2023

Improving the Performance, Energy Efficiency and Security of GPUs

Xin Wang

IMPROVING THE PERFORMANCE, ENERGY EFFICIENCY AND SECURITY OF GPUS

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at Virginia Commonwealth University.

by

XIN WANG, PH.D. VIRGINIA COMMONWEALTH UNIVERSITY

September 2015 to August 2023

Director: Wei Zhang, Ph.D., Ruixin Niu, Ph.D.,

Professor, Department of Computer Engineering

Virginia Commonwealth University

Richmond, Virginia

August, 2023

Acknowledgements

I thank my advisors, Prof. Wei Zhang and Prof. Ruixin Niu. I am graceful for Prof. Wei Zhang offering me an opportunity of working with him in this Ph.D. program, where I received excellent guidance and resourceful feedback from him. Prof. Zhang builds a path for young academics to succeed. He is always supportive of me to explore innovative research ideas and encourage me to attempt an exclusive career path. I give my thanks to Prof. Ruixin Niu who helped me a lot on my dissertation work. He patiently reviewed my work and went thought every detail of the dissertation with me. With his highly suggestive feedback and comments, I am able to improve the quality of my dissertation research. I appreciate my committee members, Dr. Afroditi V. Filippas, Dr. Changqing Luo, Dr. Preetam Ghosh and Dr. Yuichi Motai for their constructive comments on my research plan proposal. I also thank their present of my proposal and dissertation defense.

I thank my friends Tong Liu, Ping Huang, Huangfu Yijie and Hao Wen for staying around with me during the my study career in Richmond. Their accompany means a lot to me and graces my life.

I thank my girl friend Chloe whose support and encouragement gives me strength and makes me fearless to deal with my struggle.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Abstract

IMPROVING THE PERFORMANCE, ENERGY EFFICIENCY AND SECURITY OF GPUS

By Xin Wang, PH.D. VIRGINIA COMMONWEALTH UNIVERSITY

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at Virginia Commonwealth University.

Virginia Commonwealth University, 2023.

Director: Wei Zhang, Ph.D., Ruixin Niu, Ph.D.,
Professor, Department of Computer Engineering

The work in this dissertation achieves to enhance the performance, energy-efficiency, and security of the GPUs. We noticed that, as the demand of hardware resources keeps rising in GPUs, the energy consumption becomes unaffordable and places barriers for further performance boost. To resolve this issue, we have proposed several novel GPU micro-architectures that are able to assist the GPUs to execute in an energy-efficient manner. They also provide the potential for further performance enhancement in GPUs. Firstly, we proposed a GPU register packing scheme that stores multiple narrow-width operands to a single register to save register file resources. The unoccupied registers then can be turned off to save leakage energy consumption or used to accommodate additional threads to improve the performance. Secondly, we presented a drowsy register file mechanism which can improve the energy-efficiency by putting the unused register into a low power state. Thirdly, we introduced a GPU computing units power-gating strategy by noticing the considerable idleness of the computing units. The power-gating strategy achieves a

remarkable overall GPU energy reduction. Finally, in order to raise awareness about the GPU's vulnerability to side-channel attacks among the researchers, we presented our research on attacking the AES algorithms that are deployed onto the GPUs and discussed appropriate countermeasures.

There are four projects in this dissertation.

First, we propose OWAR, an Operand-Width-Aware Register packing mechanism for GPU energy saving. We first notice that a large percentage of computed results actually has fewer significant bits compared to the full width of a 32-bit register for many GPGPU applications. OWAR utilizes a GPU register packing scheme to dynamically exploit narrow-width operands and pack multiple operands into a single full width register, thereby saving RF resources. In order to efficiently use RF, OWAR then employs a power gating method to shut down unused register sub-arrays for reducing dynamic and leakage energy consumption of RF. The register operation to narrow width operands is able to be completed with fewer number of RF accesses and therefore the dynamic energy is further reduced. Finally, by observing that RF resource is still insufficient to enable all thread level parallelism (TLP) and the lack of RF resources can hurt performance by limiting the occupancy of GPU threads, with the help of RF usage optimized by register packing, OWAR allows GPUs to support more TLP through assigning additional thread blocks on SMs (Streaming Multiprocessors) for GPGPU applications that suffer from the deficiency of register resources. The extra TLP opens opportunities for hiding more memory latencies and achieving better performance. The GPGPU's energy consumption then naturally decreases due to the reduction in execution duration. We evaluate OWAR using a batch of representative GPU benchmarks. The experimental results shows that compared to the baseline without optimization, OWAR can reduce the GPGPU's total energy up to 29.6% and 9.5% on average. In addition, OWAR achieves performance improvement

up to 1.97X and 1.18X on average.

Next, we work on the drowsy technique to reduce the RF leakage energy consumption. We introduce three drowsy policies and evaluate their effectiveness on leakage energy reduction. In the first drowsy policy called immediate sleep (Drowsy-IS), registers keep staying in the drowsy mode unless they are accessed and then they are put into the drowsy mode again immediately to minimize the leakage energy consumption. The second policy named temporary awake (Drowsy-TA) holds the registers in the normal mode for a certain period after being accessed to wait for the next access. The registers are placed into the drowsy mode until that period expires without any access activity. Thirdly, we propose an adaptive policy which identifies the re-access interval (Drowsy-RI) for each register at run-time and let registers wait for the predicted intervals before putting them into the drowsy mode. The experimental results show that compared to the baseline RF, Drowsy-IS achieves 91.7% RF leakage energy reduction on average at the cost of 4.4% performance degradation. Drowsy-TA leads to only negligible performance overhead, and 82.8% leakage energy reduction. By balancing the energy saving and the performance overhead, Drowsy-RI saves more RF leakage energy (87.3%) than Drowsy-TA along with less performance degradation (2.7%) than Drowsy-IS.

Thirdly, we examine the distribution and the length of execution units idle cycles for several typical GPGPU applications to direct the energy-saving strategies to capture potential execution units power-gating opportunities. We record the idle durations of the execution units for SMs (Streaming Multiprocessors) including integer units and floating units in SPs (Streaming Processors) and SFUs (Special Function Units). Second, based on the observation of idleness, we study the effectiveness of the execution units power-gating on the leakage energy saving with two simple policies, the immediate power-gating (IPG) and idle detect power-gating (ID-PG). We

examine the polices with various parameter settings in order to offer insights on possible gains and losses from the power-gating techniques to enable smarter strategies in future research. The experimental results show that both policies can achieve satisfactory leakage energy saving on execution units. The immediate power-gating can reduct the execution units leakage energy by 84.3% when the break even time is set to 5 cycles and the idle detect power-gating can save 67.1% of the total execution units leakage energy even if the break even time goes up to 20 cycles. 3)

Finally, we propose two Side-Channel Attacks (SCAs) to demonstrate that ignoring security issues and naively moving security services onto GPUs can expose adversaries fatal vulnerabilities to thieve critical information. First, we proposed to leverage a profiling-based side-channel attack (pSCA) to expose GPUs' side-channel vulnerability and the weakness of security services provided by GPUs. Our results show that GPUs are particularly vulnerable to profiling-based side-channel attacks and need to be protected against side-channel threats. Especially, for AES-128, the proposed method can recover all key bytes in less than 1 minute, outperforming all prior SCAs we know. Aiming at protecting GPUs against these SCAs, the Randomized Coalescing (RCoal) techniques are proposed with proven effectiveness on security improvement. However, our pSCA is still able to rebuild the secure key of a GPUsupported AES algorithm under the RCoal protection in a reasonable duration without the detailed information of the RCoal configuration. Second, we propose a Profiling-Assisted Correlation-based Side-Channel Attack (pacSCA) to demonstrate that the private information processed by GPUs is under serious risk before the security issues are properly addressed. We further prove that our pacSCA can still achieve information leakage even though the Rcoal techniques have been applied which is a state-of-the-art countermeasure to the side-channel attacks. The results show that the proposed SCA can reveal the secure key of the AES-128 algorithm in less than 6

seconds. With the guard of the Rcoal techniques, the pacSCA can obtain the secure key in less than 17 minutes. The purpose of this study is to arouse further researches on GPU side-channel threats and corresponding countermeasures.

CHAPTER 1

INTRODUCTION

1.1 Background

GPUs which served as graphic-oriented computation platform are now transforming to the general-purpose applications such as computation-intensive and data-parallel scientific computing programs [1, 2, 3]. Modern GPUs also called general-purpose graphics processing units (GPGPUs) are equipped with a large number of computation units enabling to process thousands of threads or more concurrently by following a single instruction multiple data (SIMD) pattern. The software layer which is supported by GPU fitted programming environment such as NVIDA developed CUDA [4] and AMD provided OpendCL [5] utilizes the GPU's hardware to accelerate the applications tremendously. There are a wide range of existing parallelizable applications formerly running on CPUs can be tailored to GPUs for significant performance benefit. CUDA programming language allows the programmer to define the paralleled portion of an application as several kernels, each consisting of thousands of threads that execute in parallel [4]. One GPGPU application usually contains multiple CUDA kernels organized as a group of thread blocks, also called concurrent thread array (CTA). A thread block can be formed as one-dimensional, two dimensional or three-dimensional thread index based on the data structures, such as a vector, matrix or volume computation domain. The sub-level of the thread block called warp contains 32 threads with consecutive thread IDs. A warp is executed in a single instruction multiple-threads (SIMT) style, where all 32 threads in the same warp execute the same instruction working on different data fragments. Moreover, in

the SIMT execution style, only one PC is handled and a single instruction is fetched and decoded for each warp. However, threads in the same warp can access different memory address and follow different control flow paths. The cooperation of software support and hardware features allow GPU to overlap long latency by utilizing warps in a thread blocks conserving stalled warps context and switching a oldest ready warp. As a result, the performance of paralleled applications can be significantly boosted on GPGPUs.

Keeping increasing the computing units and enlarging the capacity of RF can help boost GPU performance. However, The downside of GPUs achieving ultra high throughput in exchange of the high demand on hardware resources is that energy efficiency turns out to be an issues. With the scaling up of GPGPU's execution units and register files willing to host more thread contexts, both dynamic and leakage energy of GPGPUs keep increasing and become unaffordable. Thus, we need to explore new techniques to exploit and manage the GPU hardware resources, e.g. execution unit, registers and etc., more efficiently to achieve further performance improvement without involving additional resources. Moreover, the hardware resources are demanded for peak performance requirement, however unfortunately many applications can not take full advantage of them leaving a great portion of components in idle wasting energy and thus hurting the energy-efficiency [6, 7, 8, 9, 10, 11]. Operating the GPU components in energy-efficient ways can also offer the opportunities of supporting more hardware resources and thus boosting performance.

Taking advantage of the high throughput offered by GPUs, the security services such as cryptographic application have been increasingly deployed on GPUs to enjoy remarkable performance boosting [12, 13, 14, 15, 16, 17]. Unfortunately primary existing studies concentrate on taking care of identifying vulnerabilities based on mainstream security computing platforms such as CPUs and FPGAs, while the

vulnerabilities exposed on the promising GPU-based security systems have not been thoroughly explored leaving severe vulnerabilities for adversary to pull down the whole security system. Recently, an increasing number of studies have been conducted to explore the security vulnerabilities on GPUs and propose corresponding countermeasures making effort on building a secure environment for GPU computing. Early researches focus on utilizing the software flaws of GPU programming model to leak confidential information [18, 19, 20, 21, 22]. Or, to be more specific, most of these works reveal that GPU drivers don't automatically erase data residing in GPU memory hierarchies including shared memory, global memory, local memory and registers after deallocation leading to enormous risk of leaking sensitive data through reallocating the same memory space to the adversary's program [18, 19, 20, 21].

1.2 GPU Register File Narrow-Width Operands Packing

The first topic is about GPU register file (RF) management. The main aim of this work is to propose a GPU RF management mechanism that offers an energy efficiency by taking advantage of narrow-width operands. Inspired by the narrow-width-aware register packing research on CPUs, we design a GPU register packing scheme called Operand-Width-Aware Register Packing (OWAR) which takes into account of characteristics of GPU's miroarchitecture and RF organization. This novel technique first applies the register packing and register usage prediction methods to dynamically identify narrow-width operands, pack them into a single register and predict the shrunken register usage for the on-coming thread blocks. By register packing and usage prediction, the remaining RF space increases and the demand of future thread blocks on register resources decreases. As a result, more registers become unutilized. OWAR will then turn off the sub-arrays that contain no in-use registers to save both dynamic and leakage energy. Combined with a renaming

table, OWAR is able to map multiple architectural registers that store narrow-width operands to a single physical register in run-time. Consequently, OWAR offers the illusion that extra RF space is available and make the SMs to host more thread blocks. For GPU kernels whose occupancy of threads is limited by RF resources, OWAR then can provides additional TLP that plays a role in hiding memory latencies to improve performance.

