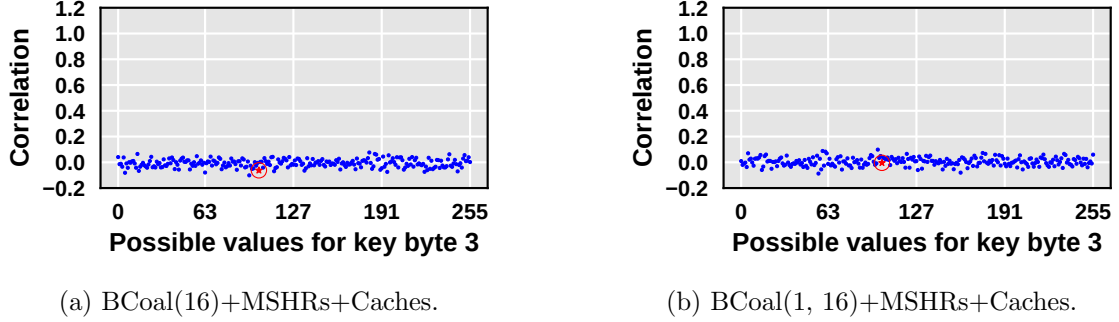


Figure 4.11: BCoal defense scheme against correlation attack for plaintext with 64 lines.
© 2020 IEEE.

similar. Subsequently, the attacker cannot distinguish between the effects of the padded and real accesses on the execution time and fails to correlate the number of real coalesced accesses and the execution time. In conclusion, the attacker fails to recover the key byte in a BCoal-enabled GPU.

Plaintext with 64 lines. Figure 4.11 shows the scatter plots for BCoal scheme using plaintext with 64 lines with MSHRs and caches enabled. We note that the key byte recovery is not possible because of the low correlation between the number of accesses and the execution time. To understand the low correlation, we refer to the correlation timing attack described by Jiang et al. in [44] for the multi-warp case, where the attacker treats each warp individually executing a plaintext with 32 lines and chooses the warp with the highest number of coalesced accesses to recover the key. Therefore, the observations made for a single warp case hold true for the multi-warp case as well. Particularly, the attacker cannot correlate the number of real coalesced accesses and the execution time due to the low variance in the number of accesses, and the homogeneity between the real and padded accesses. Therefore, the attacker fails to recover the key byte value in the multi-warp scenario.

The experimental analysis concludes that BCoal-enabled GPU successfully mitigates the correlation timing attacks in single-warp and multi-warp scenarios.

4.6.2 Theoretical Analysis of Security

We present an analytical framework to analyze the security of AES. Before a formal analysis, we consider one instruction in the last round that accesses 12 unique memory block addresses before padding. When only one bucket is used (at 16), the 4 padded memory accesses are drawn from the same memory space as the 12 real requests. Hence, there is no information leakage. In general, we will shortly prove that when BCoal uses one bucket at 16, there is no information leakage.

When multiple buckets are used, say at 12 and 16, the attacker can infer if the number of real block addresses being accessed are up to 12 or between 12 and 16, which leaks some information. However, as we show next, the leakage in general is minimal, due to the randomized mapping from plaintext lines to warps. The randomized mapping obfuscates which plaintext lines share the same warp.

To quantify the leakage of BCoal, we note that threads across different warps are not synchronized and the longest warp execution time dominates the time measurement [44]. Hence, one of the warps, the dominant warp, will have true timing. Known attacks on multiple warps [44] analyze each warp and use the longest running (dominant) warp for correlation analysis to recover the AES private keys. So it is safe to focus on an *arbitrary* warp in the rest of the analysis. Moreover, we assume the padded and real accesses are homogeneous (as described in Section 4.4.2). Hence, their probabilities of merging in MSHRs and caching are identical.

To make a fair comparison with RCoal, we follow the analytical model and assumptions of RCoal [49]. Further, we target an arbitrary last-round key byte k and assume that U is the number of real accesses for the lookup of last round table, T_4 , with respect to the key byte k , from the dominant warp. Following RCoal [49], we estimate the number of plaintext samples required to successfully recover an AES key byte, S , as

$$S \propto \left(\frac{\mu(U \times \hat{U}) - \mu(U)\mu(\hat{U})}{\sigma(U)\sigma(\hat{U})} \right)^{-2} \quad (4.1)$$

where \hat{U} is the number of coalesced accesses when the guessed key byte is identical to k , μ and σ are the mean and standard deviation of a random variable respectively.

We first prove BCoal leaks no information with one bucket.

Lemma 1 *When BCoal only uses one bucket at 16, the needed samples to break AES is infinite.*

Proof: With only one bucket, $P(\hat{U} = 16|U = u) = 1$ for any u . Hence, $\mu(\hat{U}) = 16$ and $\mu(U \times \hat{U}) = \sum_u P(u)\mu(U \times \hat{U}|U = u) = 16 \sum_u u \times P(u) = 16\mu(U)$. Hence $S = (0)^{-2} = \infty$.

■

When the number of buckets is more than one, the computation is more involved. To simplify the analysis, we further make a conservative assumption that an attacker may directly observe the unpadded memory blocks in the following analysis. Therefore, $\mu(\hat{U}) = \mu(U)$, $\sigma(\hat{U}) = \sigma(U)$.

In AES, the lookup table relevant to key byte k has 16 unique memory block addresses. With sufficiently random plaintexts and a warp with 32 threads, each thread accesses one of 16 memory block addresses in a uniform way. Hence, the number of unique block addresses U , obeys the following distribution: $P(U = i) = \frac{1}{16^{32}} \frac{16!}{(16-i)!} \{^32_i\}$, where $\{^32_i\}$ denotes the Stirling number of the second kind. Here, $\{^32_i\}$ represents the ways of partitioning 32 threads into i non-empty subsets; $\frac{16!}{(16-i)!}$, i -permutations of 16, represents the ways of forming i non-empty subsets from 16 memory block addresses. From this distribution, we can compute both $\mu(U)$ and $\sigma(U)$ by their definitions.

To compute $\mu(U \times \hat{U})$, we note that due to the random mapping from plaintext lines to warps, U and \hat{U} only depend on the frequency of accessing the 16 memory block addresses among the 64 lines of plaintext, which is defined as follows.

Definition 4 *For 16 memory blocks and 64 plaintext lines, the frequency set of all possible accesses to the block addresses are*

$$\mathcal{F} = \{(f_1, \dots, f_{16}) \mid f_1 + \dots + f_{16} = 64\}$$

Table 4.2: Security Analysis. S denotes the normalized number of samples required to successfully recover an AES key byte [49]. © 2020 IEEE.

Schemes	Correlation ρ	(normalized) S
RCoal(4)	0.15	42×
RCoal(32)	0.00	∞
BCoal(16)	0.00	∞
BCoal(1,16)	0.16	37×

where $f_i \in \mathcal{F}$ represents the frequency of accessing the i -th memory block address among the 64 plaintext lines.

Given $F \in \mathcal{F}$, $\mu(U|F) = \sum_{f_i \in F} \mu(\mathbf{1}_{\text{block } i \text{ is accessed}} | f_i)$, where $\mathbf{1}_{\text{block } i \text{ is accessed}}$ is an indicator random variable that has value 1 if block address i is being accessed in the dominating warp. Given f_i accesses to block address i , the probability that it is accessed in the dominating warp is $(1 - C_{f_i}^{64-32}/C_{f_i}^{64})$, where C_n^m denotes the binomial coefficient. Hence,

$$\mu(U|F) = \sum_{f_i \in F} 1 - C_{f_i}^{32}/C_{f_i}^{64}$$

Given $F \in \mathcal{F}$, U and \hat{U} are independently and identically distributed. Hence,

$$\mu(U \times \hat{U}) = \sum_{F \in \mathcal{F}} P(F) \mu(U|F)^2 = \sum_{F \in \mathcal{F}} P(F) \left(\sum_{f_i \in F} 1 - \frac{C_{f_i}^{32}}{C_{f_i}^{64}} \right)^2$$

Here, $P(F)$ is the probability of seeing the frequency vector F . Among all 16^{64} combinations of memory accesses from 64 threads, $C_{f_1}^{64} C_{f_2}^{64-f_1} \dots C_{f_{16}}^{64-\sum_{1 \leq j \leq 15} f_j} = \frac{(64)!}{\prod_{f_i \in \mathcal{F}} f_i!}$ match F . Hence, we have $P(F) = \frac{(64)!}{\prod_{f_i \in \mathcal{F}} f_i!} \times \frac{1}{16^{64}}$.

Putting all pieces together, we use a Python script to compute the correlation and normalized the sample size needed for a successful attack, similar to the RCoal analysis [49]. The results are summarized in Table 4.2. We note that with 1 bucket, BCoal rules out leakage entirely. With multiple warps, its security is comparable with RCoal(4), the best of the RCoal schemes. Note that the results of RCoal in Table 4.2 only applies when MSHR and caches are *disabled*. But with homogeneous padded and real accesses, the results of BCoal also applies even if MSHR and caches are *enabled*.

In summary, this theoretical security analysis demonstrates that when MSHR and caches are disabled, both RCoal and BCoal schemes provide significant security against the correlation timing attack. However, if the MSHRs and caches are enabled, RCoal becomes vulnerable due to the access merging and caching as illustrated in Figure 4.8. In contrast to RCoal, BCoal has high security even in the presence of MSHRs and caches as shown both in Table 4.2 and in Section 4.6.1.

4.6.3 Experimental Analysis of Performance

To evaluate the performance and scalability of BCoal scheme against RCoal, we plot the execution time and number of DRAM accesses in Figure 4.12 for plaintext with 32, 64 and 1024 lines. We first demonstrate the effect of different coalescing strategies in BCoal and RCoal by comparing them in the absence of MSHRs and caches in Figure 4.12a. We note that the number of DRAM accesses increases sharply with the plaintext size in RCoal as compared to BCoal due to the inefficient access coalescing in RCoal. Consequently, RCoal suffers severe performance degradation as compared to BCoal as the plaintext size increases.

Figure 4.12b demonstrates the effect of MSHRs and caches on the performance of BCoal and RCoal. Both schemes show a significant reduction in the DRAM traffic leading to reduced performance degradation. However, in the presence of MSHRs and caches, RCoal is insecure (Section 4.3.2) and BCoal is secure (Section 4.6.1 and 4.6.2). For BCoal, the performance degradation is limited to 5% and 15% for BCoal(1, 16) and BCoal(16), respectively. In summary, the performance of BCoal (with MSHRs and caches) scales well with the plaintext size as opposed to secure RCoal (without MSHRs and caches).

4.6.4 Evaluating BCoal on Other Applications

We evaluate BCoal on a wide range of applications from various suites such as CUDA-SDK (C) [97], Rodinia (R) [13], Lonestar (L) [10], Mars (M) [35], Shoc (S) [18] and Polybench (P) [108]. For these applications, we evaluate only the performance of BCoal, as the

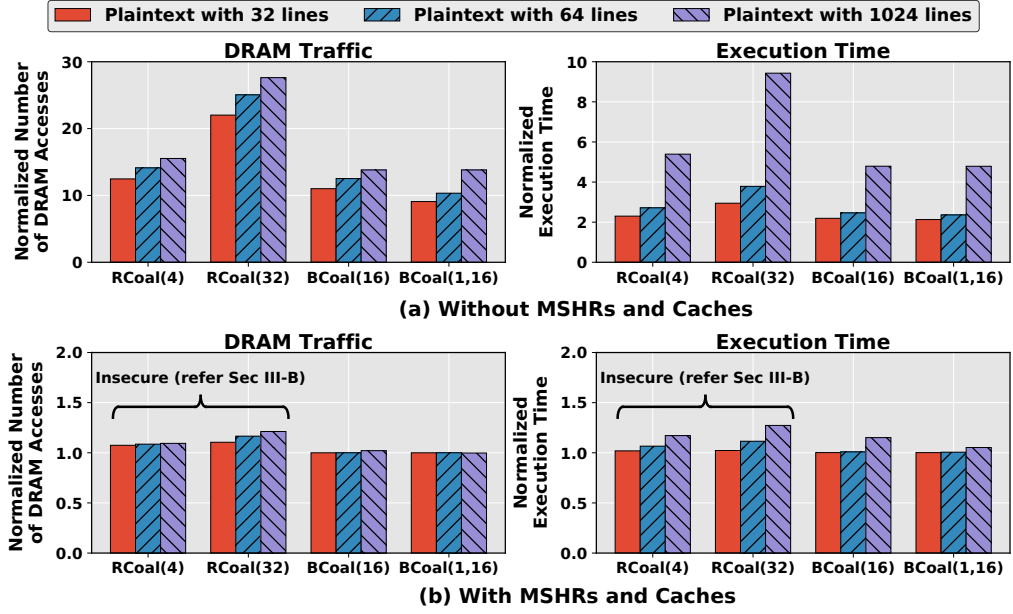


Figure 4.12: Performance of BCoal for different plaintext sizes. All results are normalized to the baseline GPU. © 2020 IEEE.

bucketing driven reduced variation in the number of coalesced accesses ensures improved security. The address range of the padded accesses is spread over the entire memory space of the respective application. We examine the effects of the number and sizes of buckets on the application performance using Figure 4.13. The MSHRs and caches are enabled for the evaluation.

Number of buckets. In Figure 4.13, the first two configurations of BCoal, BCoal(1, 16, 32) and BCoal(1, 32), demonstrate the effect of the number of buckets on various applications. Both configurations have a bucket of size 1 to reduce the DRAM traffic in applications that exhibit perfect coalescing (i.e., all threads in a warp are served by a single cache block at a given time). We notice that most applications are unaffected by the number of buckets, as they can leverage the bucket of size 1 through good coalescing profiles.

In C-CONS and C-NN, the number of DRAM accesses increase in BCoal(1, 32) as the number of coalesced accesses between 2 to 31 are padded to meet the bucket 32. The in-

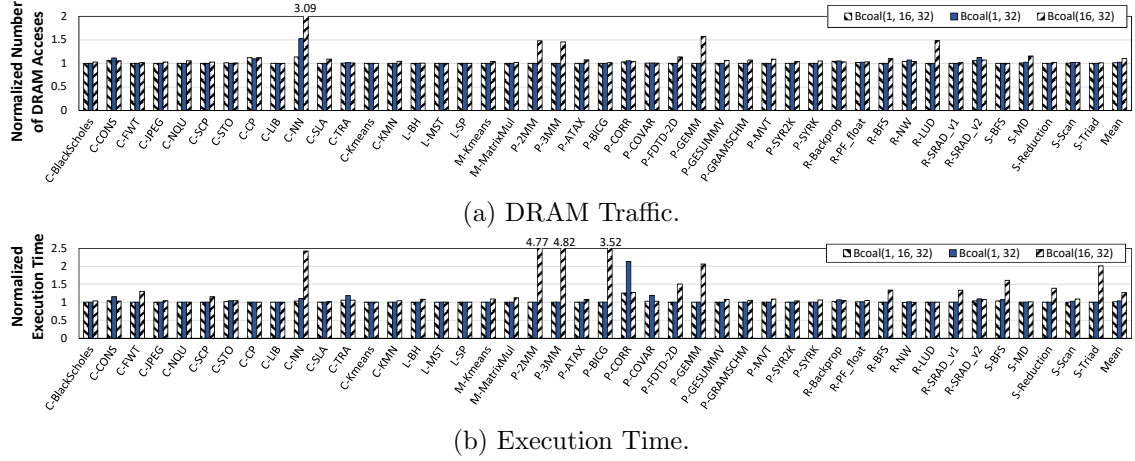


Figure 4.13: Performance Evaluation of BCoal on GPGPU applications with MSHRs/caches enabled. Results are normalized to the baseline GPU. © 2020 IEEE.

creased number of accesses in combination with high cache-misses results in increased DRAM traffic leading to increased performance degradation. In C-TRA, P-CORR and P-COVAR, although the number of DRAM accesses does not change drastically, the execution time increased in BCoal(1, 32) over BCoal(1, 16, 32). The increase in execution time is attributed to the increase in the number of L1 cache accesses in BCoal(1, 32) as it lacks the bucket of size 16. The increased L1 accesses, even if cached (thus leading to fewer DRAM accesses), are satisfied serially thereby increasing the execution time.

Sizes of buckets. BCoal(1, 32) and BCoal(16, 32) demonstrate the effect of bucket sizes on various applications. We noticed that the performance degradation is severe for BCoal(16, 32) compared to BCoal(1, 32) due to the increased number of DRAM accesses in BCoal(16, 32). In BCoal(16, 32), the smallest bucket size is 16, therefore all applications, even the ones with good coalescing profiles, generate at least 16 DRAM accesses for each memory access instruction. Subsequently, the number of DRAM accesses increase resulting in increased performance degradation.

In summary, we observe that the application performance is more affected by the sizes of buckets than the number of buckets. A careful bucket size selection can reduce the number of padded requests thereby reducing the overall data movement.

A Generic BCoal configuration. From Figure 4.13, we note that BCoal(1, 16, 32)

configuration results in only 1.15% average performance loss. The security and performance of AES with BCoal(1, 16, 32) is identical to BCoal(1, 16) because the bucket of size 32 in BCoal(1, 16, 32) is never used as the baseline number of coalesced accesses never exceed 16 as shown in Figure 4.6a. Therefore, BCoal(1, 16, 32) can be widely adopted as it offers good security at a minimal performance loss. However, for optimal security and performance tradeoff, a user can perform application-specific offline profiling of coalesced accesses (discussed in Section 4.3) to determine appropriate bucket features.

4.7 Related Work

In this section, we highlight the prior works that are the most relevant to this paper.

Attacks. Implementations of cryptographic systems on CPUs are vulnerable to timing attacks. Several AES implementations contain key-dependent memory accesses, which eventually affect the status of the data cache. Via cache-probing technique, an attacker can quickly recover the entire private key of AES and RSA by measuring the execution time of either a cryptographic algorithm (e.g., [7, 100, 9, 8, 30]) or his/her own application if the data or instruction cache is shared (e.g., [30, 142, 40, 145]). On GPUs, Jiang et al. [44] demonstrated a novel complete AES key recovery timing attack that exploits the coalescing features on a commercial GPU architecture (discussed in Section 4.2.4). They also developed a new fine-grained timing channel caused by shared memory bank conflicts in GPUs [45]. Wang et al. [130] developed partial attacks against RCoal [49] focusing on the configurations with high variance in the number of coalesced accesses. Our BCoal mechanism further reduces the variance making it a much stronger defense.

Defense mechanisms. Several hardware-based defense mechanisms have been proposed in the context of CPUs [102, 67, 144, 132, 133, 70, 136]. However, those mechanisms have been shown to work only for cache-based timing attacks and not for GPU coalescing-related vulnerabilities. The memory traffic shaping schemes to mitigate the timing attacks in CPUs have been extensively explored [146, 26, 4]. With the help of fake/dummy access

generation mechanism, these schemes enforce the memory traffic to follow either a constant rate or a pre-determined distribution over a time epoch. These schemes differ from BCoal in two ways. First, BCoal works at a finer instruction-level granularity to shape the memory traffic. The single-instruction multiple-thread (SIMT) execution model of GPUs allows parallel thread memory access generation across a warp, which is leveraged by BCoal to estimate and generate padded accesses for each sensitive instruction. Second, BCoal ensures that the real and padded accesses are to the same memory space, which helps in making their individual effects on execution time similar. This makes it harder for the attacker to distinguish padded accesses from the real accesses.

Lin et al. [68] proposed new software-based mechanisms specific to AES for reducing the information leakage due to coalescing units. On the other hand, BCoal is a generic hardware-based coalescing mechanism applicable to all security-sensitive GPGPU applications that are vulnerable to coalescing-based correlation timing attacks. This also makes BCoal complementary to other software-based implementations of cryptographic workloads. Köpf et al. [56] ensures that the execution time matches one of the discrete bucket values, while BCoal ensures the number of memory accesses generated per load instruction conform to a predefined set of values, that is buckets. Further, buckets in the prior work [56] assumes input blinding for a tight leakage bound. In BCoal, we utilize the inherent parallelism in GPUs to randomize the mapping from inputs to threads, achieving a similar blinding effect for arbitrary applications.

4.8 Conclusions

We propose a bucketing-based coalescing scheme (BCoal) to thwart the coalescing-based correlation timing attack without incurring high performance overhead. The key insight is to redesign GPU memory coalescing such that it always issues a pre-determined number of memory accesses (called buckets). Our modified coalescing unit generates additional memory accesses (if necessary) along with the real accesses to match the bucket require-

ments. These additional padded accesses reduce the variance in the total number of coalesced accesses to significantly enhance the security. BCoal carefully generates padded accesses such that they have similar caching/merging probability as that of the real accesses. Such a mechanism significantly helps in retaining the security even in the presence of the MSHRs and caches. In conclusion, we believe that BCoal addresses the memory coalescing related vulnerability in GPUs while incurring low performance overhead.

Chapter 5

Data-centric Reliability Management in GPUs

5.1 Introduction

Graphics Processing Units (GPUs) have become an inevitable part of every computing system due to their ability to provide large improvements in performance and energy efficiency compared to CPUs [60, 5, 69, 118, 59, 61, 106, 3, 2, 12, 54, 122]. Consequently, they have become the default choice for accelerating innovations in various fields such as high-performance computing (HPC), artificial intelligence (AI), and even reliability-critical autonomous vehicle software [23, 109, 122, 93, 116, 98, 91, 96, 103]. The emerging computing needs of these domains have fueled the growth of GPU architectures. Especially, the growing focus on deep learning has increased GPU demands tremendously. Almost every year AMD and NVIDIA unveil new GPU designs that incorporate significant innovations to their GPUs leading to improved performance and energy efficiency. For example, the latest Ampere architecture [99] has an L2 cache size that is 10x larger comparing to previous generations and new high bandwidth memories are being incorporated into almost all new GPUs.

The effect of the above innovations on GPU reliability is not yet well-understood.

For example, advanced DRAM architectures make single-bit and multi-bit faults more common [120, 125, 85, 86, 87]. Similarly, low voltage cache design proposals (i.e., AMD Killi [27] or IBM Dante [11]) for managing power consumption of large last-level caches in GPUs [27, 99] can cause an increased number of multi-bit faults. These multi-bit faults can lead to catastrophic failures, such as accidents of autonomous vehicles [105, 16, 65, 43]. Unfortunately, the existing ECC mechanisms cannot correct multi-bit faults. SECDED is only capable of detecting up to two-bit faults and of correcting one-bit fault only. Other mechanisms such as ChipKill [19] are currently not feasible in GPUs [58]. Popular methods such as check-pointing [90, 29, 58] come with significant overhead costs due to the large amounts of data the GPGPU applications typically process [47]. Similarly, redundant computation techniques, if not carefully performed, can lead to significant overheads in terms of both performance and energy [21, 129, 31, 75, 137].

In order to provide low-overhead reliability in GPUs, especially in the context of multi-bit faults, we take a data-centric approach. Based on our extensive application-level analysis, we find that for a large number of applications, only a limited amount of data needs additional reliability protection compared to the baseline SECDED. Such data constitutes a small fraction of the entire application memory, is read-only, and is highly accessed and shared across the majority of concurrently executing warps. We show that if this data is subject to multi-bit faults, it can lead to incorrect application output (e.g., high mis-classifications errors in the case of neural networks) as the faulty data is accessed by multiple thread instructions across the majority of warps. Interestingly, we observe that this critical portion of the data can be profiled and this information can then be passed on to hardware for developing low-overhead correction and detection mechanisms.

To the best of our knowledge, this is the first work that takes a data-centric approach towards improving GPU reliability while incurring low overhead. In summary, this paper makes the following contributions:

- We perform detailed application-level analysis to show that a small fraction of critical data (hot memory blocks) used by a large number of GPGPU application threads can

dramatically increase thread vulnerability to multi-bit faults. This data is usually read-only and can be profiled with low-overhead.

- We develop both detection and correction schemes for application resilience that prioritize reliability fortification of this identified critical data. Our resilience schemes leverage data information obtained from the application source code and access pattern for replicating *only* the hot memory blocks.

- Our reliability management schemes exhibit very limited overhead due to the small fraction of data that gets replicated and to the fact that the performance overhead of additional checks (and associated memory accesses) is largely hidden thanks to the latency tolerance property of GPUs.

Quantitatively, our resilience schemes significantly improve GPU reliability by dropping the number of silent data corruption (SDC) outcomes in the application runs by 98.97% on average, while incurring a low average performance overhead of 1.2% for detection and 3.4% for detection-and-correction scheme.

5.2 Background

In this section, we present a brief overview of the baseline GPU architecture and the sources of faults in the caches and memory. Finally, we describe the fault injection model and the error metrics for each application used throughout the paper.

5.2.1 Baseline GPU Architecture

Figure [5.1](#) shows a generic GPU architecture. It is composed of a set of cores, known as streaming multiprocessors (SMs) in NVIDIA terminology. Each SM consists of an array of processing elements (PEs) and several load/store (LD/ST) units. Furthermore, each SM is associated with an L1 cache shared across the PEs. Next, all SMs on the GPU share multiple L2 cache banks which are connected through an interconnection network. Each L2 cache bank is connected to a separate memory channel. Finally, all SMs are supported

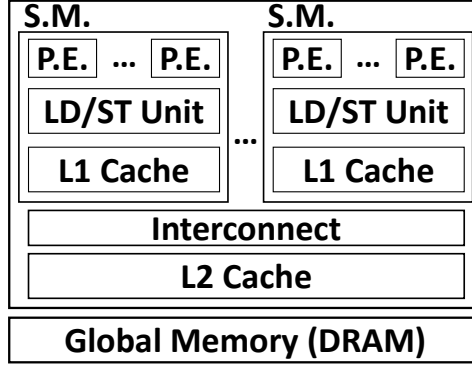


Figure 5.1: A Schematic of the Baseline GPU Architecture. © 2021 IEEE.

by high-bandwidth off-chip global memory (DRAM). Throughout the paper, we evaluate the proposed techniques on a cycle-level GPU simulator – GPGPU-Sim [6]. Note that we assume that caches and memory are already protected by SECDED ECC and hence we focus only on the effect of multi-bit errors on application output. Table 5.1 provides more details on the simulated architecture.

Table 5.1: Key configuration parameters of the simulated GPU. © 2021 IEEE.

Core Features	1400MHz core clock, SIMT width = 32 (16×2)
Resources / Core	32KB shared memory, 32KB register file, 15 SMs
L1 Caches / Core	16KB 4-way L1 data cache, 2KB 4-way I-cache 128B cache block size
L2 Caches	16-way 256 KB/memory channel (1536 KB in total), 128B cache block size
Memory Model	6 GDDR5 Memory Controllers, FR-FCFS scheduling 16 DRAM-banks, 924 MHz memory clock
Interconnect	1400MHz interconnect clock

Program Execution Model. A CUDA program consists of multiple functions known as kernels. A kernel is launched across many threads, where each thread is responsible for a set of instructions to be processed on the PEs. The threads are organized in groups, known as *Co-operative* Thread Arrays (CTAs). The number of CTAs and their size (i.e., the number of threads per CTA) are configured by the programmer at the kernel launch time. Each CTA is assigned to one SM. The number of CTAs assigned per SM is governed by the resources available per SM. Threads of a CTA are executed on the PEs at a granularity of warps, where each warp is usually a group of 32 threads. Within a warp,

all 32 threads are processed in lockstep, executing the same instructions on different data.

5.2.2 Data Memory Faults in GPUs

Hardware faults arise due to particle strikes, temperature/voltage fluctuations, or process variations [79]. Prior work has shown that GPUs are susceptible to a variety of faults [77, 125, 85, 86, 120, 121, 27]. In this work, we focus on faults occurring in the GPU memory hierarchy. Single-bit and multi-bit faults in the storage cells or the read logic of the SRAM (cache) and DRAM may cause errors in the stored data [20, 14, 53, 52]. Consequently, the application may read erroneous data resulting in silent data corruption (SDC) in its output.

The impact of the data memory faults on the application output depends on the application usage. For example, a memory fault in the GPU while executing a convolution neural network (CNN) can result in image mis-classification. If the CNN is employed in self-driving automobiles, then such mis-classification can cause catastrophic results, including loss of lives. We provide more details on the effects of faults on applications in Section 5.3.3.

GPUs use (SECDED)-based error checking and correction (ECC) codes to address the faults in GPU caches and memory [58]. SECDED ECC detects and corrects single-bit faults, and detects double-bit faults in the application memory. However, the growing multi-bit faults are harder and expensive to detect and correct. This is the focus of this paper.

5.2.3 Fault Injection Setup

5.2.3.1 Fault Model

To emulate data memory errors caused by faults in caches and DRAM, we follow the error emulation framework described by Luo et al. [74]. To this end, we inject faults in

the data memory blocks allocated by the application address space, irrespective of how they are mapped in the caches and DRAM. To clearly show the impact of increasing bit faults, we run two sets of fault injection experiments: first, we inject faults only in a single memory block. Second, we inject faults in 5 different memory blocks. For brevity, additional options are not shown.

1 data memory block: We select one 128B data memory block from the application address space. The memory block selection is determined by the objective of the fault injection experiment (refer to Sections 5.3.3 and 5.5.2 for details).

5 data memory blocks: Here, we select 5 128B data memory blocks from the application address space. As in the 1 memory block case, the block selection depends on the objective of the fault injection experiment.

Within the selected memory block(s), we randomly target a word to inject faults. The injected faults are modeled as *permanent* and *stuck-at* faults. Furthermore, for stuck-at faults, we assumed that a faulty bit is stuck at either a logical 0 or 1 with equal probability. To study the effect of multi-bit faults on the application output, we inject either 2-bit, 3-bit, or 4-bit faults at random bit locations within the target word. For each setting, we execute 1000 runs to achieve statistically significant results [62, 33, 88].

5.2.3.2 Error Metric Selection

The faults in the data memory blocks may go undetected by SECDED-ECC in GPUs and cause an incorrect application output. This is a case of silent data corruption (SDC). To identify whether a fault-injected application run results in an SDC output, we adopt metrics tailored to each application. For the applications from the Polybench suite, the output is either a single- or multi-dimensional vector. To determine whether the output is an SDC, we note how many vector elements deviate from the fault-free baseline output vector. Applications from the Axbench suite generate images as output. Therefore, we compare the output image from a fault-injected run with the output image from the fault-free baseline run. Table 5.2 details the error metric selected for each application. For each

Table 5.2: Output Error Metrics for Applications. © 2021 IEEE.

Application	Output Format	Error Metric
C-NN	Vector Classifications	Percentage of Mis-classifications in output.
P-BICG	Result Vector	Percentage of output vector elements with different values than the baseline.
P-GESUMMV	Result Vector	
P-MVT	Result Vector	
A-Laplacian	Filtered Image	Normalized Root Mean Square Error compared to the baseline image.
A-Meanfilter	Filtered Image	
A-Sobel	Edge Detected Image	
A-SRAD	Image	

application, we set a threshold value (either directly provided by the benchmark suite or reasonably set based on the application behavior) to determine output quality, that is, whether the application run resulted in an *SDC outcome*.

5.3 Motivation and Workload Analysis

In this section, we first highlight the problem of increasing memory faults in GPUs. Next, we analyze the application memory access pattern and illustrate that a small fraction of data memory (hot memory) in GPGPU applications is highly accessed and shared across multiple warps. Finally, we demonstrate the vulnerability of GPGPU applications to faults in hot memory.

5.3.1 Problem Definition and Goals of This Work

Current Trends. Innovations in GPU memory systems lead to tremendous growth in performance and energy efficiency. For example, the on-chip GPU cache sizes are consistently increasing across GPU generations to accommodate increasing working data sets, see Figure 5.2. From the DRAM perspective, memory bandwidth and capacity are growing consistently. Advanced high-bandwidth memories (HBM) in GPUs now have up to sustained bandwidth of 1-2 TB/sec with capacities of 16-32GB [99].