1.3 GPU Register File Drowsy Management

The second research topic is about breaking the energy constraint through drowsy RF, which provides an energy efficient way to operate GPGPUs' register file. We introduce three drowsy policies and evaluate their effectiveness on leakage energy reduction. In the first drowsy policy called immediate sleep (Drowsy-IS), registers keep staying in the drowsy mode unless they are accessed and then they are put into the drowsy mode again immediately to minimize the leakage energy consumption. The drawback is that the additional cycles are required to re-active registers from the drowsy mode for each access which may lead to significant performance degradation. By noticing that some registers are re-accessed in high frequency, to avoid re-activation penalty. The second policy named temporary awake (Drowsy-TA) holds the registers in the normal mode for a certain period after being accessed to wait for the next access. The registers are placed into the drowsy mode until that period expires without any access activity. The leakage energy will be wasted if no re-access occurs during the waking cycles. Finally, we propose an adaptive policy which identifies the re-access interval (Drowsy-RI) for each register at run-time and let registers wait for the predicted intervals before putting them into the drowsy mode. The Drowsy-RI is running a performance/energy tradeoff according to the characteristic of re-access intervals. The Drowsy-RI applies TA policy to the registers with short

access interval for the purpose of re-activation penalty elimination. For the registers not being accessed in the long term, IS policy is employed to save leakage energy.

1.4 GPU Execution Units Power-Gating Strategies

Thirdly, we explore the idleness pattern of execution units for different GPU workloads and try to discover the inherent opportunities for power-gating on energy-efficiency enhancement without any re-schedule techniques. The focus of this work is to analyze the inborn natures of the execution units idleness and to show the power of traditional power-gating strategies on GPU leakage energy saving. The goal of this research is to guide to the future GPU energy studies regarding to execution units power-gating. The advantage of conducting a study on original pattern is that no additional microarchitecture involvement except basic counters makes the GPU architecture unchanged and also avoid hardware overhead. In additional, the results depict a good enough energy-saving although no complicated strategies are applied.

1.5 GPU Side-Channel Attacks

The last topic is to discover the vulnerabilities of GPGPU architecture and discuss corresponding countermeasure. We propose two GPU side-channel attacks, profiling-based side-channel attack (pSCA) and profiling-assisted correlation-based side-channel Attack (pacSCA). Both SCAs can reveal the secure key of the AES-128 algorithm running on GPGPUs. We first propose a novel profiling-based side-channel attack that information leaked from GPU performance profiling can be extracted when executing on an SIMT-based GPU to fully recover the encryption secret key. Our profiling-based strategy accomplish straightforward key recovery procedure by sampling the exact number of unique cache line requests during the run-time and simply checking all 256 possibilities for each byte of all 16-byte AES key to determine

the correct answer. As compared to the existing SCAs, the number of samples and time needed for recovering the key are dramatically reduced while the accuracy can be always guaranteed. Moreover, the profiling-based side-channel attack has great scalability, as the profiling time only increases slightly with no additional samples required when the length of the AES key scales up. We further propose a profiling-assisted correlation-based side-channel attack. By comparison, pacSCA outperforms pSCA on the total key recovery time and is more powerful to resist both hardware and software defense mechanisms.

1.6 Dissertation Organization

The rest of the dissertation is organized as follows. Chapter 2 briefly introduces the GPU architecture and GPGPU programming model. Chapter 3, Chapter 4, Chapter 5 and Chapter 6 introduce the four studies included in this dissertation work. The objective of these four research topics is to establish a high-performance, energy-efficient, and highly secure GPU computing environment. The mechanisms proposed in Chapter 3, Chapter 4 and Chapter 5 attempt to improve the energy-efficiency of the GPUs. Moreover, the schemes discussed in Chapter 3 and Chapter 5 can also help the performance. Furthermore, Chapter 6 exposes the vulnerability of modern GPUs to the side-channel attacks and proposes several countermeasures to protect the applications running on the GPUs. Specifically, Chapter 3 introduces the GPU register file narrow-width operands packing technique and how it can help the GPU performance and energy-efficiency. Chapter 4 illustrates the three drowsy policies which manage the GPU register file in an energy-efficient style. Chapter 5 describes a novel power-gating strategy which shutdown the spared GPU execution units to achieve leakage energy saving. Chapter 6 introduces two GPU side-channel attacks and the countermeasures are also discussed. Finally, Chapter 7 concludes the

dissertation.

CHAPTER 2

GPU ARCHITECTURE AND GPUGPU PROGRAMMING MODEL

2.1 GPU Architecture

We evaluate our schemes on the GPU architecture that follows the Fermi architecture [23]. Figure 1 shows the overview of a typical Fermi GPU architecture. It consists of 15 GPU cores called Streaming Multiprocessors (SMs) where each SM core contains 32 single instruction multiple data execution units, 16 load/store units and 4 special function units (SFUs). Each SM core owns two warp schedulers and two instruction dispatch units, enabling to issue two independent instructions from two different warps. The 32 execution units are assigned to two shader processors (SP) each containing 16 execution lanes called SIMT lanes. Since the execution units are operating at double clock frequency of the SMs, 32 threads can then be running upon a single SP concurrently. All SMs have their own private L1 data cache, read-only texture cache, constant cache and software-managed shared memory(scratchpad memory). Total size of L1 cache and shard memory is 64KB and they can be configure to 16KB L1 cache and 48KB shared memory and vice verse through software. The read-only texture cache loads data from off-chip texture memory and is designed to identify the spatial locality of memory access patterns. The read-only constant cache works with the off-chip constant memory and is utilized to optimize the data sharing among all the threads in a warp. All SMs share an on-chip unified L2 cache partitioned into 6 tiles and an off-chip global memory. The SMs and the shared L2 cache are connected via an on-chip network. The private L1 cache per SM and shared L2 cache and global memory cooperate to perform fast memory accesses. Although

GPU architecture inherits similar memory hierarchy from CPU architecture, GPU caches result in much smaller size and much higher bandwidth than CPU caches. Moreover, Unlike CPUs which primarily count on caches to reduce memory latency, GPUs rely on massive TLP to hide memory latency and increase throughput. To support fast and low-cost context switching for massive concurrently running threads, a large number of registers are necessary in GPUs. The massive TLP supported by a large size of register files contribute most performance enhancement and achieve high throughput. On the other hand, the functionality of GPU caches on boosting performance is more like a complement instead of an essential.

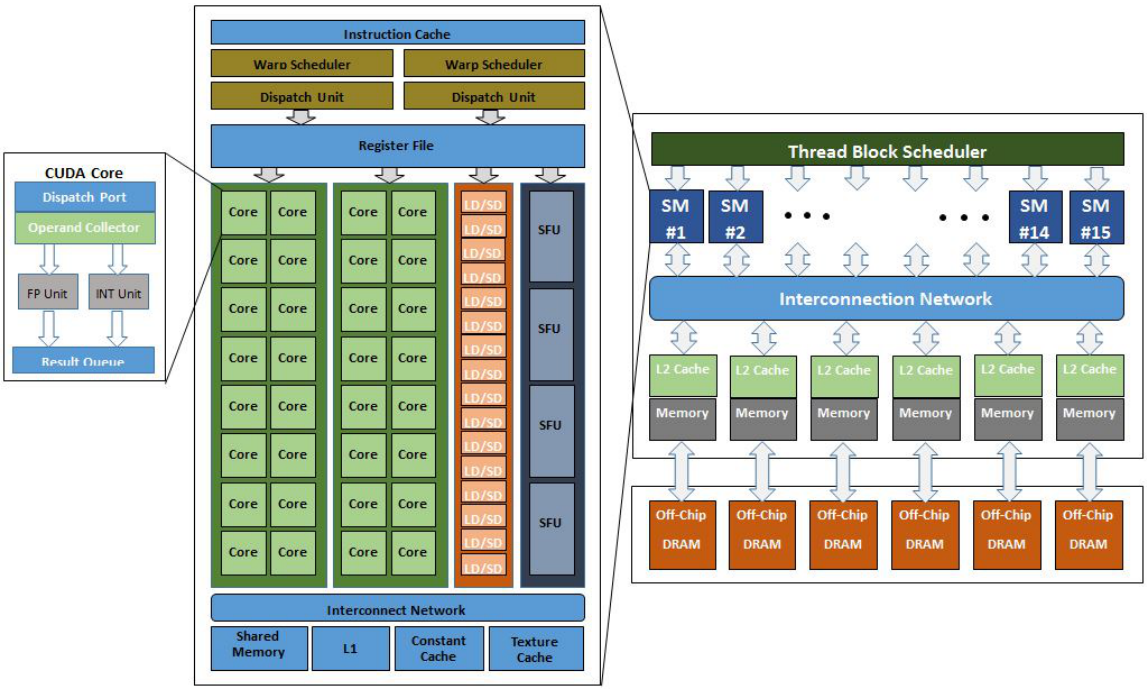


Fig. 1. Baseline GPU Architecture

2.2 CUDA Programming Model

Both GPUs evaluated in this work are CUDA supported. CUDA programming language allows the programmer to define C functions as several kernels which con-

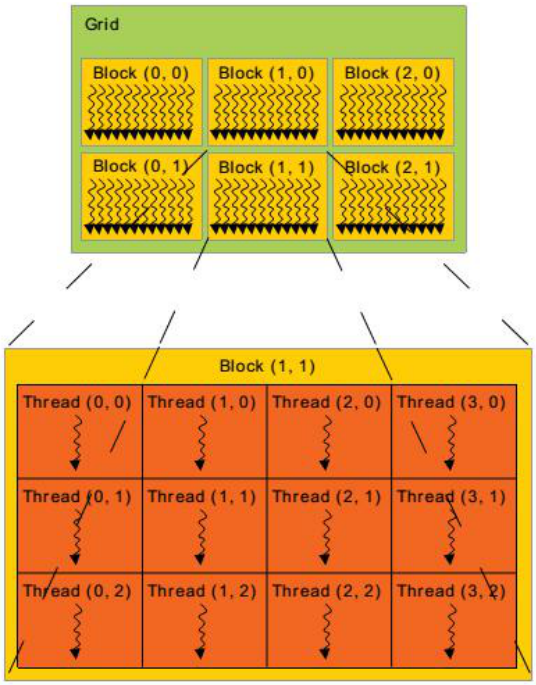


Fig. 2. Grid of Thread Blocks[4]

sist of thousands of threads that are executing in parallel [4]. Each thread within a CUDA kernel is marked with a assigned unique thread ID which is accessible through the built-in threadIdx variable. GPGPU applications always contain multiple kernels that contain a group of thread blocks. A thread block is formed by one-dimensional, two-dimensional or three-dimensional thread index allowing a vector, matrix or volume computation domain as illustrated by Figure 2. Every 32 Threads within the same thread block with consecutive thread IDs are grouped as a warp. A warp is executed in a single instruction multiple-threads (SIMT) way and has only one PC (Program Counter). The warps in a thread blocks allow GPU to overlap long latency by conserving stalled warps context and switching a oldest ready warp. The NVIDA officially set limitation for the number of threads running concurrently on a SM. For Fermi architecture, up to 48 active warps or 8 thread blocks can be hosted per SM,

this is in total 1536 active threads per SM. On the other hand, each thread consumes a portion of hardware resources such as register files and shared memory to support the execution environment and conserve execution data and the available hardware resources of a SM also limit the number of concurrent threads per SM.

Although a single warp only maintains one PC, threads in the same warp can access different memory address or follow different control flow paths. To take care of all 32 memory accesses initiated by threads in the same warp which may point to different addresses, a coalescing unit accepts all these memory requests and assign them to different cache lines according to their address. Memory requests with addresses located at the same cache line then can be merged generating unique cache line requests which then are forwarded to L1 cache controller for further processing. The cache line request if finds the data in L1 cache turns out to be a L1 cache hit and requested data can be immediately returned. In case of a L1 cache miss, the status of cache miss is forwarded to MSHR (miss-status holding registers) to check if the same request from another warp is already issued and still in progress. If the same request exists in the MSHRs, a entry to the MSHR corresponding to the requested cache line is allocated to make sure that returned cache line is offered to both warps. If the request does't exist in the MSHRs, a new MSHR entry is dispatched to store the missing cache line status. Unlike CPUs which are capable of dealing with complex branch prediction logic, GPUs follow simple logic only and are unable to predict branches for the purpose of supporting overwhelming multi-threading. Since no branch prediction supported in GPUs, the execution paths are controlled independently for each thread by an active mask vector, where the active mask vector will tell whether the branch instruction is taken or not taken for all the threads in a warp and direct them to different paths. The threads with the corresponding bit being set in the active mask vector will be executed and retired, while other threads with reset bits will discard

the execution results. The taken and not-taken paths are executed sequentially and all the threads will be joined at the convergence point automatically.