Unfortunately, the effect of memory innovations on GPU reliability is not well understood. Recent efforts are directed towards reducing on-chip power mainly by operating the caches at near-threshold voltage [28, 27] but such reduction leads to a significant in-

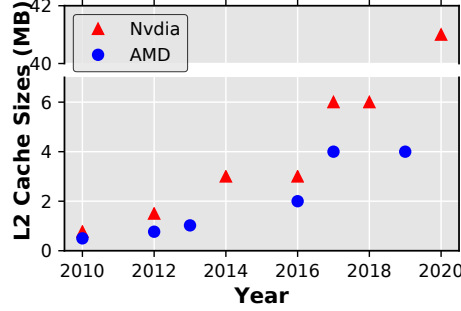


Figure 5.2: L2 Cache size trends for NVIDIA and AMD GPUs. © 2021 IEEE.

crease in multi-bit faults [28]. Previous studies have demonstrated that SECDED ECC is not sufficient to mitigate faults in DRAM [77, 120, 121]. Sridharan et al. [120, 121] showed through a field study that DRAM failures are dominated by permanent faults rather than transient faults and result in faulty data. Another field study by Martino et al. [77] compares CPU and GPU error rates and demonstrates that GPUs are three-orders of magnitude more susceptible to errors than CPUs.

Our goals. In this work, we aim to devise performance-efficient resilience schemes to address the multi-bit faults in L2-caches and DRAM. Since GPUs operate on large amounts of data in parallel, addressing the faults in the entire memory space incurs high performance and storage overhead [21]. Aiming to minimize this overhead, we propose a selective memory protection technique that is based on the observation that protecting *only a small fraction* of the data memory against multi-bit faults is sufficient to provide high reliability. To illustrate the above, we first analyze the memory access pattern of applications to identify if there is a fraction of memory with a high number of accesses comparing to the rest of the memory blocks. We show that this data is highly accessed and shared across multiple warps. We term the memory blocks of this highly accessed and shared fraction of the memory *hot memory blocks*. Next, we show that faults in hot memory blocks can result in silent data corruption (SDC) of the application output. Finally, we develop mechanisms to identify the hot memory blocks. Our analysis of several application codes shows that the hot memory blocks are usually read-only, constitute a very small fraction of the

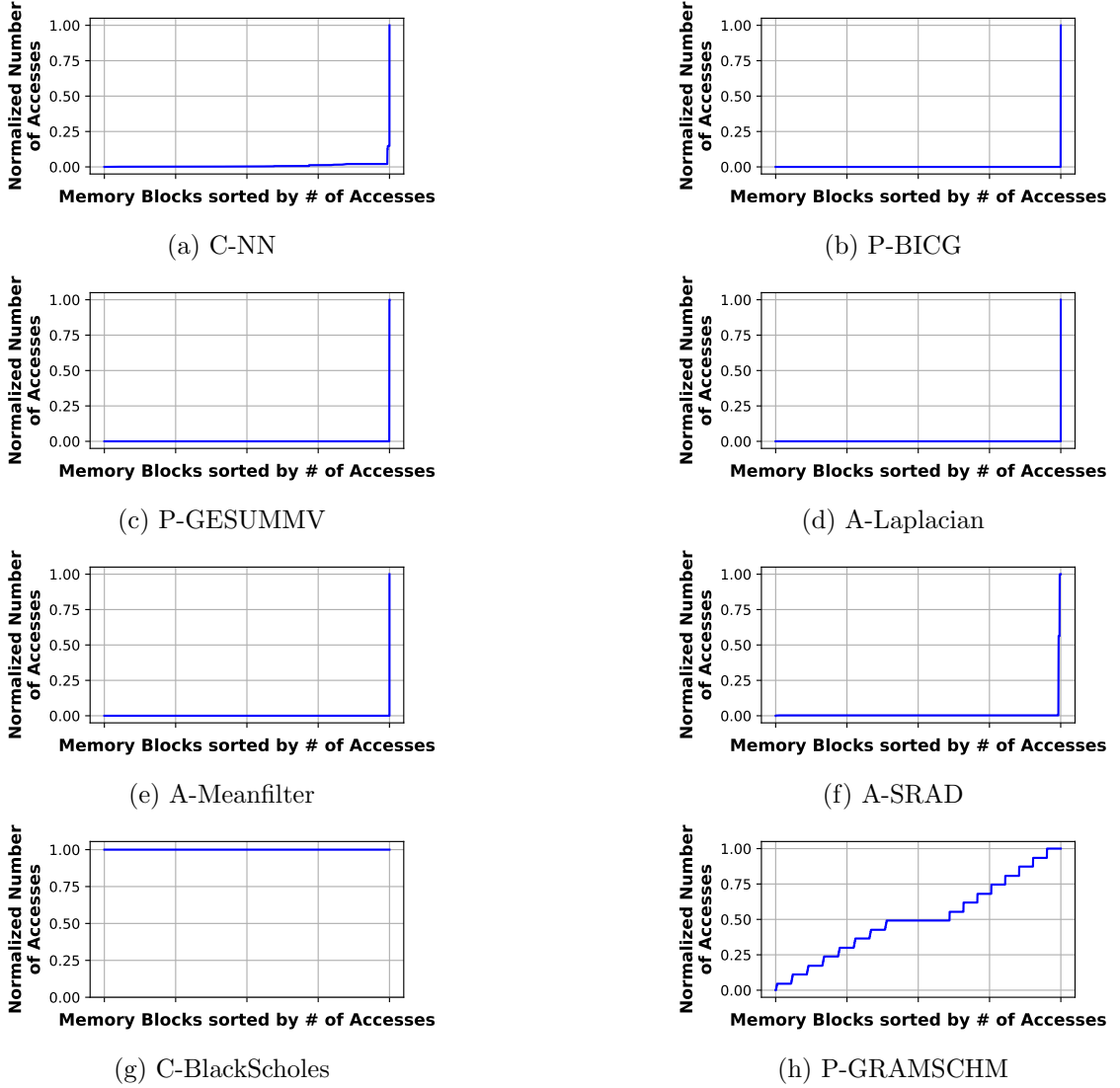


Figure 5.3: Normalized number of accesses to data memory blocks. For (a)-(f), we note that very few memory blocks experience a very high number of RD accesses compared to other blocks. © 2021 IEEE.

total application memory, and can be identified quickly. Using these insights, we propose two selective memory protection mechanisms, where we duplicate/triplicate the hot memory blocks to achieve low-overhead detection/correction schemes. We also show how reliability-performance trade-offs can be achieved by adjusting the amount of replication.

5.3.2 Application Access Pattern Analysis

Application Selection. To evaluate the impact of the proposed resilience schemes, we focus on applications with a clear and quantifiable output. Applications are drawn from popular benchmark suites including CUDA-SDK [97], Polybench [108], and Axbench [143]. We also ensure that the selected applications show variability in terms of memory access patterns.

Application Classification based on Access Pattern. We examine the read (RD) accesses to the data memory blocks from the application address space. We focus on the RD accesses as they are the most dominant ones in the memory access pattern. We analyze the access count to each memory block of the application under observation. In Figure 5.3, we show example plots for 8 applications, where the RD access counts to each memory block are sorted from low to high. Based on the access patterns in Figure 5.3, we split the applications into two primary categories. First, for applications in Figure 5.3(a)-(f), we note that *few memory blocks account for a high number of RD accesses*. Specifically, for C-NN, the memory block with the highest number of RD accesses has 4732-times more RD accesses than the memory block with the least number of RD accesses. On the other hand, for applications in Figure 5.3(g)-(h), we note that no memory blocks have high RD accesses compared to the rest of the memory blocks. For example, for C-BlackScholes, the numbers of accesses across different memory blocks are equal. Lastly, for P-GRAMSCHM, the number of accesses increases in small steps, and therefore, there are no memory blocks with a disproportionately high number of RD accesses.

Here, we focus on applications demonstrating access profiles similar to those shown in Figure 5.3(a)-(f), where a small number of memory blocks accounts for a very high number of RD accesses compared to the rest of the memory blocks. Table 5.2 lists the selected applications.

Observation I: For several GPGPU applications, a small number of data memory blocks incurs a very high number of read (RD) accesses as compared to the rest of the memory blocks.

Warp-level Spread of Highly Accessed Data. Next, we profile the RD accesses of applications listed in Table 5.2 to see if the highly accessed data memory blocks are *always* shared across multiple warps. To this end, we plot the number of warps accessing the data memory blocks, with memory blocks sorted by the total number of RD accesses from low to high, see Figure 5.4.

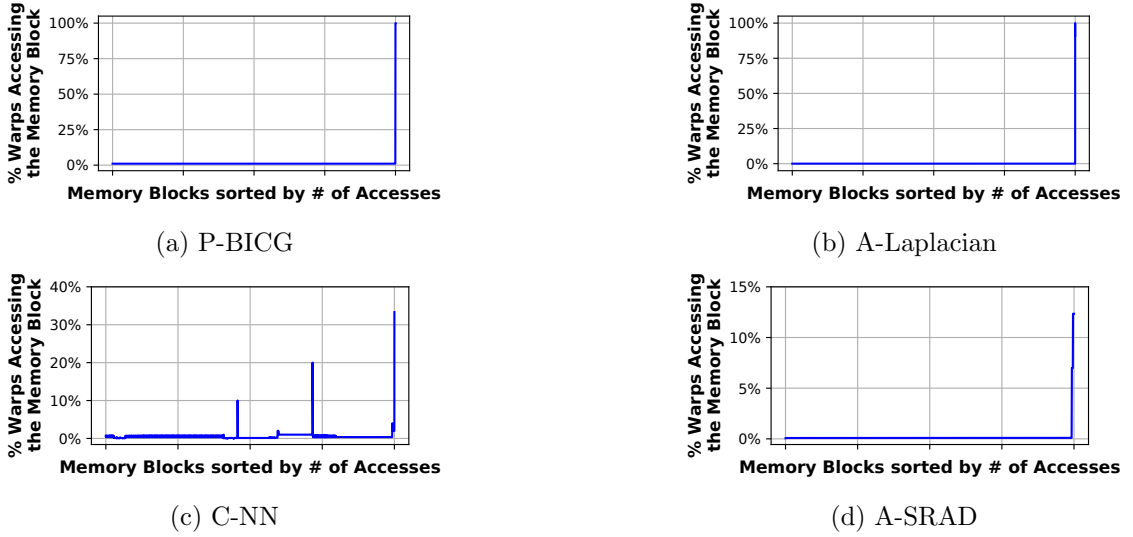


Figure 5.4: Percentage of active warps accessing the data memory blocks. © 2021 IEEE.

Figure 5.4(a)-(b) (P-BICG and A-Laplacian), show that the highly accessed memory blocks are also shared across all the active warps. This trend is representative of all applications in this study except for C-NN and A-SRAD. For C-NN and A-SRAD, see Figure 5.4(c)-(d), we note that while the highly accessed data memory blocks are not shared across all warps, they are still highly shared across multiple warps when compared to the rest of the memory blocks.

Observation II: Highly accessed data memory blocks are typically shared across a large number of warps compared to the rest of the memory blocks accessed by the applications. Therefore, an error in the *hot memory blocks* (that are highly accessed and shared) can spread across a large number of warps, making output degradation increasingly likely.

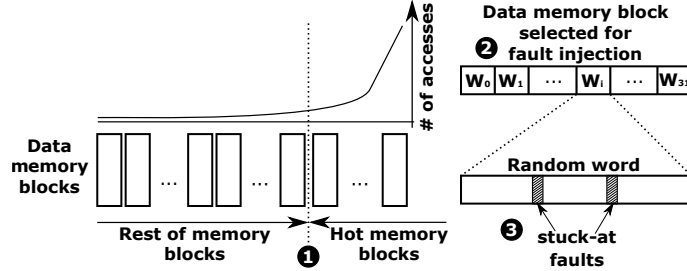


Figure 5.5: Fault injection methodology to evaluate application vulnerability of hot memory blocks to the memory faults. The data memory blocks are sorted based on the total number of accesses to each block. © 2021 IEEE.

5.3.3 Impact of Faults in Data Memory

Having identified the hot memory blocks in Section 5.3.2, here we test our hypothesis that faults in these hot memory blocks likely cause an SDC of the application output. Figure 5.5 illustrates our fault injection setup to demonstrate the effect of faults in the hot memory blocks as compared to the rest of the memory blocks. The data memory blocks are divided into two categories based on the access count profile shown in Figure 5.3: hot memory blocks and the rest of the memory blocks (1). As explained in Section 5.2.3, we do two distinct experiments: 1) with 1 block for fault injections per run and 2) with 5 blocks for fault injections per run. To demonstrate the likelihood of SDC output if faults occur in the hot memory blocks, we select random data memory blocks only from the hot memory blocks. We randomly target a word within each selected block (2) and then inject faults at random bit locations in the target word (3). Next, to show the likelihood of SDC output if faults are injected in the rest of the memory blocks, we select random data memory blocks only from that space for fault injection.

We compare the SDC of the application output in both cases. Figure 5.6 shows the number of SDC outcomes for the hot memory blocks and the rest of the memory blocks. For all applications, we notice a clear trend: the number of SDC outcomes increases as the number of faults for the selected data memory blocks increase. Furthermore, as the number of faulty data memory blocks increase (5 faulty blocks vs 1 faulty block), the

number of SDC outcomes further increases.

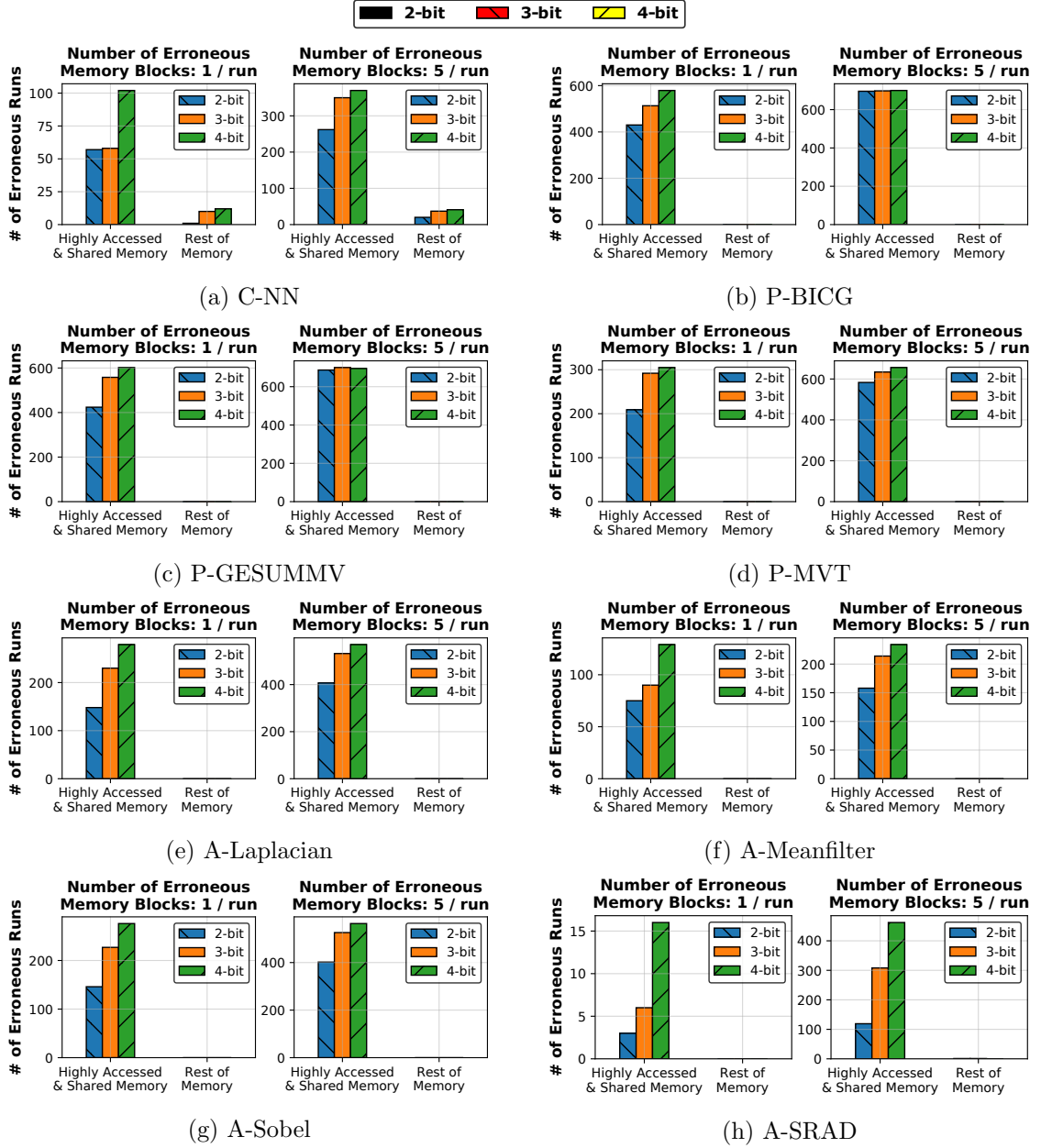


Figure 5.6: Effect of faults in the hot (highly accessed/shared) memory blocks versus the rest of the memory blocks on the application output. © 2021 IEEE.

Observation III: Faults in the hot memory blocks likely result in more SDCs in the application output comparing to faults in the rest of the memory blocks. Furthermore, as the number of faulty data memory blocks and/or the number of bit faults per data memory block increases, the probability of an SDC output increases.

5.4 Data-centric Reliability Management: Analysis, Design, and Implementation

In this section, we describe the application source code analysis to identify hot data objects. Based on this analysis, we introduce two resilience schemes that prioritize the hot memory for reliability protection to minimize SDCs while incurring low performance loss.

5.4.1 Application Source Code Analysis

As noted in Observation III, the hot memory blocks must be prioritized for reliability protection. Therefore, we first identify to which input data objects in the application source code these hot memory blocks belong. We start by examining the read-only data objects in the application source code against the load instructions in the corresponding PTX code. Here, we provide the analysis for the three representative applications – namely, P-BICG, C-NN, and A-Laplacian.

We begin with P-BICG. The relatively simple source code of this application facilitates understanding the access pattern to the data objects of interest. P-BICG application accepts three read-only input data objects – A , r and q – for two CUDA kernel functions. Listing 5.1 shows the source code for the first kernel, `bicg_kernel1`, which accepts A and r . From Listing 5.1, we note that A and r are accessed by each thread of a kernel in a for-loop on line 14. After examining the PTX code for P-BICG in relation to the addresses of the hot memory blocks, we note that only the memory blocks of data object r are highly accessed. This can be seen by examining the access patterns of A and r with respect to their access indices, $[i * NY + j]$ and $[i]$, respectively. Note that the offset for the index of the data objects A increases by a large value of $i * NY + j$. Consequently, the data memory blocks of A show low locality, and hence are not highly accessed and shared. On the other hand, the index of r increases by a small value of i , which results in uniformly strided accesses with a high locality. Consequently, the data memory blocks

of **r** are highly accessed. We notice a similar access pattern for **q** in the second kernel of P-BICG. Out of three read-only input data objects to P-BICG, **r** and **q** experience a very high number of accesses and are shared across multiple warps.

Listing 5.1: First Kernel in P-BICG.

```

1  #define NX 3072
2  #define NY 3072
3
4  __global__ void bicg_kernel1(float *A, float *r, float *s)
5  {
6      int j = blockIdx.x * blockDim.x + threadIdx.x;
7
8      if (j < NY)
9      {
10         s[j] = 0.0f;
11         int i;
12         for(i = 0; i < NX; i++)
13         {
14             s[j] += A[i * NY + j] * r[i];
15         }
16     }
17 }

```

Next, we analyze source code and the corresponding PTX code of C-NN and observe that the data objects **Layer1_Weights** and **Layer2_Weights**, which are inputs to the kernel functions of the first (shown in Listing 5.2) and second (not shown here) layers of C-NN, are highly accessed and shared across different warps. Here, we focus only on **Layer1_Weights** which incurs the highest number of accesses. Note that **Layer1_Weights** is accessed on lines 11 and 15 in **FirstLayer** kernel of C-NN, see Listing 5.2. The access generated by a thread of a block on line 11 is to the same data element of **Layer1_Weights** across threads of a cooperative thread array (CTA)-block. As a result, the corresponding data memory blocks of **Layer1_Weights** experience a high number of accesses from a large number of threads from different warps. The next access to **Layer1_Weights** on line 15 is inside a for-loop. Additionally, the offset to the index is regular and small (**[weightBegin+i]**). Consequently, as noted for P-BICG, this results in uniformly strided accesses to the data memory blocks of **Layer1_Weights**. Since a large number of threads execute the corresponding for-loop, the data memory blocks of **Layer1_Weights** get a very

high number of accesses. This is also true for the data memory blocks of `Layer2_Weights` in the next kernel of C-NN.

Listing 5.2: First layer in C-NN.

```

1  __global__ void FirstLayer(float *Layer1_Neurons, float *Layer1_Weights, float *Layer2_Neurons)
2  {
3      int blockID=blockIdx.x;
4      int pixelX=threadIdx.x;
5      int pixelY=threadIdx.y;
6      int weightBegin=blockID*26;
7      int windowX=pixelX*2;
8      int windowY=pixelY*2;
9      float result=0;
10     result+=Layer1_Weights[weightBegin];
11     ++weightBegin;
12     for(int i=0; i<25; ++i)
13     {
14         result+=Layer1_Neurons[(windowY*29+windowX+kernelTemplate[i])+(29*29*blockIdx.y)]*
15             Layer1_Weights[weightBegin+i];
16     }
17     result=(1.7159*tanhf(0.66666667*result));
18     Layer2_Neurons[(13*13*blockID+pixelY*13+pixelX)+(13*13*6*blockIdx.y)]=result;
19 }

```

Lastly, we examine the source code of A-Laplacian shown in Listing 5.3. The access profile (Figure 5.3(d)) identifies the data memory blocks of the filter data object `dLaplacianMatrix` as the most highly accessed (see line 24). In contrast to P-BICG and C-NN, the index offset of `dLaplacianMatrix` does not change linearly. However, since the entire `dLaplacianMatrix` fits in one memory block, its accesses converge to that memory block. Consequently, the data memory block of `dLaplacianMatrix` is highly accessed and shared across multiple warps. Following `dLaplacianMatrix`, `width`, and `height` are the next most highly accessed and shared data objects.

Listing 5.3: Filter Kernel in A-Laplacian.

```

1  --global-- void LaplacianFilter(Pixel* g_DataIn, Pixel* g_DataOut,
2      int* width, int* height, float* d_LaplacianMatrix)
3  {
4      --shared-- Pixel sharedMem[BLOCK_HEIGHT*BLOCK_WIDTH];
5      int x = blockIdx.x * TILE_WIDTH + threadIdx.x;
6      int y = blockIdx.y * TILE_HEIGHT + threadIdx.y;
7      if( x < FILTER_RADIUS || x > *width - FILTER_RADIUS - 1 || y < FILTER_RADIUS || y >
          *height - FILTER_RADIUS - 1)
8      {
9          int index = y * (*width) + x;
10         g_DataOut[index] = g_DataIn[index];
11         return;
12     }
13     int index = y * (*width) + x;
14     int sharedIndex = threadIdx.y * blockDim.y + threadIdx.x;
15     sharedMem[sharedIndex] = g_DataIn[index];
16     --syncthreads();
17     if( threadIdx.x >= FILTER_RADIUS && threadIdx.x < BLOCK_WIDTH - FILTER_RADIUS &&
        threadIdx.y >= FILTER_RADIUS && threadIdx.y < BLOCK_HEIGHT - FILTER_RADIUS)
18     {
19         float sum = 0;
20         for(int dy = -FILTER_RADIUS; dy <= FILTER_RADIUS; ++dy)
21             for(int dx = -FILTER_RADIUS; dx <= FILTER_RADIUS; ++dx)
22             {
23                 float centerPixel = (float)(sharedMem[sharedIndex + (dy * blockDim.x + dx
24                     )]);
25                 sum += centerPixel * d_LaplacianMatrix[(dy + FILTER_RADIUS) *
26                     FILTER_DIAMETER + (dx+FILTER_RADIUS)];
27             }
28         Pixel res = max(0, min((Pixel)sum, 255));
29         g_DataOut[index] = res;
30     }
31 }

```

We performed similar application source code analysis for all applications studied here. Table 5.3 lists the read-only input data objects for the GPU kernel functions of each application along with their respective sizes. The data objects are ordered from high to low in terms of the number of accesses, those identified as hot data objects are in bold and can be identified by examining the application source code. Lastly, from Table 5.3, we note that while the hot data objects are highly accessed and shared, they occupy significantly less space than the rest of the data objects combined. For example, in C-NN, the hot data objects, that is **Layer1_weights** and **Layer1_Weights**, occupy only 2.15% of the total application data memory. We notice a similar trend across all applications.

We also performed runtime temporal analysis on the accesses to the hot data objects. Since, in most applications, accesses to the data objects are uniformly strided with small offset, the accesses have high temporal locality (e.g., for P-BICG). For other applications such as A-Laplacian, since the hot data objects are small enough to fit in few data memory blocks, accesses to these exhibit high temporal locality.

Table 5.3: Input data objects to the GPU applications. Data objects are sorted based on the number of accesses incurred (Highest to Lowest). The emboldened data objects are classified as hot data objects (highly accessed and shared). © 2021 IEEE.

Application	Input Data Objects	Size of hot memory blocks normalized to the total application memory (in percentage)	Percentage of accesses to hot memory blocks w.r.t. the total number of accesses
C-NN	Layer1_Weights , Layer2_Weights , Layer3_Weights, Layer4_Weights, Images	2.15	34.99
P-BICG	p , r , A	0.064	5.7
P-GESUMMV	x , A, B	0.025	4.8
P-MVT	y1 , y2 , a	0.048	5.8
A-Laplacian	Filter , Filter_Height , Filter_Width , Image	0.001	73
A-Meanfilter	Filter_Height , Filter_Width , Image	0.0001	39.89
A-Sobel	Filter , Filter_Height , Filter_Width , Image	0.001	73
A-SRAD	i_N , i_S , i_E , i_W , Image	0.86	39.67

Observation IV: Through offline application source code analysis, the hot data objects forming the hot memory blocks can be identified. Furthermore, these hot data objects have a very small memory footprint (at the most 2.15%) compared to the rest of the data objects. Lastly, the hot data objects experience high temporal locality.

Note that this analysis can be adapted for other applications using available binary instrumentation tools for GPUs [127, 1]. The binary instrumentation tools offer two useful functionalities: First, the memory tracing functionality can be extended to identify the hot memory blocks. Second, the application instruction profiling at the binary level can help to identify the hot data objects. The access pattern and source code analyses are done once offline, and therefore, have no runtime overhead.

5.4.2 Detection and Correction Resilience Schemes

We leverage the information related to hot memory blocks (Observations I, II, and IV) to devise detection/correction schemes. We particularly focus on Observation III that demonstrates that the hot data objects must be prioritized for protection against multi-bit faults. As discussed in Section 5.3, the proposed resilience schemes target multi-bit faults in L2-cache and DRAM. Our resilience schemes complement the existing SECDED-ECC protection.

5.4.2.1 Multi-bit Fault Detection

As the read-only hot data objects prioritized for protection are smaller in size compared to the total application memory (refer to Table 5.3), we replicate the hot data objects for “protection”. Replication allows to easily identify the multi-bit faults by comparing their two copies.

Given an application, we first sort the data objects based on the number of their accesses and identify the hot data objects (this is done with a one-time offline source code analysis as described in Section 5.4.1). For the applications studied in this work, Table 5.3 lists all the data objects per application sorted from high to low number of accesses. The hot data objects to be prioritized for reliability protection are emboldened.

Next, we duplicate the selected data objects in the GPU DRAM at two distinct locations. During the application execution, if a memory access to the data memory blocks of one of the reliability-protected data objects is an L1-cache hit, then the normal operation takes place where the data is returned to the corresponding SM core. However, if the access is an L1-cache miss, then the LD/ST unit at the L1-cache generates two accesses, each to one of the two copies of the data memory block. Once both accesses return data to L1-cache, the copies of data are compared bit-wise to identify any multi-bit faults. If a bit mismatch is identified, then our reliability scheme generates a *terminate* signal to the GPU application causing the application to exit early and notify the user. In this case,

the user is expected to rerun the application.

Since the detection-only scheme duplicates the L1-cache missed accesses to the data memory blocks of selected read-only data objects, the main source of performance loss is the additional accesses going to the L2-cache and DRAM. To minimize performance loss, we leverage the fact that this is a detection-only scheme: if the protected data is corrupted, then the application is terminated. Therefore, it is not necessary to wait for both copies of the data to arrive before proceeding with the application execution. Instead, we devise a *lazy* bit comparison: once we receive the first data copy for a corresponding load instruction, the execution moves forward. As soon as the second copy is received, then the *lazy* comparison is performed to check for multi-bit faults. Consequently, any performance loss is minimized as the execution is not stalled.

5.4.2.2 Multi-bit Fault Detection-and-Correction

We next describe the second resilience scheme which not only detects multi-bit faults but also corrects them. To detect and correct the multi-bit faults, we employ a majority vote mechanism that is implemented via data triplication. Each copy is stored at a distinct location in the GPU DRAM with distinct memory addresses. For each L1-cache missed access for the data object covered under the reliability scheme, we generate three accesses to the L2-cache. Once all three accesses are returned to the LD/ST unit at the L1-cache, we perform a three-way bitwise comparison on the received data copies. During the comparison, if all the data copies have the same bits indicating no bit fault, then the application execution moves forward. If a bit mismatch is observed in one of the copies indicating a bit fault, then based on the majority vote the offending bit is changed to the correct value. The corrected bit value is used for the computation. Since the data copies are stored at distinct locations, the probability of the same bit fault occurring in all three data copies is minimal.

In this detection and correction scheme, we wait for all three data copies to be received in order to perform the three-way comparison for data correction. Consequently, the two

sources of performance loss are 1) the increased number of memory accesses due to the three accesses to the data and 2) the stall times when the LD/ST unit at L1-cache waits for all three accesses to return with data. For the set of applications examined here, we do not observe a significant performance loss, because the size of the input data objects prioritized for the reliability improvement is small as shown in Table 5.3.

5.4.3 Implementation Overhead

In Section 5.4.1, we identified the hot data objects via manual application of source code analysis. For an unknown application, the same access pattern analysis can be automated with the assistance of binary instrumentation tools, such as NVBit [127]. Note that this information collection is a one-time process and typically done offline. Based on the profiled information, the following steps are performed.

First, we replicate the data objects protected by our resilience schemes in GPU DRAM (either two or three times, depending on our target). We store the start addresses of each copy of the data object. These start addresses are used to generate the replication accesses to the required data index within the data object. To do so, we add the memory offset calculated for the original memory access to the respective start address. For each data object, we need either 32 bits or $(2 \times 32 =) 64$ bits to store the start addresses in the detection-only and detection-and-correction, respectively. We allocate 128 bytes for the start address storage, which accommodates $(128B/(32 \times 4) =) 32$ and $(128B/(2 \times 32 \times 4) =) 16$ data objects for detection and detection/correction, respectively. In our analysis, the maximum number of data objects to a GPU application never exceeds five (Table 5.3). We use a 32-bit adder to compute the data index mentioned above for the copy accesses.

Second, to replicate the L1-cache missed accesses to the protected data objects, we track their respective load instructions. To do so, we store the addresses of load instructions to the corresponding data objects in the LD/ST unit near L1-cache. Each load instruction needs 32 bits to store its address. We allocate 128 bytes for the instruction address storage, which accommodates $(128B/(32 \times 4) =) 32$ load instruction addresses.

In our applications, the number of load instructions does not exceed 22. The LD/ST unit checks the program counter to see if one of the load instructions to the protected data objects experiences a miss, in which case, additional accesses are generated to the copies of data objects. To compare the data copies, we use a 256-bit wide comparator for comparing the data at 32B granularity. Lastly, we allocate 128 bytes to store at most 32 load instructions awaiting the comparison of their data copies at the LD/ST unit. Note that all overheads associated with the data movement and stalls are modeled and final results already include these overheads.

5.5 Experimental Results

In this section, we experimentally evaluate the proposed detection-only and detection-and-correction resilience schemes using the applications listed in Table 5.2.

5.5.1 Performance Evaluation

Note that the results presented in this subsection come from one profiling run only. Figure 5.7 plots for each application a) the execution time for each application and b) the L1-cache missed accesses. All metrics in Figure 5.7 are plotted normalized to the baseline case (i.e., the baseline execution with no resilience scheme). Therefore, the “1.0” value on the y-axis in each plot represents the baseline value. Note that the numeric values on the x-axis correspond to the cumulative number of data objects covered under the resilience schemes. The data objects covered are by their order of importance as shown in Table 5.3. For example, for C-NN, “1” corresponds to `Layer1_weights`, while “2” corresponds to `Layer1_weights` and `Layer2_weights`, and so on.