CHAPTER 3

GPU REGISTER FILE NARROW-WIDTH OPERANDS PACKING

3.1 Introduction

Taking advantage of the ability of processing multiple data in parallel, GPUs have been increasingly used to accelerate general purpose applications like compute-intensive data-parallel scientific computing programs [1, 2, 3]. GPGPUs execute hundreds or even thousands of threads concurrently to ensure high throughput. To support massive number of simultaneous threads and rapidly context switching between these threads, a huge number of compute units and a large RF are inevitable. GPU design trend shows that GPU performance improvement continues to rely on increasing the hardware resources and operating them at higher frequency to accommodate a greater number of high performance active threads. Indeed, keeping enlarging the capacity of RF and increasing computing resources can help boost GPU performance. Unfortunately, persistently accumulating hardware resources negatively exposes GPUs under extremely pressure of power consumption and chip area [24, 25, 26]. In particular, RF with the design of massive high leakage transistors has be demonstrated to contribute significantly to GPU's total energy consumption. Thus, we need to explore new techniques to exploit and manage GPU registers more efficiently.

Since RF is one of the largest structures on the GPU chip, we propose a GPU register packing scheme which takes into account the GPU microarchitecture and GPU RF organization. This novel technique dynamically detects and profiles narrow-width operands to pack multiple narrow-width operands into a single register and

predicts the packed register usage. Based on the profiling, a smaller register usage is predicted for the on-coming thread blocks. As a result, more registers become unutilized. OWAR will then turn off the sub-arrays that contain no in-use registers to save both dynamic and leakage energy. Combined with a renaming table, OWAR is able to map multiple architectural registers that store narrow-width operands to a single physical register in run-time. Consequently, OWAR offers the illusion that extra RF space is available and make the SMs to host more thread blocks. For GPU kernels whose occupancy of threads is limited by RF resources, OWAR then can provides additional TLP that plays a role in hiding memory latencies to improve performance.

3.2 Related Work

The narrow-width operand packing technique is first employed on CPUs, especially multi-threaded CPU processors to mitigate RF pressure. By noticing that many operands called narrow-width operands have fewer significant bits compared to the full width of a 32-bit register, several work [27, 28, 29, 30, 31] propose to detect them and merge multiple narrow-width operands. The narrow-width operand packing technique is then designed to save power consumption [29] and improve performance [28, 29] as well as register file reliability [30, 31].

To the best of our knowledge, there are no GPU-exclusive register packing technique existing so far. However several work has introduced similar techniques [32, 33, 34, 7]. Observing that the register values of threads within the same warp are similar, Lee et al.[32] presents Warped-Compression, a warp-level register compression scheme which removes data redundancy of register values through register compression to enable power reduction opportunities. This technique saves 25% of the total register file power consumption. Tan et al. [33] proposed the Narrow-Width-Aware register write

back method which combines two narrow-width writes to share data bus resource and hence enhance the performance. Gilani et al. [34] notice that many operands require considerably fewer bits for accurate representation and computations. They propose a sliced GPU architecture which is much alike the method designed for CPUs in [27]. Their approach that improves performance of the GPU up to 15% by dual-issuing instructions to two 16-bit execution slices.

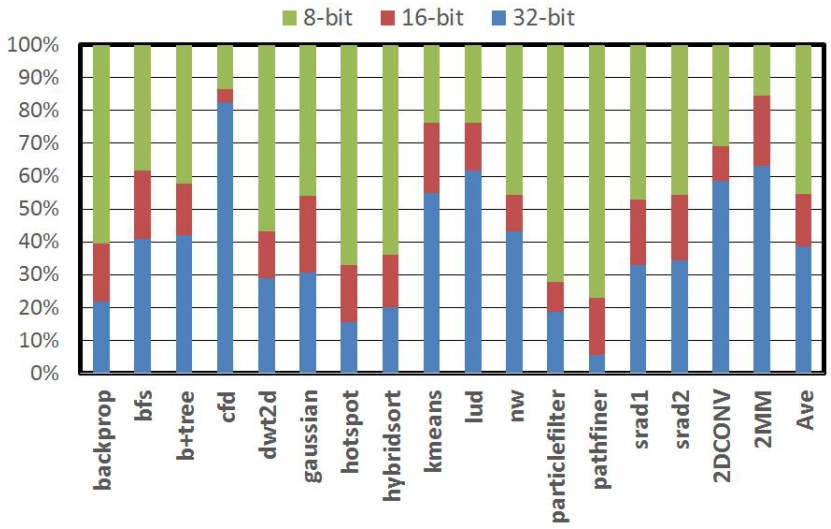


Fig. 3. Width Distribution

3.3 GPU Register Packing

3.3.1 Motivation

We study the operand width across 17 GPU benchmarks. We classify the operand widths into three levels: 8-bit, 16-bit and 32-bit. Figure 3 shows the width distribution of values written into registers. On average, 45.3% of all values consume only 8 bits of a full 32-bit register, 16.1% of all values can be represented by only 16 bits and the remaining 38.6% needs a full-size register with 32 bits. Obviously, using 32-bit registers to store all the operands of different width can be a significant waste of

RF resource. The evaluation results provide us an evidence of opportunities to save plenty of RF space by packing narrow-width operands into a single 32-bit register.

While the narrow-width packing techniques are successful for CPUs, they can not be directly used on GPUs due to the fundamental architectural and register file differences between CPU and GPU. Unlike CPUs which utilize large caches to reduce memory access latency and only limited register files to maintain contexts of tens of threads at most, GPUs are throughput-oriented and need to support tens of thousands of concurrent threads, which have much larger register files to provide efficient context switching between massive active threads. Taking advantage of GPU's RF organization and inspired by work in [35], we develop a narrow-width packing mechanism. [35] discover a significant amount of value similarity in the register file which indicates that values of an operand for all 32 threads within a warp are quite similar. This finding allows a thread-level packing to uniformly pack all 32 threads register values into a relatively small register space. For instance, if the operand stored in a register R0 is less than or equal to 16 bits for each thread within the same warp, the register space required to store all 32 operands then can be reduced from 1024 bits (32*32) to 512 bits (16*32). All 32 thread registers in a warp are mapped on consecutive banks with the same entry index. In the context of the RF design, accessing a single bank entry can read 256 bits data which may contain 8 operand values with full 32-bit size each, 16 operand values with 16-bit size each, or 32 operand values with 8-bit size each.

We evaluate the minimum number of registers used (called register usage in the dissertation) when taking into account the narrow-width operands and find that this number for all warps from thread blocks within the same CUDA kernel barely changes. Figure 4 shows that only 0.13 mispredictions per thousand threads on average across 17 GPU benchmarks. The largest misprediction of register usage ratio is 0.15% for

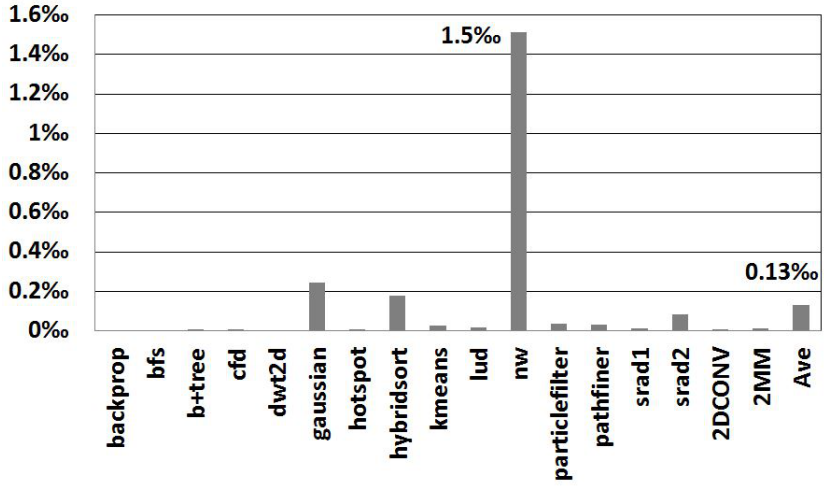


Fig. 4. Register usage Miss Prediction Ratio

benchmark `nw`, indicating only 1.5 mispredictions per thousand threads. Based on this finding, we propose to dynamically profile the register file usage after narrow-width operands packing for in-flight thread blocks to predict future usage which is taken as a guideline for scheduling incoming thread blocks.

3.3.2 Overview

Figure 5 illustrates an overview of the GPU register packing approach. Block 1 (Section 3.3.3) involves dynamically detecting narrow-width operands and profiling operand width boundaries at write back stage. Block 2 (Section 3.3.4) illustrates a table for thread-level packing and register usage prediction. Block 3 (Section 3.3.5) is the renaming table based register reallocation mechanism that maps architectural registers to physical registers and corrects the register value width by assigning the size-increased register with large space, in case of mispredictions.

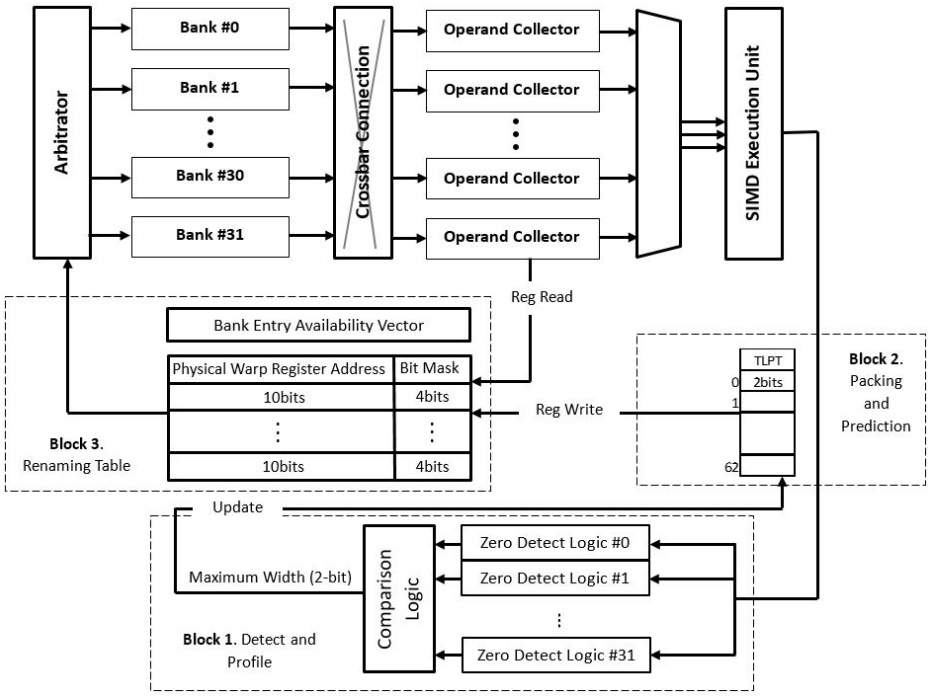


Fig. 5. Overview of the GPU Register Packing Approach

3.3.3 Detecting and Profiling Narrow-Width Operands Dynamically

The narrow-width operands are detected at the write-back stage by using 32 zero detection logic units [27] as shown in Figure 5. The 32 zero detection logic units are required for 32 threads within the same warp. Each unit inspects one operand width from a single thread and outputs a 2-bit operand width indicator. We classify all operand widths into three levels, 8-bit, 16-bit and 32-bit. Therefore only 2 bits are necessary to represent all cases. Specifically, we define that "01" represents 8-bit operand, "10" represents 16-bit operand and "11" represents 32-bit operand. All the outputs from 32 zero detection logic units then are compared to find the upper bound width of 32 values which can guarantee sufficient bits for all operands. The upper bound width is forwarded to thread-level packing table with register index in the form of profiling information.

3.3.4 Register Packing and Register Usage Prediction

As shown in Figure 5, a thread-level packing table (TLPT) is added to each SM to store all narrow-width information provided by the narrow-width detecting stage. When the TLPT receives an operand width from the narrow-width detecting logic, it makes a comparison between the newcomer and the current maintained operand width and it only keeps the width that is wider. The TLPT also provides misprediction information to the renaming table and we will discuss this later in Section 3.3.4. In NVIDIA Fermi architecture, each thread can use up to 63 registers. In this case, the TLPT has 63 entries and each entry is indexed by a register id and holds 2 bits data indicating the bit width of the value stored in the corresponding register. For example, "10" in entry 5 shows that 16 bits are sufficient for all 32 values stored in register #5. The total TLPT size per SM is calculated by Equation 3.1 below.

$$\begin{aligned} TLPT\ Size &= (\#\ Entry\ \times \#\ bit\ per\ entry) \\ &= 63 \times 2\,bits = 126\,bits \end{aligned} \tag{3.1}$$

When a GPU program starts running, the TLPT is empty and it will be filled dynamically during run-time. The TLPT keeps itself updated with the widest width (i.e. upper bound) fed by the narrow-width detecting and profiling stage for all registers. When all threads in a thread block are completed for the first time, the register usage per thread after register packing can be predicted by adding the sizes of all packed registers together. This register usage prediction is sent to the thread block scheduler to loosen the register file constraint for incoming thread blocks scheduling. A misprediction occurs when the TLPT notices that a current computing result can't be written back due to the insufficient size of the target register. A mechanism taking care of misprediction will be introduced in Section 3.3.5.

3.3.5 A Renaming Table to Address Misprediction

The register renaming is conventionally used on CPUs to eliminate the false data dependencies due to the reusing of architectural registers among several sequential instructions. The false data dependencies can be eliminated through renaming the registers for certain instructions. The renaming table in CPUs offers additional instruction level parallelism (ILP) which can be utilized to improve performance. However, the renaming table in [36] is applied in an opposite way. Instead of duplicating reused architectural registers which is consuming more physical registers, [36] achieves to reduce the number of physical registers without losing the architectural register space. Inspired by work in [36], we leverage register renaming technique to build an virtual view that the total number of architectural registers exceeds the capacity of physical registers so that the GPU application performance which is limited by the RF resources can be promoted.

As shown in Figure 5, a renaming table is added to the GPU RF to hold all mapping information. In GPU RF, 32 full size 32-bit registers are grouped into a warp register, each for a thread in the same warp. In the baseline architecture, each SM has a 128KB register file which is divided into 32 banks and each bank has 128 256-bit wide entries. To access a physical warp register, 4 entries from consecutive banks with same index should be visited. In our design, the renaming table operates registers in the warp level to cooperate with GPU RF. Each entry of the renaming table is indexed by global architectural id. The global architectural id can be simply derived with corresponding warp id, local architectural register id and the total number of architectural registers used by a thread. The content of a renaming table entry includes two data fields, 10 bits physical warp register address (for a 128KB register file, there are 1024 warp registers in total and each warp register

is 128 bytes) and 4 bits for a bit-mask, which compose of reallocation details for one architectural warp register. The 4-bit mask in a renaming table entry which denotes 4 entries from consecutive banks with same index is used to identify the location of an architectural warp register within a physical warp register. Originally, an architectural warp register can only be one-to-one mapped to a physical warp registers. However, the renaming table enables to map multiple architectural registers to a single physical register or even to discrete banks of a physical warp register. For example, 2 16-bit architectural warp registers can be assigned to a physical warp register and a 16-bit architectural warp registers can be mapped to two RF bank entries which are not necessary to be contiguous in a physical warp register, whose first half is mapped to the second entry of a physical warp register and second half is stored in the last entry indicated by a 4-bit mask,"0101". Moreover, a bank entry availability vector per SM (The size is 4096 bits, since the total number of physical warp registers per SM is 1024 and each physical warp register contains 4 bank entries according to the baseline RF architecture) is required and each bit is used to indicate if a corresponding bank entry of a physical warp register is assigned to an architectural warp register or remains unused [36].

Using a renaming table with 2048 entries, the GPU RF with 1024 physical warp registers can support up to 2048 architectural warp registers if the sufficient narrow-width operands can be discovered during the run-time. Keeping increasing the number of entries of renaming table can potentially provide even more architectural warp registers, however, its effectiveness highly depends on the amount of narrow-width operands which is varied for different GPU applications and the hardware overhead raises meanwhile which can lead to a great waste in case of a lack in narrow-width operands. All 17 benchmarks we examined in this work earns a 47% average reduction on RF resources from register packing, thereby we propose to configure the renam-

ing table with 2048 entries (twice of the number of total physical registers) to meet the requirement of most applications and avoid unnecessary hardware overhead and resource waste. The total size of the renaming table and the bank entry availability vector is calculated by Equation 3.2 below.

$$\begin{aligned} &Renaming\,Table + Availability\,Vector = \\ &\#\,Entry \times \#\,bit\,per\,entry + \#Bank\,entry \\ &= 2048 \times 14\,bits + 4096\,bits = 32768\,bits \end{aligned} \tag{3.2}$$

As shown in Figure 5, when writing back results to RF, the width of computing results are first examined by the narrow-width detecting logic and then forwarded to the TLPT. The detected width is compared to the width maintained by the TLPT which will keep the width unchanged if the newly detected width is narrower than or the same with the old one. On the other hand, the TLPT updates the width of corresponding register and inform the renaming table with a new width as a misprediction message if the newly detected width is wider. In case of a misprediction, the renaming table attempts to re-map the width-mismatched architectural register to another physical register with sufficient free entries. Whether there is a mispredciton or not, after visiting the renaming table, the physical warp register address and the data field indicator (a 4-bit mask) of the corresponding architectural warp register can be retrieved, which direct the bank arbitrator to write the computing result to certain entries of the target physical register. Reading RF is quite straightforward without considering the misprediction.

3.3.6 Handling Redundant Registers

Due to employing register packing technique, OWAR is able to meet the architectural register usage and at the same time lower the demand on physical register resources. Consequently, a great portion of GPU RF can be saved and is left to be underutilized resulting in energy inefficiency and resource waste. In this work, we explore two candidates for handling redundant registers.

Running beyond the limit: Instead of turning off the redundant hardware resources, we propose to fully utilize them to attempt performance improvement. The performance of many GPU applications is prevented from further improvement by the limitation of hardware resources, especially the number of registers. Since OWAR offers additional RF resources, SM is able to host more co-running threads which can enhance the ability of hiding memory latencies and hence contributes to better performance. The overall energy consumption is then reduced due to the shortened execution time. The power gating method can be added to further reduce the energy consumption, however the benefits from power gating drop off as most sub-arrays that could be shut down, are reused by excess threads.

Power gating: We apply a traditional coarser-grained sub-array power gating [37]. GPU RF is partitioned horizontally into several sub-arrays. Taking advantage of register renaming, the in-service physical registers can be concentrated in certain sub-arrays, making the rest of sub-arrays, if any, unoccupied. To save energy consumption, the spared sub-arrays are then power gated. An in-active sub-array will wake up only when active sub-arrays run out of free registers. Under circumstance of sufficient architectural register supply, the sub-array level power gating scheme shuts down excessive hardware resources to avoid energy waste at little cost of performance loss.

3.3.7 Overheads of OWAR

The total hardware overheads of OWAR are composed primarily of three components, a TLPT, a renaming table and a bank availability indicator. The TLPT size is 16 bytes and the size of the renaming table is $28672bits = 3.5KB$ as calculated in Equation 3.1 and 3.2, respectively. According to the baseline RF organization, the total number of bank entries is 4096. Thus, the size of the bank availability indicator is $4096bits = 0.5KB$ (Each bit indicates the status of a bank entry, 0 for **free**, 1 for **unused**). The hardware cost of OWAR implementation ends up totaling $126 + 28672 + 4096 = 32894bits \approx 4KB$ accounting for 3% of GPU RF size. The energy consumption of the additions are included into GPU's total energy model and are described in Section 3.4. We also take performance overhead caused by sub-array wake-up delay into consideration. Whenever a new sub-array is required to be re-activated from power gated status, the pipeline is stalled until the sub-array is completely ready for service. The sub-array wake-up penalty is conservatively set to one cycle by default, although it is proven to be less than one cycle by calculation using CACTI-P [37].

3.4 Experimental Results of Performance-Orientated OWAR

3.4.1 Performance Improvement

To allow more thread blocks to run on SM concurrently, we loosen the constraints of maximum number of thread blocks and warps. All benchmarks we evaluated are grouped into three levels based on how their performance is improved by register packing: (1) `hotspot`, `b+tree`, `backprop`, `bfs`, `srad1`, `dwt2d`, `srad2` and `gaussian` (Obvious improvement). (2)`lud`, `nw`, `hybridsort`, `pathfiner`, `cfd`, `particlefilter` and `kmeans` (Slight improvement, below 5%). (3) `2MM` and `2DCONV` (Negative effect).

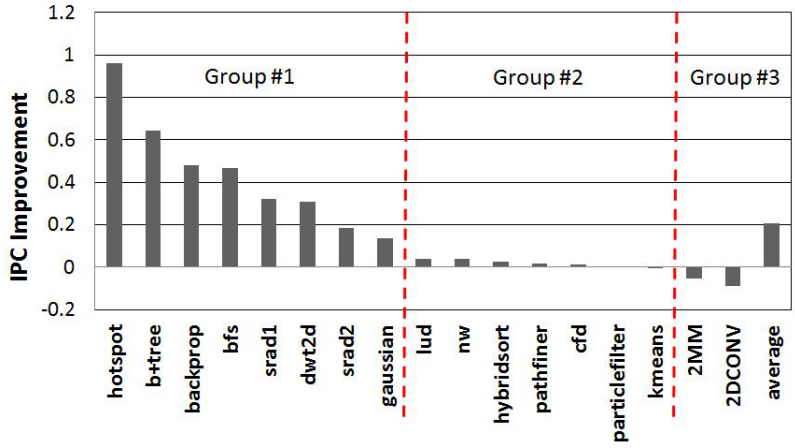


Fig. 6. IPC Improvement

Figure 6 shows the IPC improvement of 17 benchmarks. Performance of benchmarks in Group 1 is enhanced significantly by up to 96.2% and 43.8% on average. The IPC improvement for benchmarks in Group 2 is limited to 1.7% only. For `2MM` and `2DCONV` in Group 3, our scheme degrades the performance by 7%.

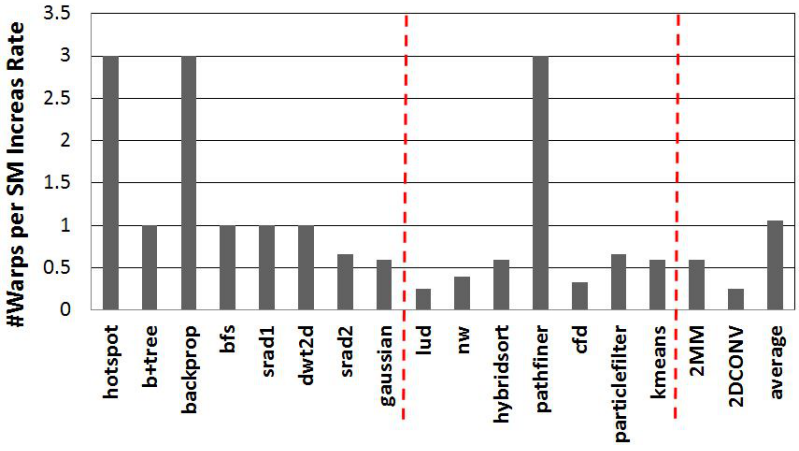


Fig. 7. Active # Warps per SM Increase Rate

By examining Figure 7 that shows the increase rate of the number of active warps on SM and Figure 8 that depicts the difference of the number of warps after employing register packing, we can draw that the IPC improvement of our scheme

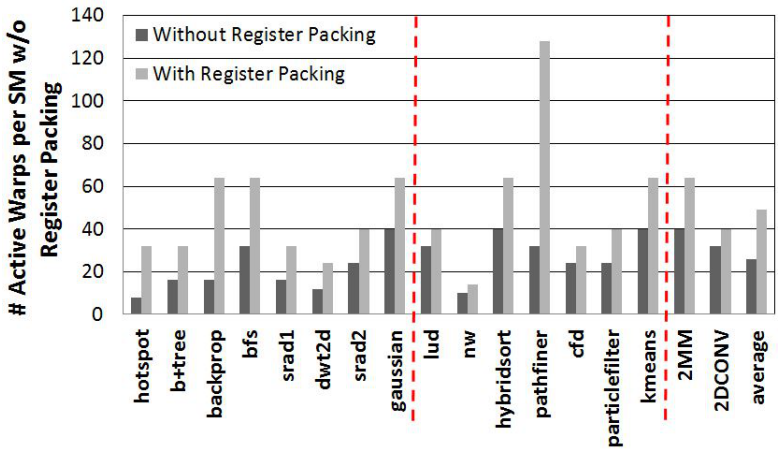


Fig. 8. Active Warps per SM w/o Register Packing

roughly follows two rules: (1) The more additional warps that are scheduled on SM, the higher the chances to achieve more performance improvement, because the additional active warps on SM offer more TLP to hide memory latency. (2) The performance of kernels with a small number of warps on SM is easier to be enhanced by register packing, because there are abundant memory latencies left to be hidden due to the lack of TLP originally. While most performance improvement can be explained by using these two rules, there are several exceptions. For example, `backprop` has equal number of warps with `b+tree` before register packing and the warp increase rate of `backprop` is triple of that of `b+tree` after register packing, however, it has less significant improvement than `b+tree`. Another exception is `pathfiner` in Group 2 which only gains 1.5% performance improvement with 300% warps increase. We will explain these exceptions as well as the reason for performance degradation of two benchmarks in Group 3 by including cache resource contention. The rest of the benchmarks in Group 2 show either low warp increase rate or relatively large number of initial warps, leading to moderate performance improvement. On average, the warp number increase rate is 140.8%, 47.5% and 42.5% for Group 1, Group 2, and Group

3 respectively and the initial warp number on SM for each group is 20.5, 28.9 and 36. The average IPC improvement across 17 benchmarks is 20.5%.