5.5.1.1 Detection-Only

For evaluating performance, we focus on the overhead due to the duplication of accesses in the detection-only resilience scheme. Therefore, we ignore the cases where data memory

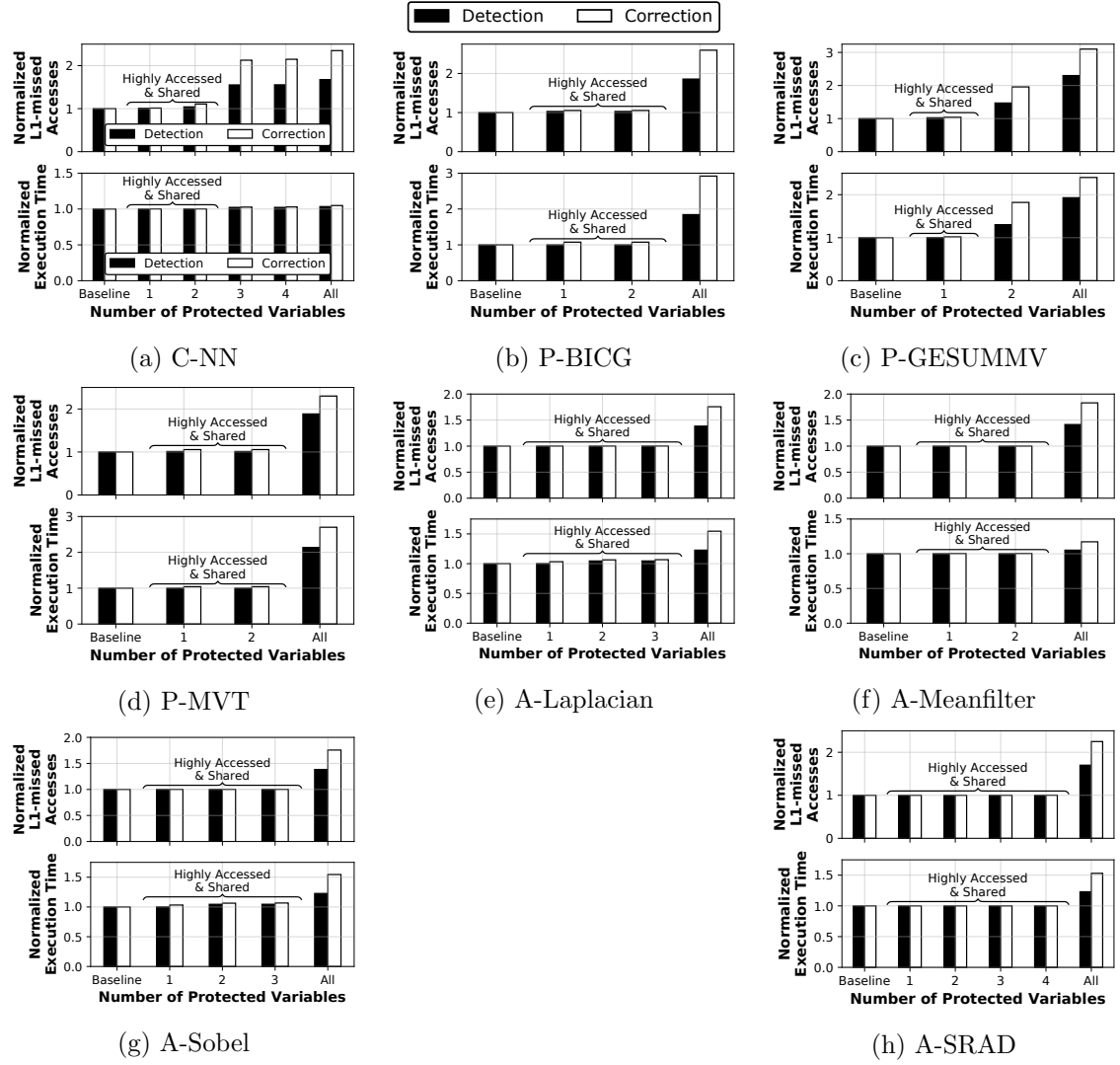


Figure 5.7: Performance overhead of Detection-only (dark bar) and Detection-and-Correction (white bar) resilience schemes. All numbers are normalized to the baseline case (no reliability protection, 1.0). The hot data objects reside in hot memory blocks. © 2021 IEEE.

errors result in application crashes. From Figure 5.7, we make the following observations. First, across all applications, as the number of data objects covered by the detection-only resilience scheme increases, the respective application execution times increase. This loss in performance is consistent with the increase in L1-cache missed accesses due to duplication. Second, the L1-cache missed accesses increase fractionally when we cover *only the hot data objects*, which is attributed to their small memory footprint in addition

to their spatial and temporal locality (Observation IV). Lastly, the detection-only scheme implements a lazy bit-wise evaluation, where application execution proceeds when any copy of the duplicated data arrives at the LD/ST unit of L1-cache. (Recall that the execution does not stall awaiting both accesses to arrive.) Therefore, when only the hot data objects are protected, the corresponding performance loss on average is only 1.2%, see Figure 5.7. In contrast, when *all* data objects are protected, the average performance loss becomes 40.65% due to the steep increase in duplicated accesses.

5.5.1.2 Detection-and-Correction

We make the following observations from Figure 5.7 regarding the detection-and-correction scheme. First, similar to the detection-only scheme, as the number of protected data objects increases, the L1-cache missed accesses increase but this increase is larger comparing to detection-only. This is expected as accesses are now triplicated. In addition, to correct the fault(s), execution is stalled for all three accesses to arrive with data. Consequently, the execution time increases as a function of the volume of the protected data objects.

It is interesting to note that when we enable detection-and-correction for hot data objects only, the corresponding average performance loss is only 3.4% as the increase in the number of L1-cache missed accesses is still minimal (almost at the same level as for the detection-only scheme). If all application data objects were to be triplicated, the average performance loss shoots to 74.24%. Overall, the performance loss triggered by the detection-and-correction is much higher than for the detection-only scheme if all data objects are protected but it is nearly at the same level with detection-only if only hot data objects are protected.

5.5.2 Reliability Evaluation

We evaluate the two resilience schemes based on the percentage of SDC outcomes for 1000 fault injection experiments. Recall that this number of experiments is necessary to achieve results of statistical significance (95% confidence intervals with $\pm 3\%$ error

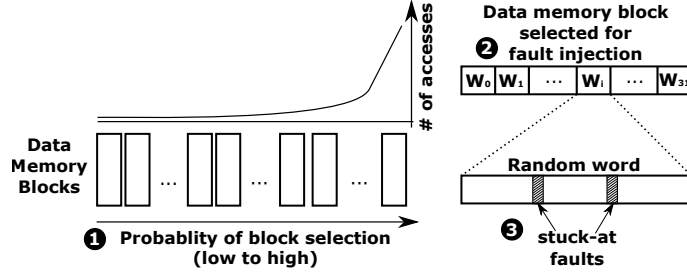


Figure 5.8: Fault injection for evaluating fault detection-and-correction: the probability of a memory block selection depends on the number of its L1-missed accesses (since the proposed schemes address faults in L2-caches and DRAM). © 2021 IEEE.

margins) [62, 33].

We inject faults in the entire application memory space, see Figure 5.8. Specifically, for reliability evaluation, we select the data memory block(s) where the faults are to be injected based on its number of L1-missed accesses (a missed access forces bringing data from L2-caches and DRAM which are highly susceptible to faults) during an application run (❶). Recall that we perform two distinct experiments: 1) with 1 block for fault injections per run and 2) with 5 blocks for fault injections per run (see Section 5.2.3). We randomly target a word within the selected memory block(s) for fault injections (❷) and then inject faults at random bit locations in the selected word (❸).

Figure 5.9 plots the number of SDC outcomes versus the (cumulative) number of data objects protected by the resilience schemes. Across all applications, the baseline case with no enabled resilience scheme is more susceptible to faults. As we cumulatively protect more data objects, the number of SDC outcomes reduces. When either the per memory block bit faults or the number of faulty memory blocks increases, the number of SDC outcomes increases as well. Across all applications, protecting hot data objects with the proposed resilience schemes decreases the number of SDC outcomes significantly (an average drop of 98.97%) across all fault injection configurations.¹

¹In some cases, we observe that the number of SDC outcomes is less than 3% (the statistical error margins). However, the majority of cases, especially at higher fault rates, demonstrate clear benefits of our schemes.

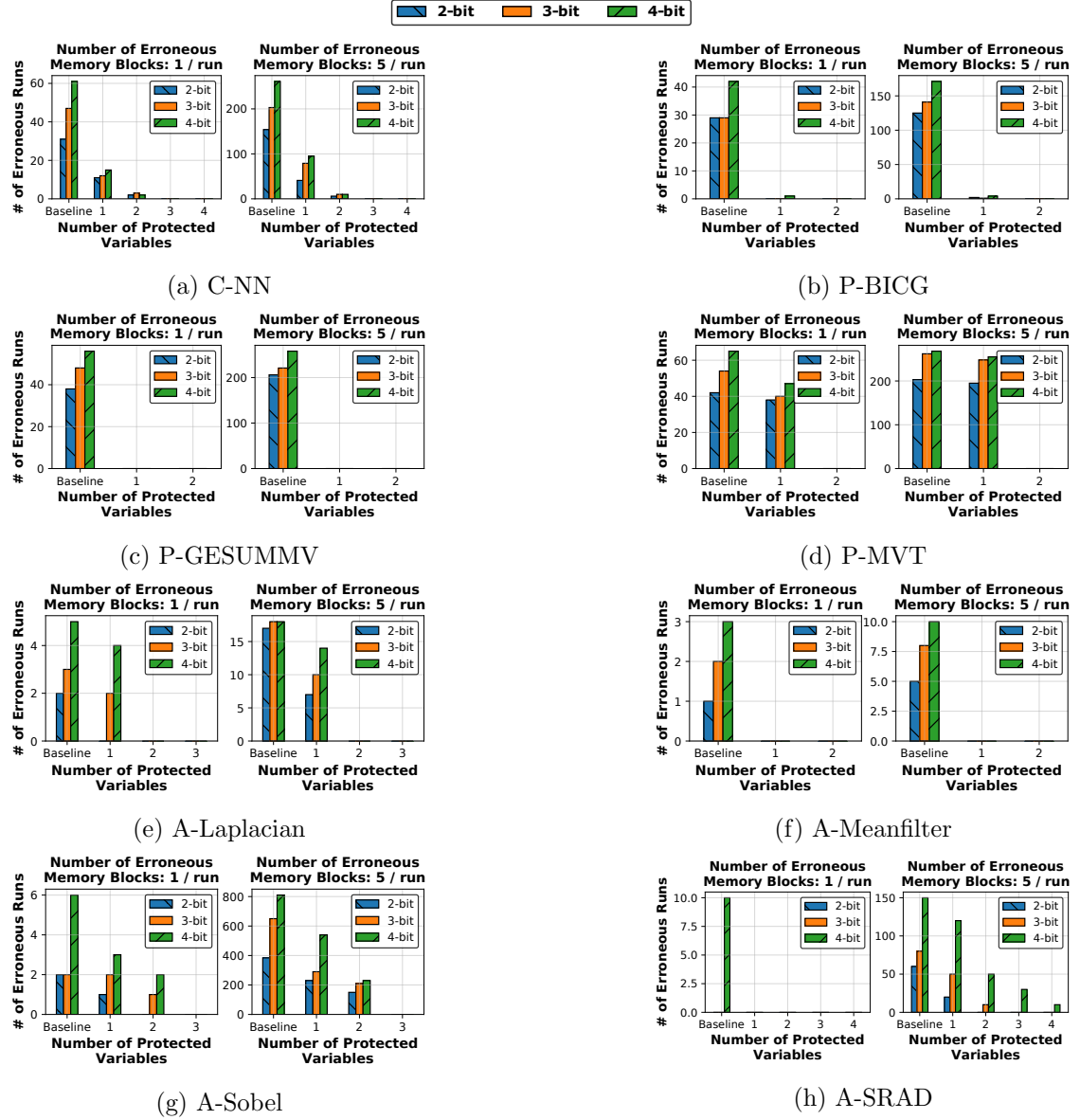


Figure 5.9: Silent data corruption due to faults in L2-cache and DRAM: The x-axis represents the number of protected data objects cumulatively increasing, starting from the baseline case (no data objects are protected). The y-axis shows the number of SDC outputs out of 1000 runs for each error injection configuration. The detection-only/detection-and-correction schemes stop the multi-bit data memory errors caused by the faults from propagating to the output. © 2021 IEEE.

5.5.3 Reliability and Performance Tradeoff

Figure 5.9 shows that in the absence of resilience schemes, GPGPU applications are highly vulnerable to multiple memory faults. Yet, as we cumulatively protect an increasing number of data objects, the number of SDC outcomes decreases, but at a small performance cost. Figure 5.7 shows that since the focus is on protecting a limited number of hot data objects, the performance degradation due to protection is indeed minimal. Across all applications, with the detection-and-correction (the detection-only) scheme, on average hot memory blocks can be protected with a performance loss of only 3.4% (1.2%) resulting in a 98.97% drop in the number of SDC outcomes. On the contrary, if all data are protected the performance loss becomes 74.24% (40.7%). By selecting the number of data objects to be protected and especially when protection is applied to hot data objects only, the desired reliability and performance tradeoff can be achieved.

5.6 Related Works

To our knowledge, this is the first work that makes a case for data-centric reliability management in GPUs. In this section, we briefly discuss the works that are most related to ours.

Memory/Cache Errors. Sridharan et al. [120] discovered that almost half of the DRAM faults are multi-bit failures, and more than 50% of the DRAM faults are permanent. Tiwari et al. [125] showed through a large scale GPU study that GPU DRAM is most vulnerable to multi-bit errors compared to the rest of GPU hardware. Furthermore, two independent studies demonstrate the necessity of improved ECC, such as Chipkill, instead of SECDED due to increasing multi-bit errors in DRAMs [77, 121]. Due to increasing cache sizes, several efforts have been developed to operate caches at low voltage to improve power efficiency. Recent works have demonstrated experimentally that bit faults increase as the operating voltage of the cache reduces [28, 11, 27]. In this paper, we address these multi-bit faults in cache/memory via low-overhead detection/correction mechanisms.

Error Injection Studies. GPU-Qin injects fault at the micro-architecture level to simulate transient faults in GPUs, excluding caches and memory [24, 25]. LLFI [64] is an LLVM compiler-based fault injection framework for GPUs, where an intermediate representation is modified to simulate error injection. SASSIFI [33] directly injects faults into low-level SASS instructions. PCFI [110] inject errors in different parts of instructions to simulate errors in the GPU register files and memory. Unlike the compiler-based methods used in GPU-Qin and SASSIFI, Tselonis et al. [126] propose GUFi to validate the feasibility of using the commonly used GPGPU simulator, GPGPU-Sim [6] to study the reliability of GPGPU applications. Nie et al. [88] propose a fault-site pruning mechanism that dramatically reduces the number of required fault-injection experiments in GPGPU applications to obtain results of high statistical significance, this pruning methodology is also adapted for multi-bit faults [138]. SUGAR [139] speeds up the evaluation of GPGPU application error resilience by judicious input sizing and illustrates how analyzing a small fraction of the input is sufficient to estimate application resilience with high accuracy while dramatically reducing experimentation time.

Reliability Solutions. Redundant computations by modifying source code are explored for fault tolerance as GPUs have a large number of on-chip cores [21]. Thread remapping into reliable and unreliable warps can facilitate partial replication mechanisms for error detection/correction at the warp level and shows superior performance to standard duplication/triplication [137]. Nie et al. [84] show that when a quantifiable loss in output quality is acceptable to the user, one can reduce the overhead of protection/recovery mechanisms by taking advantage of resilience patterns of threads at different hierarchies (i.e., kernel/thread-block/warp). Compiler-based redundant multithreading (RMT) compares the outputs from replicated computations for error detection, albeit with a highly variable performance loss [129, 31]. Mahmoud et al. [75] introduce a replication algorithm to duplicate select GPU instructions while maintaining low performance loss. Another approach for fault-tolerance is checkpoint-restart, where upon the fault occurrence the application restarts from the last checkpoint [90, 29]. However, the associated overhead

of the checkpoint-restart mechanism is prohibitive [58].