3.4.2 Cache Impact on Register Packing Performance

GPUs rely on the massive threading to hide the latency of memory access. However, running with the maximum thread-level parallelism (TLP) does not necessarily lead to the optimal performance due to the excessive thread contention for cache resource. We classify the negative impact of cache resource contention into two aspects, L1 cache miss increase and L1 reservation fails. In the case of heavy cache resource contention, L1 cache miss increase leads to more accesses to lower level memory hierarchy and thus larger memory access latencies. The L1 reservation fail is another source of L1 access penalty. The limited L1 cache resources such as miss-status holding registers(MSHRs) which are always fully captured by overwhelming access misses can lead to heavy contentions between threads. GPUs can only have limited cache misses in-flight. The status of on-going cache misses is stored in MSHRs. When a memory request is generated due to an L1 cache miss, the status of cache miss is forwarded to MSHR to check if the same request from another warp is already issued and still in progress. If the same request exists in the MSHRs, an entry to the MSHR corresponding to the requested cache line is allocated to make sure that the returned cache line is offered to both warps. If the request does not exist in the MSHRs, a new MSHR entry is dispatched to store the missing cache line status. Since both MSHR entries and entries to the same MSHR are limited, MSHR full occurs when no new MSHR entry can be allocated or no new entry to a specific MSHR can be allocated. An active warp is stalled under both situations aforementioned and it keeps retrying the next cycles until an empty MSHR entry is available. The increase of the frequency of MSHR full can introduce more stall cycles and lead to performance degradation.

In one sense, the frequency of reservation fail is an indicator of cache contention. The increase rate of L1 cache misses is shown in Figure 10 and Figure 11 depicts the increase of L1 reservation fails per kilo instructions.

The benchmark `pathfinder` in Group 2 gains only 1.5% performance improvement despite 300% increase of additional warps, because the global miss rate of `pathfinder` is only 0.85% (as shown in Figure 9) leaving very few memory latencies to be hidden. For benchmark `pathfiner`, although 96 more warps can be scheduled on SM and the negative impact from cache resource contention is minimal (the number of both L1 misses and reservation fails is nearly unchanged), the 32 original warps are already sufficient for latency hiding, thus there is no room to improve performance of `pathfinder` by scheduling additional warps on SM.

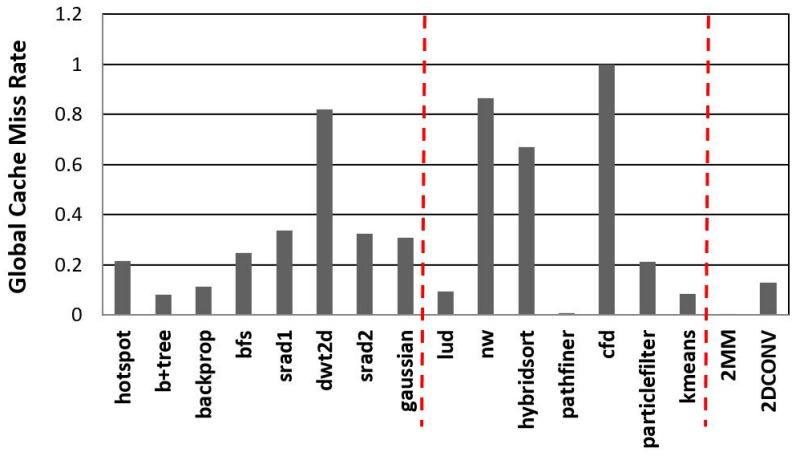


Fig. 9. Global Cache Miss Rate

Our register packing scheme can reduce the number of L1 reservation fails for `gaussian` and `nw` instead of increasing for the rest of benchmarks, as shown in Figure 11. Due to cache locality, when scheduling more warps on SM, more cache misses from different warps requesting for the same data can be assigned to the same MSHR. The number of L1 reservation fails is then reduced as MSHRs are more efficiently

used by merging more cache misses to the same MSHR, contributing to performance improvement for some benchmarks like `gaussian` and `nw`.

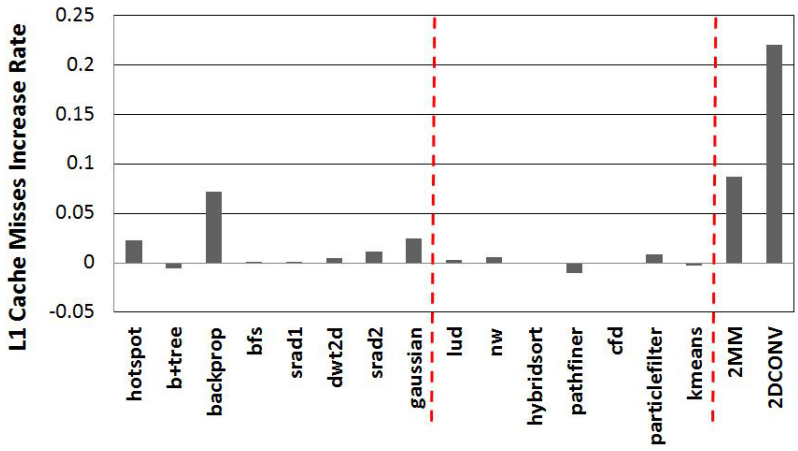


Fig. 10. L1 Cache Misses Increase Rate

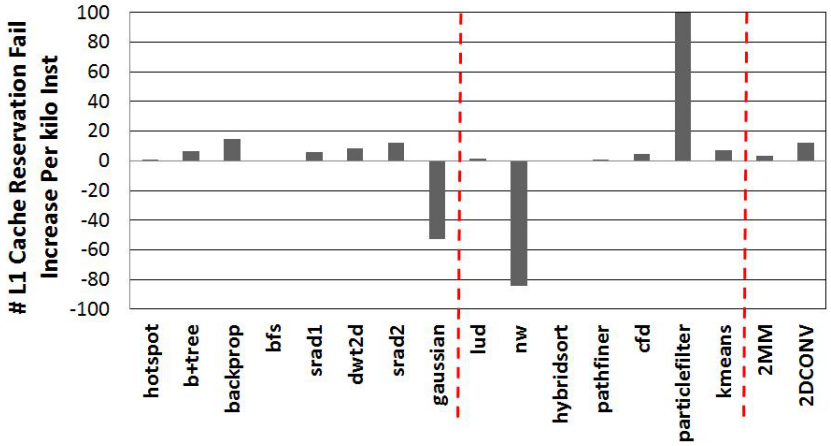


Fig. 11. L1 Cache Reservation Fail Increase Per kilo Instructions

In most cases, the increased cache contention including L1 misses and reservation fails hurt the performance of our register packing approach. For `backprop` in Group 2, the L1 cache misses increase rate is 7.2% compared to -0.57% for `b+tree` which explains why the performance of `backprop` is improved less than that of `b+tree` even with a larger increase of the number of warps. Register packing helps `particlefilter`

Table 1. Energy consumption of renaming table compared to RF

	RF	Renaming table
Leakage power	9.75 mW	0.08mW
Dynamic energy per read	53.6 pJ	0.61 pJ
Dynamic energy per write	57.0 pJ	0.63 pJ

execute more warps concurrently (from 24 warps to 40 warps); however, the sharply increased L1 reservation fails as shown in Figure 11 diminish the benefit from register packing, leading to minimal performance improvement. Moreover, the two benchmarks `2MM` and `2DCONV` in Group 3 which are under the pressure of both L1 cache misses and reservation fails suffer from our scheme. Their performance is reduced by 5.2% and 8.8% respectively.

3.4.3 Power Consumption

The power parameters of renaming table and RF calculated by CACTI [38] are listed in Table 1. The leakage energy of the renaming table only accounts for 0.79% of RF leakage energy. The dynamical energy spending on the write or read of the renaming table slightly increases the total RF dynamic energy (1.13% increase for read, 1.10% increase for write). Thus it can be seen that the energy overhead introduced by renaming table is negligible in comparison to the energy consumption of RF. Thus, power consumption will be naturally reduced if GPU applications can finish executing earlier. Figure 12 shows power consumptions reduction of our register packing scheme. The power consumption is reduce by 6% on average for all benchmarks in Group 1. `particlefilter` is the only benchmark that consumes more power in Group 2 as a consequence of heavy cache contention. The rest benchmarks in Group 2 achieve 1.5% power consumption reduction on average. Due to the performance

degradation, the power consumption is increased by 3.1% and 4.2% respectively for `2MM` and `2DCONV` in Group 3. For all 17 benchmarks, the power consumption is slightly reduced by 2.5%.

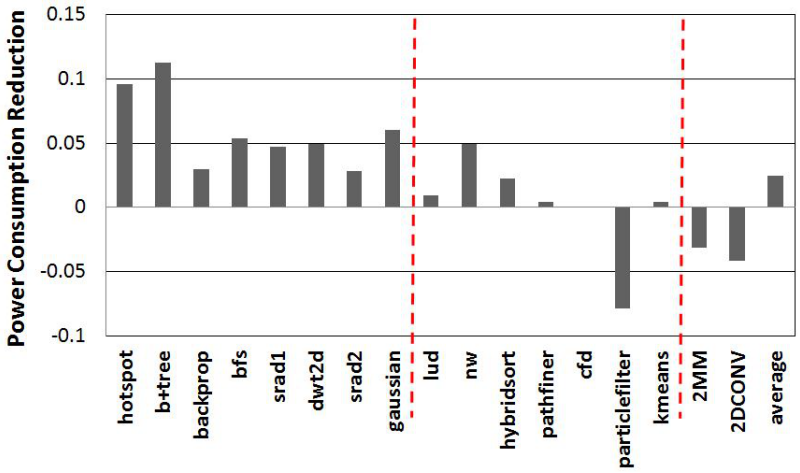


Fig. 12. Power Consumption Reduction

3.5 Experimental Results of Energy-Efficiency-Orientated OWAR

We evaluated two OWAR based GPU RF management approaches, OWAR-PG (OWAR w/ power gating) and OWAR-TO-PG (OWAR w/ thread overrun and power gating). OWAR-PG focuses on reducing RF energy consumption by shutting down spared RF resources. OWAR-TO-PG attempts to improve the energy efficiency through making the most use of the additional RF resources freed by OWAR. And the power gating is reserved by OWAR-TO-PG to further reduce the energy consumption. We compared the energy improvement of these two schemes with the baseline (w/o OWAR). We also measured the performance improvement of OWAR-TO-PG.

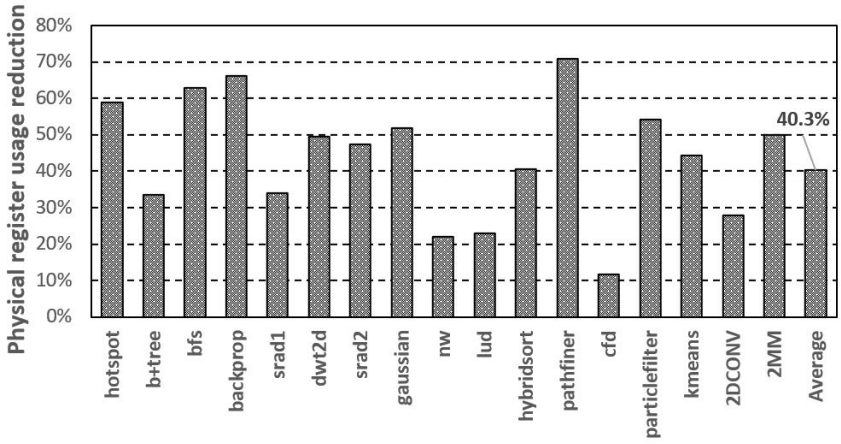


Fig. 13. Reduction in physical register usage

3.5.1 RF Utilization

As aforementioned in Section 3.3, OWAR is able to split a physical register up to four parts and assign them to multiple architectural registers containing narrow-width results. Although the architectural register usage per warp stays the same, the corresponding physical register usage is significant reduced by OWAR. As shown in Figure 13, OWAR can achieve up to 70.8% off the original register usage per warp and the average reduction for all evaluated benchmarks is 40.3%. The register usage reduction results in high under-utilization of RF resources. During the run-time, we sample the number of in-service registers periodically and take the average of all samples as an estimation of the RF utilization of an application. Figure 20 shows the GPU RF utilization rates for 17 benchmarks by using RF management approach proposed in this work. With no optimization, 77.9% of total RF is used on average. As expected, the RF utilization rate drops sharply to 45.7% when OWAR-PG is employed, indicating that a great number of registers which are supposed to be used, are now saved by OWAR. The large idle portion of RF can be power gated by OWAR-PG for the purpose of saving RF's dynamic and leakage energy.