For caches operating at low voltage, Killi [27] offers a variable ECC mechanism for a subset of L2 cache lines, while disabling the cache lines with more than one fault at the cost of cache capacity. Chandramoorthy et al. [11] implement a boosted SRAM cache, where the cache voltage is boosted for each read and write operation.

Prior works suggest heterogeneous reliability solutions for CPU workloads [38, 74, 63, 73, 32]. Hukerikar et al. [38] devise a software-based parity mechanism to improve the reliability of critical program objects in HPC applications. Luo et al. [74] show that applications exhibit different memory error resiliency based on the error location in DRAM and propose a hardware/software mechanism to enhance memory reliability. Li et al. [63] profile scientific applications to relate changes in application behavior and the location and frequency of error. SDCTune [73] identifies and protects SDC-prone program data based on static and dynamic features. Hari et al. [32] deploy low-cost program detectors in the SDC-crucial section of the program to identify and reduce SDCs. Ranger [15] restricts output values of selected layers in CNNs to minimize error propagation to improve CNN resilience.

The schemes proposed in this work complement the SECDED ECC by detecting and correcting multi-bit faults in the GPU L2 cache and DRAM. To the best of our knowledge, this is the first work that identifies the most vulnerable data in the context of GPGPU application resilience. Based on this information, our schemes protect the highly-used input data objects and provide improved reliability at a low overhead.

5.7 Conclusions

Multi-bit faults are typically an unwanted side-effect of GPU memory performance innovations. In this paper, we perform an in-depth application-level analysis of memory access patterns and show that a large number of applications work on a limited number of hot data objects of highly-accessed data, which are also shared by a majority of warps. Such

highly accessed and shared data is vulnerable to faults potentially leading to silent data corruption in the application output. We show that as hot data objects constitute a small fraction of the total memory footprint, protecting them against faults is an inexpensive solution that provides high application resilience in the presence of multi-bit faults.

Chapter 6

Exploration of the Reliability of ML Models

The emerging machine learning (ML) models are becoming increasingly complex. As a result, the ML models need larger storage space and are getting slower. Consequently, several optimizations have been introduced to reduce the ML model size and the execution times. The popular optimizations include weight compression [22], weight pruning [140] and weight quantization [134]. However, the reliability of ML models in the presence of these optimizations is not explored in detail. To bridge this gap, this chapter explores the reliability of ML models that employ weight quantization.

6.1 Advantages of Weight Quantization in ML Models

This initial study specifically focuses on the weight quantization of the machine learning models, where the data type of weight parameters is quantized from the default 32-bit floating-point (FP32) to a smaller data type, for example, 8-bit integer (INT8). We evaluate the effect of the weight quantization on the model size, execution time, and classification accuracy across different ML models using the ImageNet dataset [115]. We use PyTorch [104] Framework for the evaluation. The results are summarized in Table 6.1.

Table 6.1: Comparison between 32-bit floating-point and weight quantized (INT8) ML models.

ML Model	Number of Convolution Layers	Size (MB)		Execution Speedup	Accuracy (%)	
		FP32	INT8		FP32	INT8
AlexNet [57]	5	234	59	4.02X	56.624	56.092
VGG19 [119]	16	575	144	5.26X	72.36	47.146
ResNet50 [36]	53	103	76	18.55X	76.012	75.854
InceptionV3 [123]	94	109	25	11.60X	77.248	77.062

From Table 6.1, we note that the weight quantization offers significant storage size reduction and execution speedups across all ML models studied here. Furthermore, except for VGG19, the degradation in classification accuracy is minimal.

6.2 Impact of Memory Faults on the Quantized Models

In this section, we evaluate the reliability of FP32 and INT8 ML models when memory faults occur. We quantify the reliability of an ML model by observing the classification accuracy. To do so, we focus on the images correctly classified during a baseline (fault-free) execution run. We use validation images from ImageNet dataset [115].

6.2.1 Fault Model

The convolution layers in the image classifier ML models are computationally important layers. Therefore, we focus on the faults in the convolution layers. Specifically, we inject faults in the weights of the convolution layers. We select a random weight element from a random convolution layer of the ML model under study. In the selected weight element we randomly flip two bits irrespective of the data type of the weight element. The faults are modeled as *permanent* faults. To meet the statistical significance, we execute 3000 fault injection runs [62, 33, 88].

6.2.2 Impact of Memory Faults

Figure 6.1 illustrates a histogram to demonstrate the impact of memory faults on the classification accuracy of FP32 and INT8 AlexNet models. In Figure 6.1, the x-axis

represents the percentage of mis-classifications per execution run. The bin size on the x-axis is 0.25%. On the y-axis, we plot the number of execution runs corresponding to the percentage of mis-classifications on the x-axis.

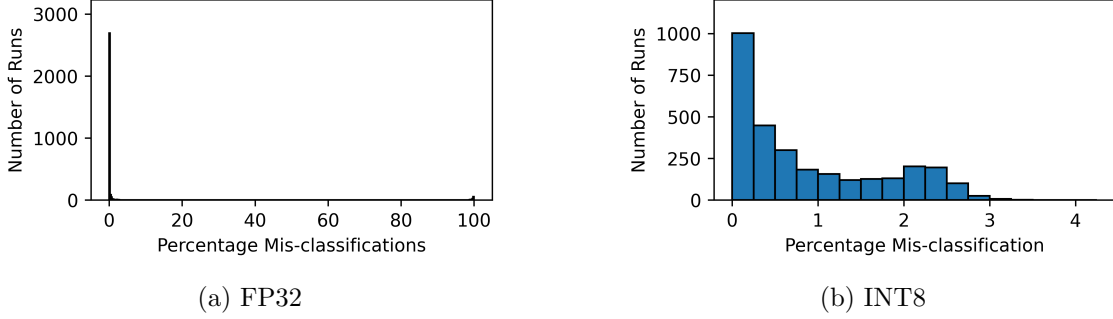


Figure 6.1: Impact of faults in a weight element on the classification accuracy in FP32 and quantized (INT8) AlexNet model.

In Figure 6.1, for FP32 model, we note that around 90% (2690 out of 3000) of execution runs experience a negligible percentage of mis-classifications (at most 2.5%) when the faults are injected in a weight element. In contrast to the FP32 model, for the weight quantized INT8 ML model, only around 33.4% (1002 out of 3000) of execution runs experience a very low percentage of mis-classifications. Around 66.7% of execution runs experience more than 2.5% of mis-classifications. *Therefore, we conclude that the weight quantized (INT8) ML model is more vulnerable to the faults in the weights compared to the FP32 ML model.*

Next, we identify the convolution layers most vulnerable to the faults. To this end, we segregate the fault injected execution runs in Figure 6.1 based on in which layer the faults were injected. Figure 6.2 illustrates the impact of faults on the classification accuracy for the individual convolution layers of AlexNet when selected fault injection. For both – FP32 and INT8 – models, we note that when the faults occur in the earlier convolution layers, the probability of mis-classification is higher as compared to when the faults occur in the latter layers. *Therefore, we conclude that the earlier layers of an ML model should be prioritized for reliability protection.*

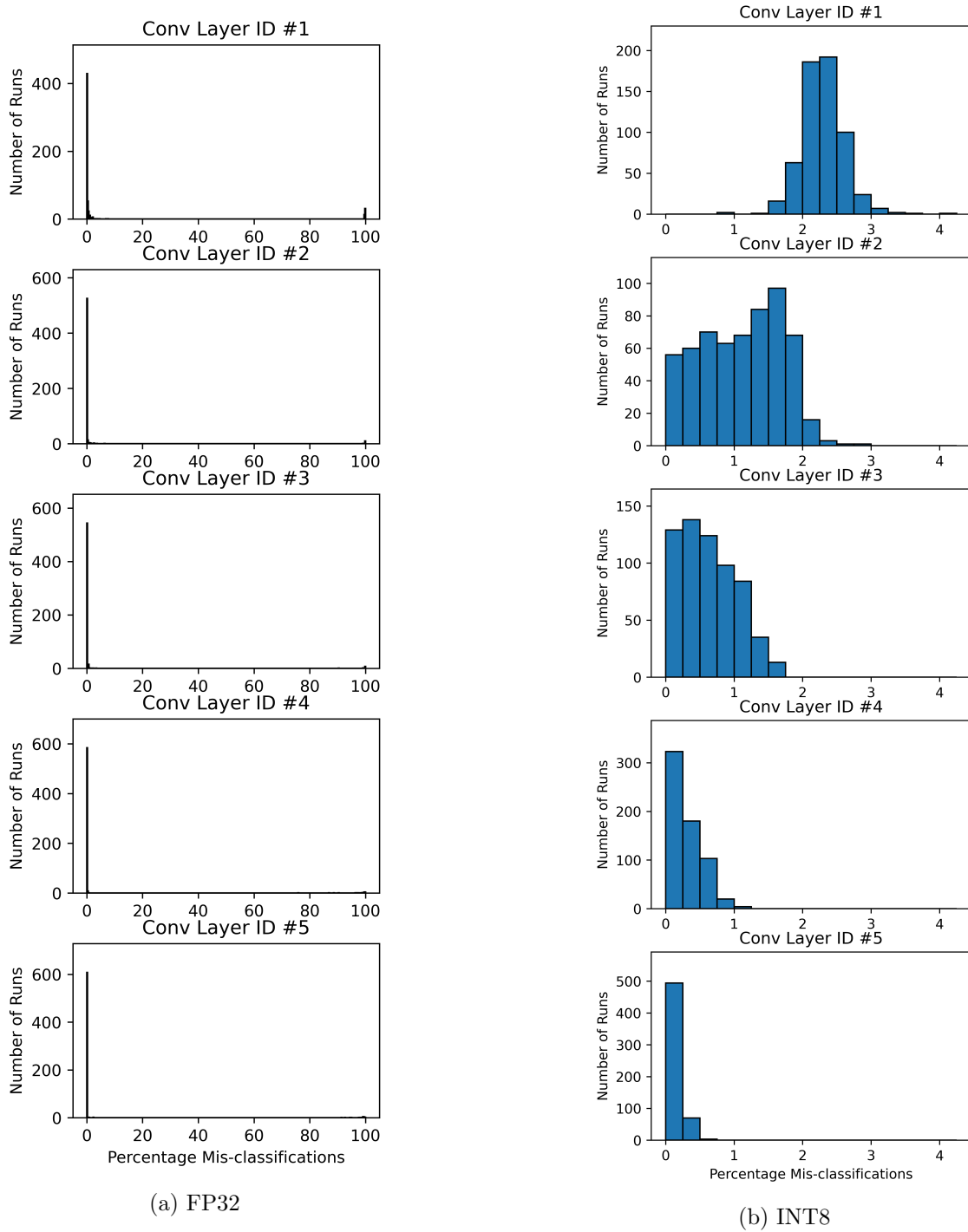


Figure 6.2: Contribution of faults in weights of different convolution layers of float 32 (FP32) and quantized (INT8) ML models on the classification accuracy.

The future work may consider exploring different low-overhead reliability techniques to improve the resiliency of the weight quantized ML models.

Chapter 7

Conclusion and Future Research Directions

7.1 Summary of Dissertation Contributions

GPUs exploit the inherent parallelism in applications to offer significant performance and power efficiency benefits over CPUs. As a result, GPUs are being increasingly deployed across a wide range of applications, such as financial computing, machine learning, medical imaging. Most of the aforementioned applications process confidential information, for example, encryption keys, user medical/financial data, etc. Furthermore, GPUs are also deployed in critical applications, such as self-driving cars, scientific computing, etc. These applications demand reliable compute operations on GPUs as faulty computations could lead to either fatalities or loss of resources. Consequently, our research focuses on secure and reliable GPU computations. To this end, we make the following contributions:

1. Improving GPU Security

For GPU security, our research focuses on mitigating a previously demonstrated correlation timing attack on GPUs. The attack exploited the deterministic nature of the memory access coalescing in GPUs to correlate the number of accesses and the execution time to recover the AES encryption keys. We introduce two hardware-based defense mech-

anisms to prevent the leakage of confidential data due to the correlation timing attack.

Our first defense mechanism, RCoal, randomizes the GPU coalescing mechanism so that a random number of memory accesses are generated which in turn makes the execution time unpredictable. Consequently, an attacker cannot correlate the number of accesses and the execution time, thus fails to recover the AES encryption keys. We evaluate RCoal scheme through the theoretical and empirical analysis to demonstrate that RCoal mitigates the correlation timing analysis. Furthermore, RCoal offers a tradeoff between the desired security and performance overhead.

While RCoal improves the GPU security against the correlation timing attack, it also incurs a high performance overhead as RCoal disables caches and miss-status holding registers (MSHRs) for security. To address the performance loss of RCoal, we introduce a bucketing-based coalescing mechanism, BCoal. BCoal always generates a constant number of accesses, termed as a bucket, by padding the original number of accesses with additional accesses whenever necessary. Additionally, BCoal generates the padded accesses such that the memory optimizations performed by the caches and MSHRs do not affect the security offered adversely. Consequently, BCoal offers security against the correlation timing attack with a minimal performance overhead. Our theoretical and empirical analysis show that BCoal successfully thwarts the correlation timing attack. BCoal also provides a tradeoff between the security and performance overhead, which can be set by the user.

2. Improving GPU Reliability

For GPU reliability, our research focuses on addressing the multi-bit data memory faults in the L2 cache and DRAMs while incurring a low performance overhead. To this end, we adopt a data-centric approach, wherein a user can decide the data memory coverage of the reliability scheme based on the desired performance overhead. We begin with application profiling to identify the fraction of the data memory highly accessed and shared across GPU threads. We call this highly accessed and shared memory as *hot* memory. We, furthermore, note that the memory footprint of the hot memory is very small compared to the rest of the application memory. Using the fault injection

simulations, We demonstrate that the memory faults in the hot memory will more likely result in GPU output corruption as compared to the memory faults in the rest of the memory. Consequently, we prioritize the hot memory for reliability protection. To this end, we propose two reliability schemes, detection-only and detection-and-correction, to address the memory faults. Both our reliability schemes offer a tradeoff between the desired reliability and performance overhead.

7.2 Future Research Directions

The machine learning (ML) applications are deployed in different applications, such as medical imaging and self-driving cars. These applications demand protection against attacks that may steal confidential data, such as patient details. Furthermore, the aforementioned applications need a reliability guarantee to ensure correct application output, for example, in a self-driving car. As a result, the security and reliability aspects of the ML applications are now leading research challenges.

On the security front, innovative attacks are launched against the hardware accelerators, such as GPU, to steal the private user information processed by the ML applications and the proprietary ML model parameters. Therefore, to improve the security of ML applications the following questions should be considered:

- What are the software and hardware vulnerabilities of ML applications that an attacker can exploit to launch an attack?
- How to mitigate the attacks on the ML applications while maintaining low performance overhead?

On the reliability front, the reliability of ML applications could be studied in light of different optimizations applied to the ML models to speedup their execution time and reduce their storage size. The baseline (unoptimized) ML models are usually quite resilient to the faults. However, the model optimizations, such as weight quantizations, can make the ML applications more vulnerable to the faults in the weight elements of the ML

model. Consequently, following questions are raised in regards to the reliability of the ML applications:

- How to quantify the adverse effects of different model optimizations, such as weight quantization and pruning, on the reliability of ML applications?
- How to improve the reliability of ML applications when different model optimizations are applied to the underlying ML models?

Bibliography

- [1] The CUDA Profiling Tools Interface (CUPTI). <https://docs.nvidia.com/cuda/cupti/>.
- [2] MOHAMMAD ABDEL-MAJEED AND MURALI ANNAVARAM. Warped register file: A power efficient register file for GPGPUs. In *HPCA*, 2013.
- [3] MOHAMMAD ABDEL-MAJEED, DANIEL WONG, AND MURALI ANNAVARAM. Warped gates: gating aware scheduling and power gating for GPGPUs. In *MICRO*, 2013.
- [4] SHAIZEEN AGA AND SATISH NARAYANASAMY. InvisiMem: Smart Memory Defenses for Memory Bus Side Channel. In *ISCA*, 2017.
- [5] MANISH ARORA, SIDDHARTHA NATH, SUBHRA MAZUMDAR, SCOTT B BADEN, AND DEAN M TULLSEN. Redefining the Role of the CPU in the Era of CPU-GPU Integration. *IEEE Micro*, pages 4–16, 2012.
- [6] A. BAKHODA, G. L. YUAN, W. W. L. FUNG, H. WONG, AND T. M. AAMODT. Analyzing CUDA workloads using a detailed GPU simulator. In *ISPASS*, 2009.
- [7] DANIEL J. BERNSTEIN. Cache-timing attacks on AES. cr.yp.to/papers.html#cachetiming, 2005.

- [8] ANDREY BOGDANOV, THOMAS EISENBARTH, CHRISTOF PAAR, AND MALTE WIE-NECKE. Differential Cache-Collision Timing Attacks on AES with Applications to Embedded CPUs. In *CT-RSA*. 2010.
- [9] JOSEPH BONNEAU AND ILYA MIRONOV. Cache-collision timing attacks against AES. In *CHES*. 2006.
- [10] M. BURTSCHER, R. NASRE, AND K. PINGALI. A quantitative study of irregular programs on GPUs. In *IISWC*, 2012.
- [11] N. CHANDRAMOORTHY, K. SWAMINATHAN, M. COCHET, A. PAIDIMARRI, S. EL-DRIDGE, R. V. JOSHI, M. M. ZIEGLER, A. BUYUKTOSUNOGLU, AND P. BOSE. Resilient low voltage accelerators for high energy efficiency. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 147–158, 2019.
- [12] N. CHATTERJEE, M. O’CONNOR, G. H. LOH, N. JAYASENA, AND R. BALASUBRAMONIAN. Managing DRAM Latency Divergence in Irregular GPGPU Applications. In *SC*, 2014.
- [13] SHUAI CHE, M. BOYER, JIAYUAN MENG, D. TARJAN, J.W. SHEAFFER, SANG-HA LEE, AND K. SKADRON. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IISWC*, 2009.
- [14] QIKAI CHEN, HAMID MAHMOODI-MEIMAND, SWARUP BHUNIA, AND KAUSHIK ROY. Modeling and Testing of SRAM for New Failure Mechanisms Due to Process Variations in Nanoscale CMOS. In *23rd IEEE VLSI Test Symposium (VTS 2005)*, 1-5 May 2005, Palm Springs, CA, USA, pages 292–297.
- [15] ZITAO CHEN, GUANPENG LI, AND KARTHIK PATTABIRAMAN. A Low-cost Fault Corrector for Deep Neural Networks through Range Restriction, 2020.

- [16] ZITAO CHEN, GUANPENG LI, KARTHIK PATTABIRAMAN, AND NATHAN DEBARDELEBEN. Binfi: An efficient fault injector for safety-critical machine learning systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery.
- [17] TONY CHENEAU, AYMEN BOUDGUIGA, AND MARYLINE LAURENT. Significantly improved performances of the cryptographically generated addresses thanks to ECC and GPGPU. *computers & security*, 29(4):419–431, 2010.
- [18] ANTHONY DANALIS, GABRIEL MARIN, COLLIN MCCURDY, JEREMY S. MEREDITH, PHILIP C. ROTH, KYLE SPAFFORD, VINOD TIPPARAJU, AND JEFFREY S. VETTER. The Scalable Heterogeneous Computing (SHOC) benchmark suite. In *GPGPU*, 2010.
- [19] TIMOTHY J. DELL. A white paper on the benefits of chipkill-correct ecc for pc server main memory. 1997.
- [20] LUIGI DILILLO AND BASHIR M. AL-HASHIMI. March CRF: an efficient test for complex read faults in SRAM memories. In *Proceedings of the 10th IEEE Workshop on Design & Diagnostics of Electronic Circuits & Systems (DDECS 2007)*, Kraków, Poland, April 11-13, 2007, pages 173–178.
- [21] MARTIN DIMITROV, MIKE MANTOR, AND HUIYANG ZHOU. Understanding software approaches for gpgpu reliability. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, page 94–104, New York, NY, USA, 2009. Association for Computing Machinery.
- [22] MINGHUI DONG, SHIPING WEN, ZHIGANG ZENG, ZHENG YAN, AND TINGWEN HUANG. Sparse fully convolutional network for face labeling. *Neurocomputing*, 331, 12 2018.

- [23] ANDERS EKLUND, PAUL DUFORT, DANIEL FORSBERG, AND STEPHEN M LA-CONTE. Medical image processing on the gpu-past, present and future. *Medical Image Analysis*, 2013.
- [24] B. FANG, K. PATTABIRAMAN, M. RIPEANU, AND S. GURUMURTHI. Gpu-qin: A methodology for evaluating the error resilience of gpgpu applications. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 221–230, 2014.
- [25] B. FANG, K. PATTABIRAMAN, M. RIPEANU, AND S. GURUMURTHI. A systematic methodology for evaluating the error resilience of gpgpu applications. *IEEE Transactions on Parallel and Distributed Systems*, 27(12):3397–3411, 2016.
- [26] C. W. FLETCHERY, L. REN, X. YU, M. VAN DIJK, O. KHAN, AND S. DEVADAS. Suppressing the Oblivious RAM timing channel while making information leakage and program efficiency trade-offs. In *HPCA*, pages 213–224, 2014.
- [27] SHRIKANTH GANAPATHY, JOHN KALAMATIANOS, BRADFORD M. BECKMANN, STEVEN RAASCH, AND LUKASZ G. SZAFARYN. Killi: Runtime fault classification to deploy low voltage caches without MBIST. In *25th IEEE International Symposium on High Performance Computer Architecture, HPCA 2019, Washington, DC, USA, February 16-20, 2019*, pages 304–316. IEEE, 2019.
- [28] SHRIKANTH GANAPATHY, JOHN KALAMATIANOS, KEITH KASPRAK, AND STEVEN RAASCH. On characterizing near-threshold sram failures in finfet technology. In *Proceedings of the 54th Annual Design Automation Conference 2017, DAC '17, New York, NY, USA, 2017*. Association for Computing Machinery.
- [29] ROHAN GARG, APOORVE MOHAN, MICHAEL SULLIVAN, AND GENE COOPERMAN. Crum: Checkpoint-restart support for cuda’s unified memory. pages 302–313, 09 2018.

- [30] DAVID GULLASCH, ENDRE BANGERTER, AND STEPHAN KRENN. Cache Games—Bringing Access-Based Cache Attacks on AES to Practice. In *S&P*, pages 490–505, 2011.
- [31] M. GUPTA, D. LOWELL, J. KALAMATIANOS, S. RAASCH, V. SRIDHARAN, D. TULLSEN, AND R. GUPTA. Compiler techniques to reduce the synchronization overhead of gpu redundant multithreading. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2017.
- [32] S. K. S. HARI, S. V. ADVE, AND H. NAEIMI. Low-cost program-level detectors for reducing silent data corruptions. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pages 1–12, 2012.
- [33] S. K. S. HARI, T. TSAI, M. STEPHENSON, S. W. KECKLER, AND J. EMER. 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)*, pages 249–258, 2017.
- [34] OWEN HARRISON AND JOHN WALDRON. AES Encryption Implementation and Analysis on Commodity Graphics Processing Units. In *CHES*, 2007.
- [35] BINGSHENG HE, WENBIN FANG, QIONG LUO, NAGA K. GOVINDARAJU, AND TUYONG WANG. Mars: A MapReduce Framework on Graphics Processors. In *PACT*, 2008.
- [36] K. HE, X. ZHANG, S. REN, AND J. SUN. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.
- [37] WEIZHE HUA, ZHIRU ZHANG, AND G. EDWARD SUH. Reverse Engineering Convolutional Neural Networks Through Side-channel Information Leaks. In *DAC*, 2018.

- [38] S. HUKERIKAR AND C. ENGELMANN. Havens: Explicit reliable memory regions for HPC applications. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, 2016.
- [39] HYNIX. Hynix GDDR5 SGRAM Part H5GQ1H24AFR Revision 1.0.
- [40] GORKA IRAZOQUI, MEHMET SINAN INCI, THOMAS EISENBARTH, AND BERK SUNAR. Wait a minute! A fast, cross-VM attack on AES. In *RAID*, pages 299–319. 2014.
- [41] K. IWAI, T. KUROKAWA, AND N. NISIKAWA. AES Encryption Implementation on CUDA GPU and Its Analysis. In *ICNC*, 2010.
- [42] KEON JANG, SANGJIN HAN, SEUNGYEOP HAN, SUE MOON, AND KYOUNG SOO PARK. SSLShader: Cheap SSL Acceleration with Commodity Processors. In *NSDI*, 2011.
- [43] SAURABH JHA, SUBHO S. BANERJEE, TIMOTHY TSAI, SIVA KUMAR SASTRY HARI, MICHAEL B. SULLIVAN, ZBIGNIEW T. KALBARCZYK, STEPHEN W. KECKLER, AND RAVISHANKAR K. IYER. ML-based fault injection for autonomous vehicles: A case for bayesian fault injection. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2019, Portland, OR, USA, June 24-27, 2019*, pages 112–124. IEEE, 2019.
- [44] Z. H. JIANG, Y. FEI, AND D. KAEI. A complete key recovery timing attack on a GPU. In *HPCA*, 2016.
- [45] ZHEN HANG JIANG, YUNSI FEI, AND DAVID KAEI. A Novel Side-Channel Timing Attack on GPUs. In *VLSI*, 2017.
- [46] ADWAIT JOG, ONUR KAYIRAN, ASIT K. MISHRA, MAHMUT T. KANDEMIR, ONUR MUTLU, RAVISHANKAR IYER, AND CHITA R. DAS. Orchestrated Scheduling and Prefetching for GPGPUs. In *ISCA*, 2013.

- [47] ADWAIT JOG, ONUR KAYIRAN, NACHIAPPAN C. NACHIAPPAN, ASIT K. MISHRA, MAHMUT T. KANDEMIR, ONUR MUTLU, RAVISHANKAR IYER, AND CHITA R. DAS. OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance. In *ASPLOS*, 2013.
- [48] GURUNATH KADAM, EVGENIA SMIRNI, AND ADWAIT JOG. Data-centric Reliability Management in GPUs. In *51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2021, Taipei, Taiwan, June 21-24, 2021*. © 2021 IEEE. Reprinted, with permission.
- [49] GURUNATH KADAM, DANFENG ZHANG, AND ADWAIT JOG. RCoal: Mitigating GPU Timing Attack via Subwarp-based Randomized Coalescing Techniques. In *HPCA*, 2018. © 2018 IEEE. Reprinted, with permission.
- [50] GURUNATH KADAM, DANFENG ZHANG, AND ADWAIT JOG. BCoal: Bucketing-Based Memory Coalescing for Efficient and Secure GPUs. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 570–581, 2020. © 2020 IEEE. Reprinted, with permission.
- [51] ONUR KAYIRAN, ADWAIT JOG, MAHMUT T. KANDEMIR, AND CHITA R. DAS. Neither More Nor Less: Optimizing Thread-level Parallelism for GPGPUs. In *PACT*, 2013.
- [52] SAMIRA MANABI KHAN, DONGHYUK LEE, AND ONUR MUTLU. PARBOR: An Efficient System-Level Technique to Detect Data-Dependent Failures in DRAM. In *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2016, Toulouse, France, June 28 - July 1, 2016*, pages 239–250.
- [53] JOSEF KINSEHER, MORITZ VÖLKER, AND ILIA POLIAN. Improving Testability and Reliability of Advanced SRAM Architectures. *IEEE Trans. Emerg. Top. Comput.*, 7(3):456–467, 2019.

- [54] DAVID KIRK AND W. W. HWU. *Programming Massively Parallel Processors*. Morgan Kaufmann, 2010.
- [55] JOHN KLOOSTERMAN, JONATHAN BEAUMONT, MICK WOLLMAN, ANKIT SETHIA, RON DRESLINSKI, TREVOR MUDGE, AND SCOTT MAHLKE. Warppool: Sharing requests with inter-warp coalescing for throughput processors. In *MICRO*, 2015.
- [56] BORIS KÖPF AND MARKUS DÜRMUTH. A Provably Secure And Efficient Countermeasure Against Timing Attacks. In *CSF*, pages 324–335, 2009.
- [57] ALEX KRIZHEVSKY, ILYA SUTSKEVER, AND GEOFFREY E. HINTON. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1, NIPS’12*, page 1097–1105, Red Hook, NY, USA, 2012. Curran Associates Inc.
- [58] K. LEE, M. B. SULLIVAN, S. K. S. HARI, T. TSAI, S. W. KECKLER, AND M. EREZ. On the trend of resilience for gpu-dense systems. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks – Supplemental Volume (DSN-S)*, pages 29–34, 2019.
- [59] SHIN-YING LEE AND CAROLE-JEAN WU. Characterizing GPU Latency Hiding Ability. In *ISPASS*, 2014.
- [60] VICTOR W. LEE, CHANGKYU KIM, JATIN CHHUGANI, MICHAEL DEISHER, DAE-HYUN KIM, ANTHONY D. NGUYEN, NADATHUR SATISH, MIKHAIL SMELYANSKIY, SRINIVAS CHENNUPATY, PER HAMMARLUND, RONAK SINGHAL, AND PRADEEP DUBEY. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *ISCA*, 2010.
- [61] JINGWEN LENG, TAYLER HETHERINGTON, AHMED ELTANTAWY, SYED GILANI, NAM SUNG KIM, TOR M. AAMODT, AND VIJAY JANAPA REDDI. GPUWattch: Enabling Energy Optimizations in GPGPUs. In *ISCA*, 2013.

- [62] RÉGIS LEVEUGLE, A CALVEZ, PAOLO MAISTRI, AND PIERRE VANHAUWAERT. Statistical fault injection: Quantified error and confidence. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 502–506. European Design and Automation Association, 2009.
- [63] DONG LI, JEFFREY VETTER, AND WEIKUAN YU. Classifying soft error vulnerabilities in extreme-Scale scientific applications using a binary instrumentation tool. pages 1–11, 11 2012.
- [64] G. LI, K. PATTABIRAMAN, C. CHER, AND P. BOSE. Understanding error propagation in gpgpu applications. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 240–251, 2016.
- [65] GUANPENG LI, SIVA KUMAR SASTRY HARI, MICHAEL SULLIVAN, TIMOTHY TSAI, KARTHIK PATTABIRAMAN, JOEL EMER, AND STEPHEN W. KECKLER. Understanding error propagation in deep learning neural network (dnn) accelerators and applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, New York, NY, USA, 2017. Association for Computing Machinery.
- [66] QINJIAN LI, CHENGWEN ZHONG, KAIYONG ZHAO, XINXIN MEI, AND XIAOWEN CHU. Implementation and analysis of AES encryption on GPU. In *HPCC-ICESS*, 2012.
- [67] XUN LI, VINEETH KASHYAP, JASON K. OBERG, MOHIT TIWARI, VASANTH RAM RAJARATHINAM, RYAN KASTNER, TIMOTHY SHERWOOD, BEN HARDEKOPF, AND FREDERIC T. CHONG. Sapper: A language for hardware-level security policy enforcement. In *ASPLOS*, 2014.
- [68] ZHEN LIN, UTKARSH MATHUR, AND HUIYANG ZHOU. Scatter-and-Gather Revisited: High-Performance Side-Channel-Resistant AES on GPUs. In *GPGPU*, 2019.

- [69] ERIK LINDHOLM, JOHN NICKOLLS, STUART OBERMAN, AND JOHN MONTRYM. Nvidia tesla: A unified graphics and computing architecture. *Micro, IEEE*, 28(2):39–55, 2008.
- [70] FANGFEI LIU AND RUBY B. LEE. Random Fill Cache Architecture. In *MICRO*, 2014.
- [71] FANGFEI LIU, Y. YAROM, QIAN GE, G. HEISER, AND R.B. LEE. Last-Level Cache Side-Channel Attacks are Practical. In *S&P*, 2015.
- [72] GU LIU, HONG AN, WENTING HAN, GUANG XU, PING YAO, MU XU, XIURUI HAO, AND YAOBIN WANG. A program behavior study of block cryptography algorithms on GPGPU. In *FCST*, 2009.
- [73] Q. LU, K. PATTABIRAMAN, M. S. GUPTA, AND J. A. RIVERS. SDCTune: A model for predicting the SDC proneness of an application for configurable protection. In *2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 1–10, 2014.
- [74] Y. LUO, S. GOVINDAN, B. SHARMA, M. SANTANIELLO, J. MEZA, A. KANSAL, J. LIU, B. KHESSIB, K. VAID, AND O. MUTLU. Characterizing Application Memory Error Vulnerability to Optimize Datacenter Cost via Heterogeneous-Reliability Memory. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 467–478, 2014.
- [75] ABDULRAHMAN MAHMOUD, SIVA KUMAR SASTRY HARI, MICHAEL B. SULLIVAN, TIMOTHY TSAI, AND STEPHEN W. KECKLER. 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*. IEEE Press, 2018.