Unlike OWAR-PG, OWAR-TO-PG dedicates to improve the RF utilization and the performance by increasing the capacity of concurrent threads. Provided that the performance is boosted, the improvement of energy efficiency is then applied not just to RF, but to all the components included in GPU. Consequently, OWAR-TO-PG contributes to a higher RF utilization (81.1% on average) compared to baseline. Even though RF is further exploited by OWAR-TO-PG, for some benchmarks like `bfs`, `gaussian` and `nw`, there are still considerable unused registers remaining, which thereby can be power gated to enable more RF energy savings.

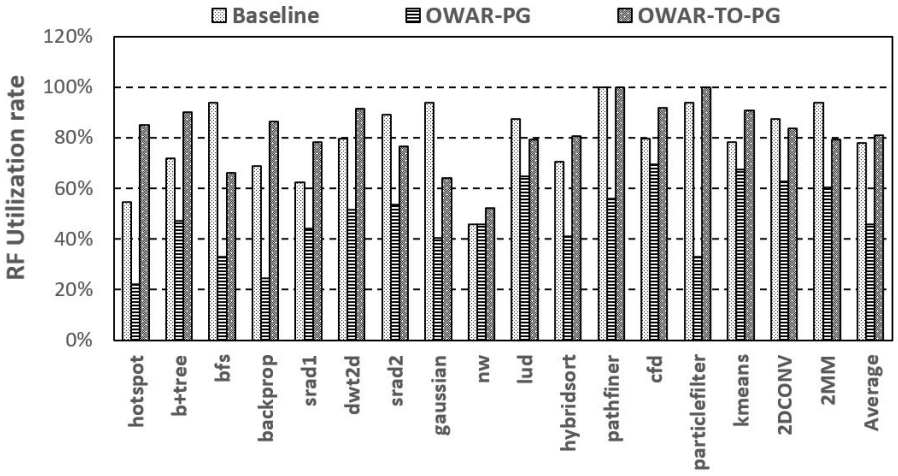


Fig. 14. GPU RF utilization of Baseline, OWAR-PG, OWAR-TO-PG

3.5.2 Reducing the Number of RF Bank accesses

Based on our baseline RF architecture, accessing to a single bank entry can only fetch eight 32-bit register values and reading an operand for a warp instruction needs to access up to four banks in the absence of RF optimization. Through the help of OWAR, fewer bank accesses are necessary to collect all values from RF, for example one access for 8-bit width data, two accesses for 16-bit width data, if the desired data can be represented with narrowed width and is stored in packed registers. We keep

the trace of accesses to narrow-width values and record the total number of RF bank accesses for each benchmark. As shown in Figure 15, both OWAR-PG and OWAR-TO-PG have the power to complete register operations with minimal number of bank accesses and thus reduce overall bank accessing times. The benchmark `pathfiner` represents the most significant reduction (28.6%, OWAR-PG and OWAR-TO-PG achieve the same result). On average, the total number of RF bank accesses is reduced by 13.1% for OWAR-PG and OWAR-TO-PG. In addition, it's reasonable to assume that the reduction should benefit energy efficiency by lowering RF dynamic energy consumption.

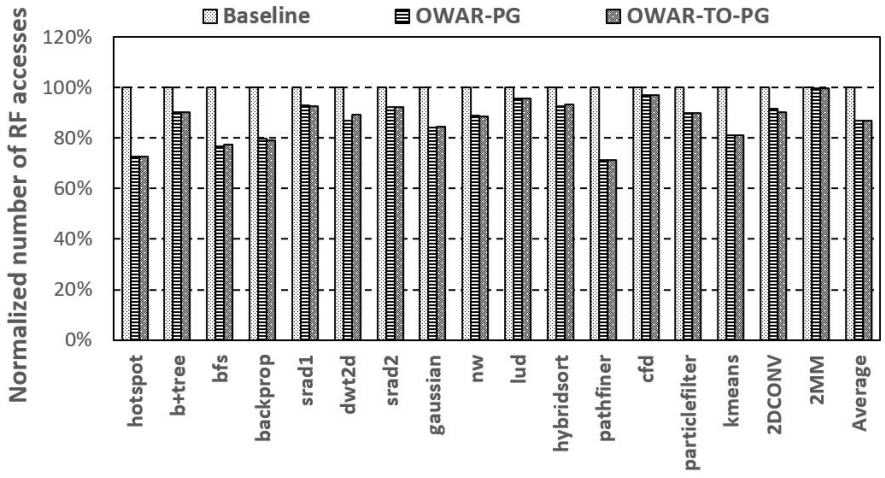


Fig. 15. Normalized number of RF bank accesses

3.5.3 Performance Overhead and Energy Saving

Performance overhead: We evaluate the performance overhead of sub-array re-activation. We assume that it takes one cycle to completely reactivate a sub-array from power gated status. As shown in Figure 16, the performance of only a few (`b+tree`, `backprop`, `lud` and `hybirdsort`) among 17 benchmarks is negatively affected by sub-array re-activation latency. Most of the benchmarks maintain their

performance. The reason is that the sub-array re-activation barely happens during application execution, resulting in negligible performance overhead. Unlike other benchmarks, The execution cycle of benchmark `2DCONV` is reduced instead because the overhead cycle unintentionally alleviates the high pressure of cache resource contention. The performance overhead caused by sub-array re-activation is only 0.2% on average as shown in Figure 16 for all benchmarks except `2DCONV`.

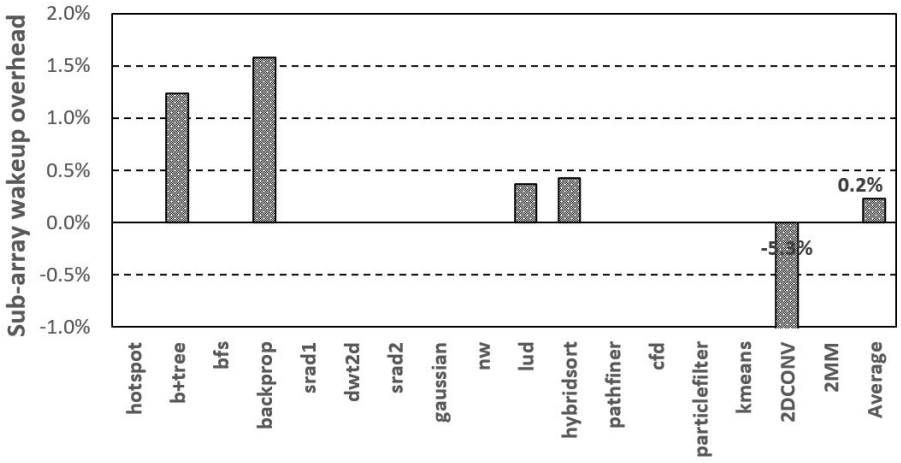
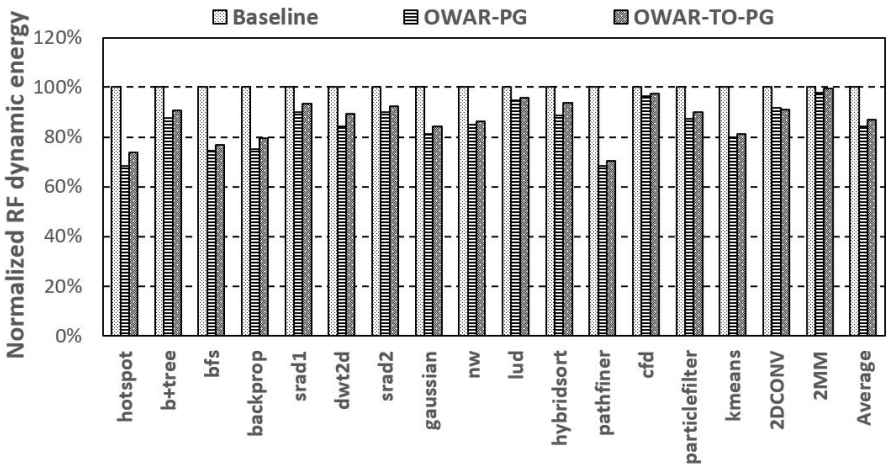
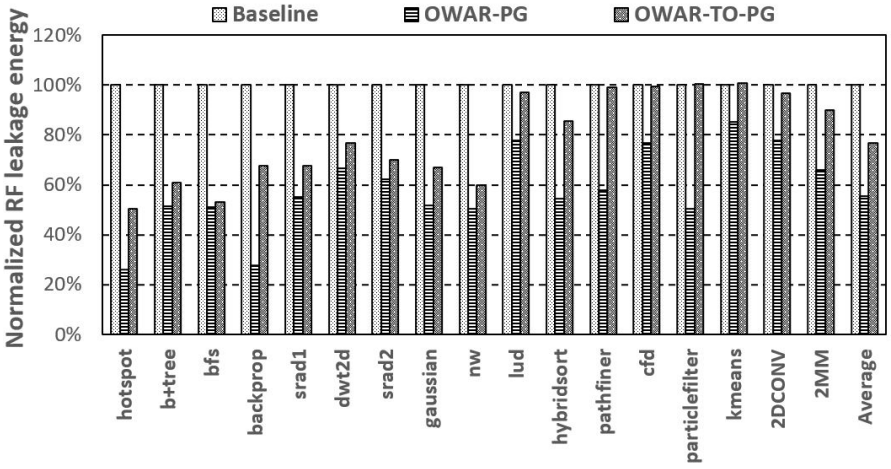


Fig. 16. Sub-array re-active overhead

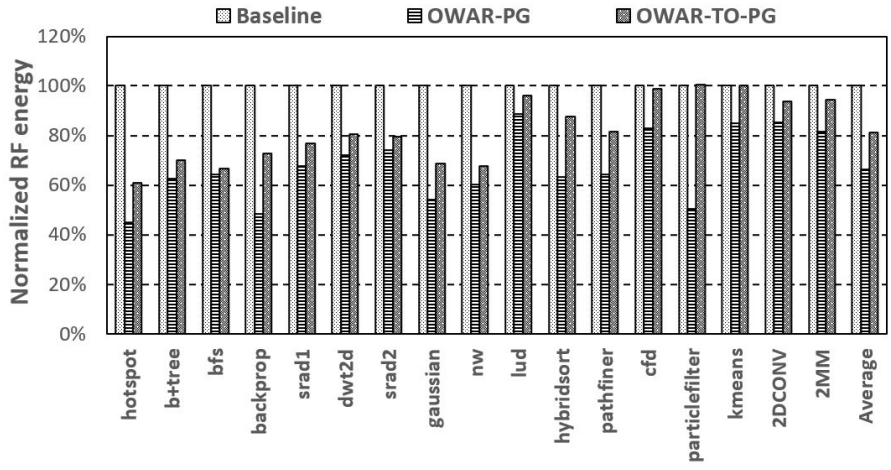
Energy reduction: First, we focus on evaluating the efficacy of OWAR-PG and OWAR-TO-PG on RF energy reduction. Figure 25 represents the dynamic, leakage and total RF energy of two OWAR schemes, respectively. RF dynamic savings are generated by RF accessing frequency reduction and sub-array level power gating. OWAR-PG and OWAR-TO-PG produce similar reduction in the number of RF accesses, while OWAR-PG is better at power gating due to sufficient free registers. Thus, OWAR-PG can save more RF dynamic energy than OWAR-TO-PG (OWAR-PG: 15.7%, OWAR-TO-PG: 13.0%). Similarly, OWAR-PG achieves more RF leakage energy reduction than OWAR-TO-PG (OWAR-PG: 44.4%, OWAR-TO-PG: 23.1%). Overall, for the purpose of RF energy saving only, OWAR-PG is better than OWAR-