- [76] STEFAN MANGARD. Hardware countermeasures against dpa—a statistical analysis of their effectiveness. In *Cryptographers' Track at the RSA Conference*, pages 222–235. Springer, 2004.
- [77] CATELLO DI MARTINO, ZBIGNIEW KALBARCZYK, RAVISHANKAR K. IYER, FABIO BACCANICO, JOSEPH FULLOP, AND WILLIAM KRAMER. Lessons learned from the analysis of system failures at petascale: The case of blue waters. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '14, page 610–621, USA, 2014. IEEE Computer Society.
- [78] FREDERIC P. MILLER, AGNES F. VANDOME, AND JOHN MCBREWSTER. *Advanced Encryption Standard*. Alpha Press, 2009.
- [79] SHUBU MUKHERJEE. Architecture design for soft errors. Morgan Kaufmann, Burlington, 2008.
- [80] HODA NAGHIBIJOUYBARI, KHALED N KHASAWNEH, AND NAEL ABU-GHAZALEH. Constructing and characterizing covert channels on GPGPUs. In *MICRO*, 2017.
- [81] HODA NAGHIBIJOUYBARI, AJAYA NEUPANE, ZHIYUN QIAN, AND NAEL ABU-GHAZALEH. Rendered Insecure: GPU Side Channel Attacks are Practical. In *CCS*, 2018.
- [82] MICHAEL NEVE AND JEAN-PIERRE SEIFERT. Advances on access-driven cache attacks on aes. In *Selected Areas in Cryptography*, volume 4356, pages 147–162. Springer, 2006.
- [83] SAMUEL NEVES AND FILIPE ARAUJO. On the performance of GPU public-key cryptography. In *ASAP*, pages 133–140, 2011.
- [84] BIN NIE, ADWAIT JOG, AND EVGENIA SMIRNI. Characterizing Accuracy-Aware Resilience of GPGPU Applications. In *20th IEEE/ACM International Symposium*

- on Cluster, Cloud and Internet Computing, CCGRID 2020, Melbourne, Australia, May 11-14, 2020*, pages 111–120, 2020.
- [85] BIN NIE, DEVESH TIWARI, SAURABH GUPTA, EVGENIA SMIRNI, AND JAMES H. ROGERS. A large-scale study of soft-errors on GPUs in the field. In *2016 IEEE International Symposium on High Performance Computer Architecture, HPCA 2016, Barcelona, Spain, March 12-16, 2016*, pages 519–530, 2016.
- [86] BIN NIE, JI XUE, SAURABH GUPTA, CHRISTIAN ENGELMANN, EVGENIA SMIRNI, AND DEVESH TIWARI. Characterizing Temperature, Power, and Soft-Error Behaviors in Data Center Systems: Insights, Challenges, and Opportunities. In *25th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS 2017, Banff, AB, Canada, September 20-22, 2017*, pages 22–31, 2017.
- [87] BIN NIE, JI XUE, SAURABH GUPTA, TIRTHAK PATEL, CHRISTIAN ENGELMANN, EVGENIA SMIRNI, AND DEVESH TIWARI. Machine Learning Models for GPU Error Prediction in a Large Scale HPC System. In *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018, Luxembourg City, Luxembourg, June 25-28, 2018*, pages 95–106, 2018.
- [88] BIN NIE, LISHAN YANG, ADWAIT JOG, AND EVGENIA SMIRNI. Fault Site Pruning for Practical Reliability Analysis of GPGPU Applications. In *51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018*, pages 749–761, 2018.
- [89] N. NISHIKAWA, K. IWAI, AND T. KUROKAWA. High-Performance Symmetric Block Ciphers on CUDA. In *ICNC*, 2011.
- [90] A. NUKADA, H. TAKIZAWA, AND S. MATSUOKA. Nvcr: A transparent checkpoint-restart library for nvidia cuda. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 104–113, 2011.

- [91] NVIDIA. Computational finance.
- [92] NVIDIA. Computational finance.
- [93] NVIDIA. How to harness big data for improving public health.
<http://www.govhealthit.com/news/how-harness-big-data-improving-public-health>.
- [94] NVIDIA. Parabricks.
- [95] NVIDIA. Programming Guide.
- [96] NVIDIA. Researchers deploy gpus to build world's largest artificial neural network. *<http://nvidianews.nvidia.com/Releases/Researchers-Deploy-GPUs-to-Build-World-s-Largest-Artificial-Neural-Network-9c7.aspx>.*
- [97] NVIDIA. CUDA C/C++ SDK Code Samples, 2011.
- [98] NVIDIA. Jp morgan speeds risk calculations with nvidia gpus. 2011.
- [99] NVIDIA. NVIDIA A100 Tensor Core GPU Architecture, 2020.
- [100] DAG A. OSVIK, ADI SHAMIR, AND ERAN TROMER. Cache attacks and countermeasures: the case of AES. *CT-RSA*, 2006.
- [101] DAG ARNE OSVIK, ADI SHAMIR, AND ERAN TROMER. Cache attacks and countermeasures: The case of aes. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology*, CT-RSA'06, 2006.
- [102] DAN PAGE. Partitioned cache architecture as a side-channel defense mechanism. In *Cryptology ePrint Archive, Report 2005/280*, 2005.
- [103] SEUNG IN PARK, SEAN P PONCE, JING HUANG, YONG CAO, AND FRANCIS QUEK. Low-cost, high-speed computer vision using nvidia's cuda architecture. In *Applied Imagery Pattern Recognition Workshop, 2008. AIPR'08. 37th IEEE*, pages 1–7. IEEE, 2008.

- [104] ADAM PASZKE, SAM GROSS, FRANCISCO MASSA, ADAM LERER, JAMES BRADBURY, GREGORY CHANAN, TREVOR KILLEEN, ZEMING LIN, NATALIA GIMELSHEIN, LUCA ANTIGA, ALBAN DESMAISON, ANDREAS KOPF, EDWARD YANG, ZACHARY DEVITO, MARTIN RAISON, ALYKHAN TEJANI, SASANK CHILAMKURTHY, BENOIT STEINER, LU FANG, JUNJIE BAI, AND SOUMITH CHINTALA. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.
- [105] KEXIN PEI, YINZHI CAO, JUNFENG YANG, AND SUMAN JANA. Deepxplore: Automated whitebox testing of deep learning systems. In *proceedings of the 26th Symposium on Operating Systems Principles*, pages 1–18, 2017.
- [106] BHARATH PICHAI, LISA HSU, AND ABHISHEK BHATTACHARJEE. Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces. In *ASPLOS*, 2014.
- [107] ROBERTO DI PIETRO, FLAVIO LOMBARDI, AND ANTONIO VILLANI. Cuda leaks: A detailed hack for cuda and a (partial) fix. *ACM Trans. Embed. Comput. Syst.*, 2016.
- [108] LOUIS-NOËL POUCHET. Polybench: the polyhedral benchmark suite. 2012.
- [109] GUILLEM PRATX AND LEI XING. Gpu computing in medical physics: A review. *Medical physics*, 38:2685, 2011.
- [110] F. G. PREVILON, C. KALRA, D. TIWARI, AND D. R. KAEI. Pcfi: Program counter guided fault injection for accelerating gpu reliability assessment. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 308–311, 2019.

- [111] MOINUDDIN K. QURESHI. New Attacks and Defense for Encrypted-address Cache. In *ISCA*, 2019.
- [112] MINSOO RHU, MICHAEL SULLIVAN, JINGWEN LENG, AND MATTAN EREZ. A Locality-Aware Memory Hierarchy for Energy-Efficient GPU Architectures. In *MICRO*, 2013.
- [113] TIMOTHY G. ROGERS, MIKE O’CONNOR, AND TOR M. AAMODT. Cache-Conscious Wavefront Scheduling. In *MICRO*, 2012.
- [114] TIMOTHY G. ROGERS, MIKE O’CONNOR, AND TOR M. AAMODT. Divergence-Aware Warp Scheduling. In *MICRO*, 2013.
- [115] OLGA RUSSAKOVSKY, JIA DENG, HAO SU, JONATHAN KRAUSE, SANJEEV SATHEESH, SEAN MA, ZHIHENG HUANG, ANDREJ KARPATHY, ADITYA KHOSLA, MICHAEL BERNSTEIN, ALEXANDER C. BERG, AND LI FEI-FEI. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.
- [116] IVY SCHMERKEN. Wall street accelerates options analysis with gpu technology. *2008-11-07)[2009-11-02]*. <http://wallstreetandtech.com/technology-risk-management/showArticle.jhtml>, 2009.
- [117] A. SETHIA, D. A. JAMSHIDI, AND S. MAHLKE. Mascar: Speeding up GPU warps by reducing memory pitstops. In *HPCA*, 2015.
- [118] ANKIT SETHIA, GANESH DASIKA, MEHRZAD SAMADI, AND SCOTT MAHLKE. APOGEE: Adaptive Prefetching on GPUs for Energy Efficiency. In *PACT*, 2013.
- [119] KAREN SIMONYAN AND ANDREW ZISSERMAN. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations*, 2015.

- [120] V. SRIDHARAN AND D. LIBERTY. A study of dram failures in the field. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2012.
- [121] VILAS SRIDHARAN, NATHAN DEBARDELEBEN, SEAN BLANCHARD, KURT B. FERREIRA, JON STEARLEY, JOHN SHALF, AND SUDHANVA GURUMURTHI. Memory errors in modern systems: The good, the bad and the ugly. In *In Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15*, pages 297–310, 2015.
- [122] SAM S. STONE, JUSTIN P. HALDAR, STEPHANIE C. TSAO, WEN MEI W. HWU, BRADLEY P. SUTTON, AND ZHI-PEI LIANG. Accelerating advanced MRI reconstructions on GPUs. *J. Parallel Distrib. Comput.*, 68(10):1307–1318, 2008.
- [123] CHRISTIAN SZEGEDY, VINCENT VANHOUCKE, SERGEY IOFFE, JONATHON SHLENS, AND ZBIGNIEW WOJNA. Rethinking the inception architecture for computer vision. *CoRR*, abs/1512.00567, 2015.
- [124] KRIS TIRI, ONUR ACHIÇMEZ, MICHAEL NEVE, AND FLEMMING ANDERSEN. An analytical model for time-driven cache attacks. In *International Workshop on Fast Software Encryption*, pages 399–413. Springer, 2007.
- [125] DEVESH TIWARI, SAURABH GUPTA, GEORGE GALLARNO, JIM ROGERS, AND DON MAXWELL. Reliability lessons learned from gpu experience with the titan supercomputer at oak ridge leadership computing facility. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '15*, New York, NY, USA, 2015. Association for Computing Machinery.
- [126] S. TSELONIS AND D. GIZOPOULOS. Gufi: A framework for gpus reliability assessment. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 90–100, 2016.

- [127] ORESTE VILLA, MARK STEPHENSON, DAVID NELLANS, AND STEPHEN W. KECKLER. Nvbit: A dynamic binary instrumentation framework for nvidia gpus. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 372–383, New York, NY, USA, 2019. Association for Computing Machinery.
- [128] STAVROS VOLOS, KAPIL VASWANI, AND RODRIGO BRUNO. Graviton: Trusted Execution Environments on GPUs. In *OSDI*, 2018.
- [129] J. WADDEN, A. LYASHEVSKY, S. GURUMURTHI, V. SRIDHARAN, AND K. SKADRON. Real-world design and evaluation of compiler-managed gpu redundant multithreading. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 73–84, 2014.
- [130] XIN WANG AND WEI ZHANG. Cracking Randomized Coalescing Techniques with An Efficient Profiling-Based Side-Channel Attack to GPU. In *HASP*, 2019.
- [131] ZHENGHONG WANG AND RUBY B. LEE. Covert and side channels due to processor architecture. In *ACSAC*, pages 473–482, 2006.
- [132] ZHENGHONG WANG AND RUBY B. LEE. New cache designs for thwarting software cache-based side channel attacks. In *ISCA*, 2007.
- [133] ZHENGHONG WANG AND RUBY B. LEE. A Novel Cache Architecture with Enhanced Performance and Security. In *MICRO*, 2008.
- [134] C. WU, L. ZHANG, Q. LI, Z. FU, W. ZHU, AND Y. ZHANG. Enabling flexible resource allocation in mobile deep learning systems. *IEEE Transactions on Parallel and Distributed Systems*, 30(2):346–360, 2019.
- [135] QIUMIN XU, HODA NAGHIBIJOUYBARI, SHIBO WANG, NAEL B. ABU-GHAZALEH, AND MURALI ANNAVARAM. GPUGuard: mitigating contention based side and covert channel attacks on GPUs. In *ICS*, 2019.

- [136] MENGJIA YAN, BHARGAVA GOPIREDDY, THOMAS SHULL, AND JOSEP TORRELLAS. Secure Hierarchy-Aware Cache Replacement Policy (SHARP): Defending Against Cache-Based Side Channel Attacks. In *ISCA*, pages 347–360, 2017.
- [137] LISHAN YANG, BIN NIE, ADWAIT JOG, AND EVGENIA SMIRNI. Enabling Software Resilience in GPGPU Applications via Partial Thread Protection. *CoRR*, abs/2103.02825, 2021.
- [138] LISHAN YANG, BIN NIE, ADWAIT JOG, AND EVGENIA SMIRNI. Practical Resilience Analysis of GPGPU Applications in the Presence of Single- and Multi-Bit Faults. *IEEE Trans. Computers*, 70(1):30–44, 2021.
- [139] LISHAN YANG, BIN NIE, ADWAIT JOG, AND EVGENIA SMIRNI. SUGAR: Speeding Up GPGPU Application Resilience Estimation with Input Sizing. *Proc. ACM Meas. Anal. Comput. Syst.*, 5(1):1:1–1:29, 2021.
- [140] WENZHU YANG, LILEI JIN, WANG SILE, ZHENCHAO CUI, XIANGYANG CHEN, AND LIPING CHEN. Thinning of convolutional neural network with mixed pruning. *IET Image Processing*, 13, 01 2019.
- [141] F. YAO, M. DOROSLOVACKI, AND G. VENKATARAMANI. Are Coherence Protocol States Vulnerable to Information Leakage? In *HPCA*, pages 168–179, 2018.
- [142] YUVAL YAROM AND KATRINA FALKNER. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack. In *USENIX Security*, pages 719–732, 2014.
- [143] AMIR YAZDANBAKHSI, DIVYA MAHAJAN, PEJMAN LOTFI-KAMRAN, AND HADI ESMAEILZADEH. Axbench: A multiplatform benchmark suite for approximate computing. *IEEE Design & Test*, PP:1–1, 11 2016.

- [144] DANFENG ZHANG, YAO WANG, G. EDWARD SUH, AND ANDREW C. MYERS. A Hardware Design Language for Timing-Sensitive Information-Flow Security. In *ASPLOS*, 2015.
- [145] YINQIAN ZHANG, ARI JUELS, MICHAEL K. REITER, AND THOMAS RISTENPART. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *CCS*, pages 990–1003, 2014.
- [146] YANQI ZHOU, SAMEER WAGH, PRATEEK MITTAL, AND DAVID WENTZLAFF. Camouflage: Memory traffic shaping to mitigate timing attacks. In *HPCA*, pages 337–348, 2017.

VITA

Gurunath Kadam

Gurunath Kadam is a PhD Candidate in the Department of Computer Science at The College of William & Mary. His PhD advisor is Prof. Adwait Jog. His research focuses on the security and reliability of emerging computing systems and general-purpose accelerators, such as GPUs. His PhD research appeared in DSN 2021, HPCA 2020 and HPCA 2018. Previously, he received his Bachelor of Engineering in Electrical Engineering from the University of Mumbai, India in 2006. He received his Master of Science in Information and Communication Engineering from the Technical University of Darmstadt, Germany in 2012. He worked as a research intern at Intel Labs, OR in the fall of 2018.