(a) RF dynamic energy

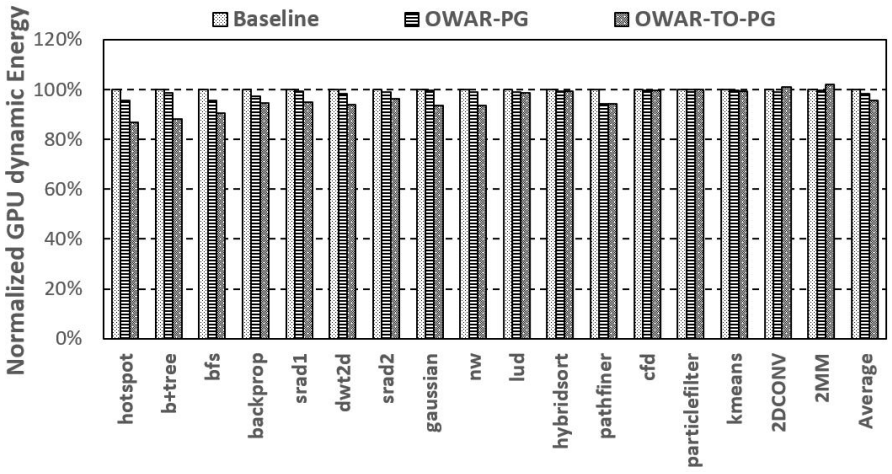


(b) RF leakage energy

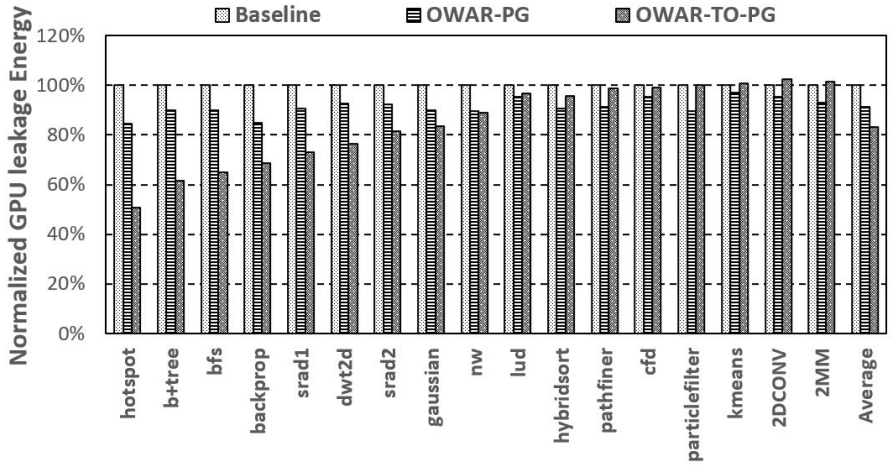


(c) Total RF energy

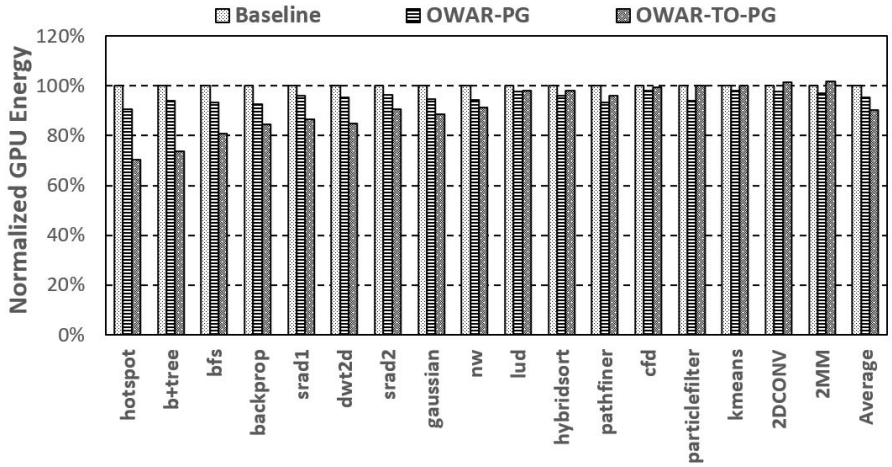
Fig. 17. RF energy reduction.



(a) GPU dynamic energy



(b) GPU leakage energy



(c) Total GPU energy

Fig. 18. GPU energy reduction.

TO-PG (OWAR-PG: 33.7%, OWAR-TO-PG: 18.9%), for the reason that OWAR-PG can save more RF energy through power gating.

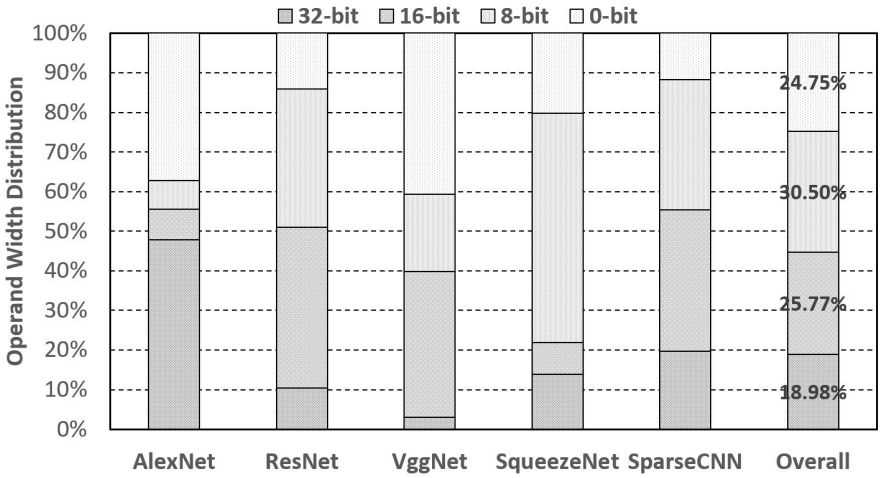
Second, we examine the power of OWAR-PG and OWAR-TO-PG in reducing GPU's total energy. Figure 18 illustrates that OWAR-TO-PG is the better option in case that performance is significantly improved. In contrast, OWAR-TO-PG is less powerful than OWAR-PG if it results in only limited performance benefit. Generally speaking, OWAR-TO-PG outperforms OWAR-PG in overall energy reduction (OWAR-TO-PG: 9.5%, OWAR-PG: 4.8%), because the former can not only reduce the energy consumption of RF but also that of other components in GPU owing to the performance improvement, while the latter only reduces the energy consumption of RF.

3.6 An Application Scenario of OWAR: Build Energy-Efficient GPU Computing Environment for Machine Learning Algorithms

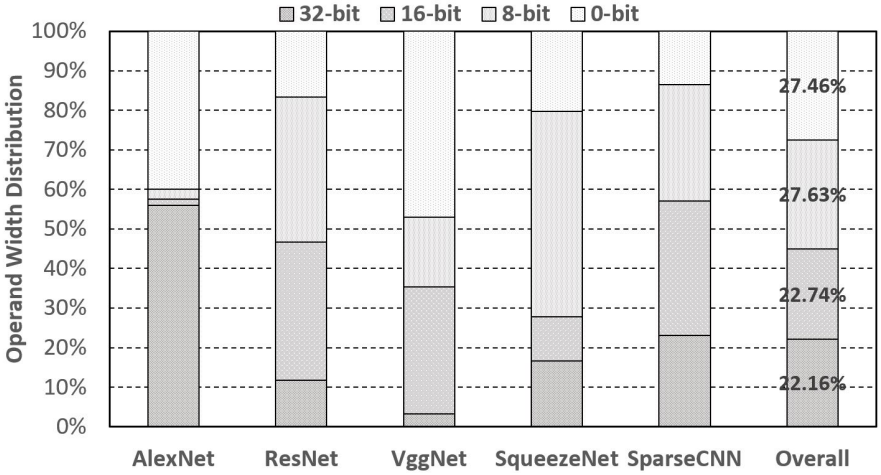
Machine learning algorithms can gain a dramatic speedup from GPU's massive threading ability. More and more domains such as social media, automotive vehicles, medical, consumer electronics, have applied the machine learning algorithms to analyze the big-database and reveal hidden knowledge from abstract information. The databases that machine learning algorithms typically work on are increasing exponentially due to the highly developed internet, information and sensing technologies. Consequently, machine learning algorithms which are high computational complexity and data intensive are usually running for a long period on CPUs, since CPUs are designed for more general-purpose applications. Instead, machine learning algorithms are gradually deployed on GPGPU computing platforms for the performance acceleration. Owing to GPGPU's nature of hosting massive threads and its redundant floating-point operation units and high memory bandwidth, it is a great fit to ac-

commodate machine learning algorithms which are constructed with lots of matrix multiplications. However, the energy consumption on GPGPU has become a concern for machine learning algorithms. We observed that RF occupancies of modern machine learning algorithms are relatively low leaving a great waste of GPU's RF leakage energy. Furthermore, we found that the data maintained by the RF contains a large fraction of narrow-width operands for machine learning algorithms. Therefore, we propose to use OWAR to improve the energy efficiency of GPGPUs for machine learning algorithms.

We evaluated the energy-efficiency improvement of the register packing scheme with 5 popular machine learning algorithms, AlexNet [39], Vgg-16 [40], ResNet [41], SqueezeNet [42] and SparseCNN [43]. AlexNet, a well-known convolutional neural network (CNN), has gained prominence for its effectiveness in image recognition tasks. The Vgg architecture serves as the foundation for revolutionary models in object recognition. As a deep neural network, VggNet not only exceeds baseline performance on numerous tasks and datasets beyond ImageNet but also remains one of the most widely adopted image recognition architectures to this day. The original ResNet architecture, known as ResNet-34, consisted of 34 weighted layers. It introduced a groundbreaking approach to incorporating additional convolutional layers into a CNN while mitigating the vanishing gradient problem. This was achieved through the utilization of shortcut connections, where certain layers are bypassed, transforming a conventional network into a residual network. SqueezeNet is a convolutional neural network that incorporates specific design strategies to effectively minimize the number of parameters. One of its notable techniques involves the implementation of fire modules, which utilize 1x1 convolutions to "squeeze" the parameters. By employing this approach, SqueezeNet achieves parameter reduction while maintaining the network's performance and capabilities. SparseCNN is demonstrated to decrease pa-



(a) Operands Width for RF Write Accesses



(b) Operands Width for RF Read Accesses

Fig. 19. Operands Width Distribution in Machine Learning Algorithms.

rameter redundancy and computational complexity with negligible recognition loss by utilizing a sparse decomposition technique. For all these machine learning models, the convolution layers demand the highest computational resources owing to the intensive floating-point matrix multiplication. Therefore, we applied the register packing mechanism to the kernel of convolution layers for the evaluation.

3.6.1 Characteristic Analysis of Machine Learning Algorithms on RF Utilization and Energy Comsumption

Narrow-Width Operands: Figure 19a shows the width of operands for RF write accesses. Overall, Only 18.98% of the operands use the full size of a 32-bit register. For RF read accesses, the percentage is 22.16% as shown in Figure 19b. The rest of operands can be all represented with less bits. The significant content of narrow-width operands in machine learning algorithms implies that the narrow-width operands packing technique is a great fit for saving RF resources and also the energy consumption.

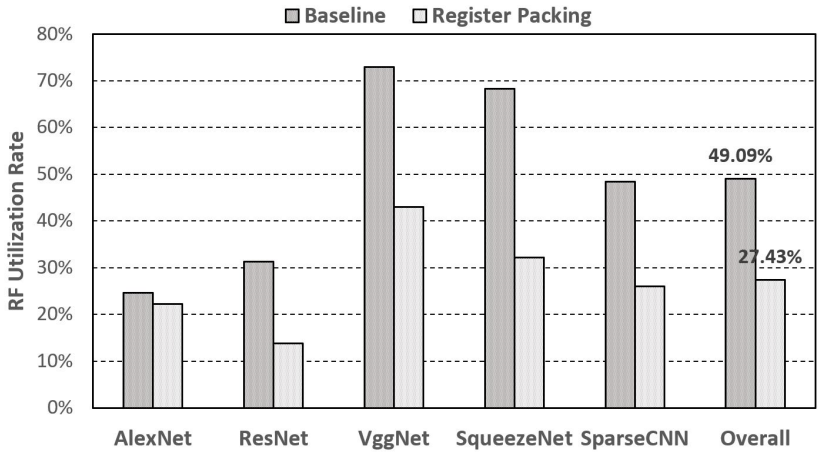


Fig. 20. GPU RF utilization of **Baseline** and **Register Packing** in Machine Learning Algorithms.

RF Utilization Figure 20 depicts the GPU RF utilization rates achieved through the implementation of register packing RF management. Without any optimization, only half of the RF (49.09%) is utilized approximately on average. As expected, with register packing employed, the RF utilization rate further drops to 27.43%, indicating that registers that would have otherwise been used are now saved through register packing. The idle portion of the RF can be power gated to conserve dynamic and

Table 2. Factors to Limit the Number of Concurrent Threads

Algorithms	Limitation
AlexNet	Shared Memory Per SM
ResNet	Maximum Number of CTAs Per SM
Vgg-16	Maximum Number of Register Per SM
SqueezeNet	Maximum Number of CTAs Per SM
SparseCNN	Maximum Number of Threads Per SM

leakage energy.

Table 2 depicts the factors that prevent the concurrency. Only for Vgg-16, the number of concurrent threads on a SM is limited by the register resource and the RF utilization patio (72.92%) is the highest among all five machine learning algorithms accordingly. For other applications, a SM can not execute additional threads due to the restriction of the shared memory or maximum number of CTAs/threads per SM. Therefore, the RF resource is not fully utilized leaving a great waste of leakage energy.

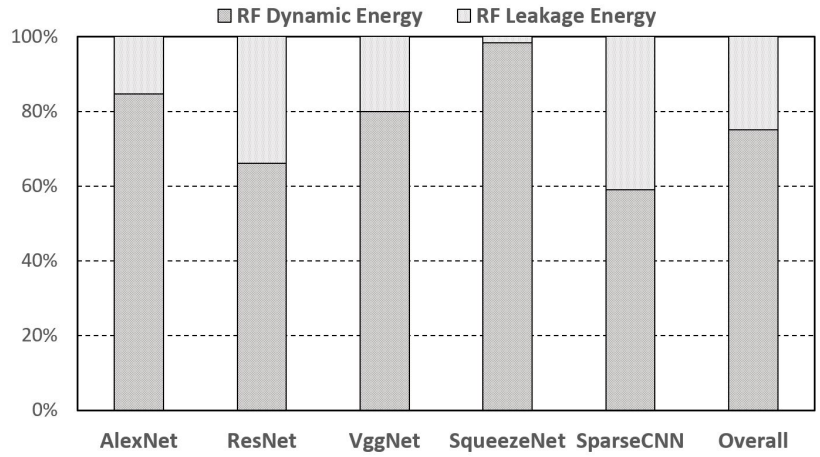


Fig. 21. Total RF Energy Breakdown

Baseline RF Energy Consumption: Figure 21 shows the GPU RF's dynamic and leakage energy for five applications. The leakage energy dissipation contributes most of the total RF energy consumption, which is 75.14% on average.The narrow-width

operand packing is able to reduce both dynamic and leakage energy. Obviously, we can expect that the reduction of RF leakage energy is much more considerable than RF dynamic energy saving, since the leakage energy is the majority of the total RF energy and the leakage energy wasting on the vacant RF banks can be thoroughly eliminated by the power-gating method. The percentage of RF energy in GPU's total energy is given in Figure 22. On average, the RF is responsive to 18.29% of the total GPU energy. The fact indicates that it is meaningful to pursue an energy-efficient RF management as it plays a prominent part in improving the energy-efficiency of the entire GPU.

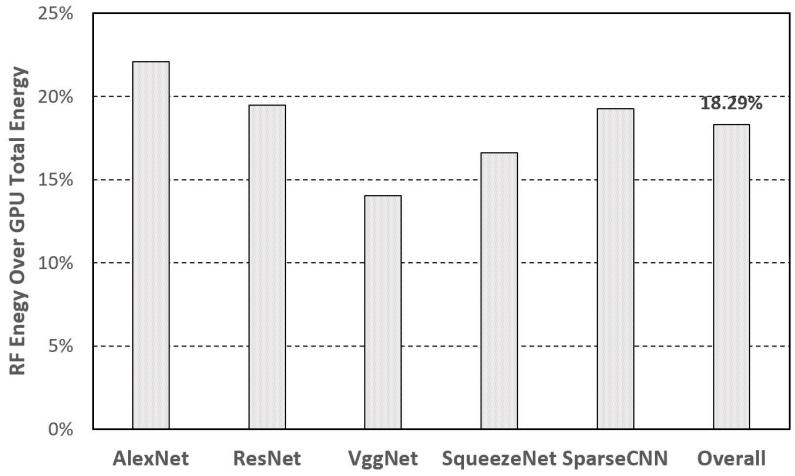


Fig. 22. Percentage of RF Energy over GPU's Total Energy

3.6.2 Performance Overhead

To assess the performance impact of sub-array re-activation during the application of register packing, we consider a one-cycle duration for complete reactivation from a power-gated state. As illustrated in Figure 23, the performance overheads resulting from sub-array are minimal. The SqueezeNet contributes the most significant performance degradation among all five applications and the overhead on performance

is 3.17% which is still negligible. For the other four applications, the performance overhead are all under 1%. On average, the register packing scheme only introduces 0.73% more execution cycles. Figure 24 shows that the miss prediction rate is very low (0.69% on average), which illustrates that the predicted RF usage is pretty accurate and the power-gated RF banks are barely reactivated. The low miss prediction rate also explains the insignificant performance overhead in Figure 23.

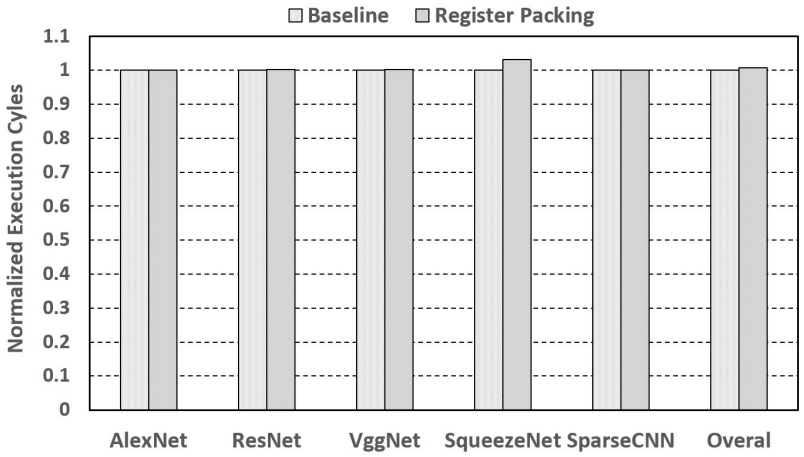


Fig. 23. Normalized performance of Baseline and Register Packing

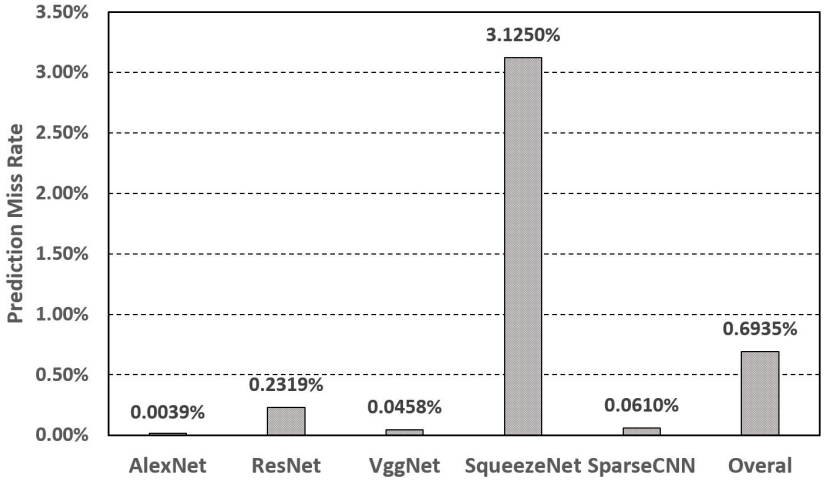
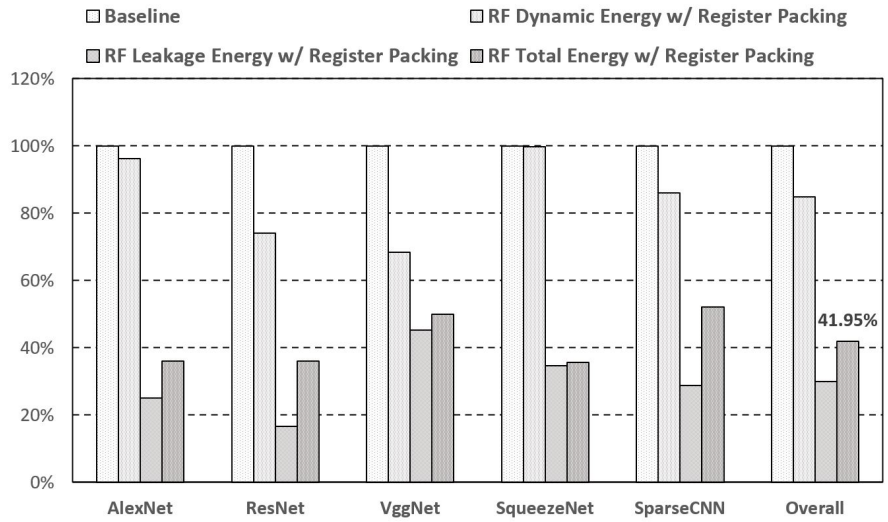


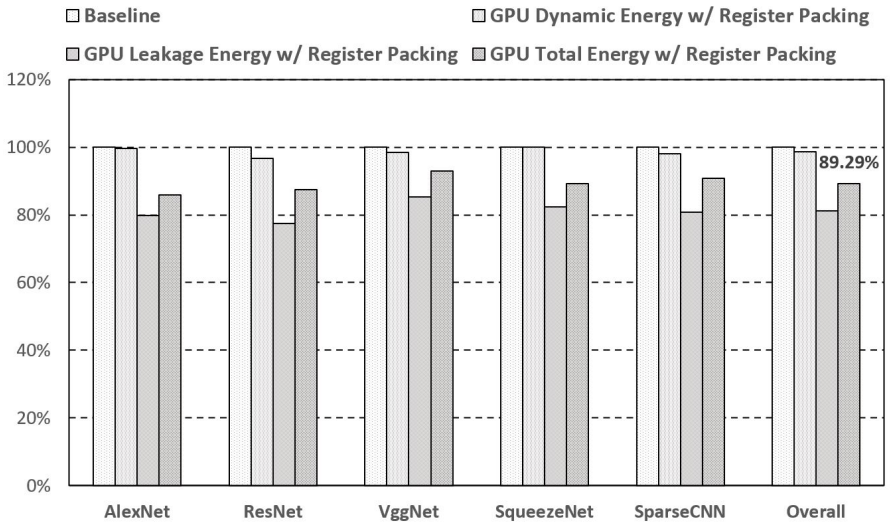
Fig. 24. Miss Prediction Rate of Register Packing

3.6.3 Energy Reduction

In Section 3.6.1, we highlighted that the GPU RF consumes a significant amount of energy, accounting for 18.29% of the total GPU energy consumption (as depicted in Figure 22). Also, we demonstrate that leakage energy constitutes a substantial portion of the total energy consumed by the RF (75.14% for leakage energy and 24.86% for dynamic energy) as illustrated in Figure 21. Consequently, reducing leakage energy is crucial for enhancing RF energy efficiency. Our evaluation focuses on assessing the effectiveness of the RF narrow-width operand packing scheme in reducing RF energy consumption. Figure 25a presents the dynamic, leakage, and total RF energy consumption for the five machine learning applications. RF dynamic energy savings are achieved through reduced RF access frequency and RF leakage energy reduction is accomplished by power gating at the sub-array level, which are implemented via register packing management. The register packing approach demonstrates a RF dynamic energy reduction of 15.1% and a RF leakage energy reduction of 69.97% on average. The total RF energy is reduced by up to 64.34% and 58.05% on average. Furthermore, we examine the effectiveness of the register packing approach in reducing the total energy dissipation of the GPU. Figure 25b illustrates that GPU RF narrow-width operand packing technique yields the considerable reduction in overall energy dissipation, with potential energy savings of up to 14.14% and an average of 10.71%.



(a) RF Energy Reduction



(b) GPU Energy Reduction

Fig. 25. Energy reduction of Register Packing.

CHAPTER 4

GPU REGISTER FILE DROWSY MANAGEMENT

4.1 Introduction

Owing to the large size of the GPU RF which is composed of thousands of high leakage transistors, a considerable fraction of GPU's total energy is consumed by RF [44, 45]. We compared RF energy consumption to the total energy of GPUs by evaluating a batch of representative GPU benchmarks. 15.7% of the GPU's total energy belongs to RF for the benchmarks evaluated in this work. The fact shows us the opportunity to improve the overall energy efficiency of GPUs by reducing the energy consumed by RF. Moreover, the GPU RF utilization rate is only 79.5%, indicating that a large number of registers are actually idle in run-time, leading to a waste of leakage energy. The low utilization of GPU RF is also reported in [46, 47, 48]. We propose to leverage the drowsy technique and power-gate the unused registers to avoid this energy inefficiency.

The drowsy technique was first proposed to reduce leakage energy of CPU caches [49]. CPUs typically utilize large caches to overcome the long-term latency from memory accesses and register files in CPUs only maintain execution states for one or a small number of threads. Therefore, CPUs have much larger caches than register files. By comparison, GPUs can tolerate memory latency by rapidly switching among many threads. The GPU pipelines are filled with massive executable threads to eliminate stalls. Thus, the register file in the GPU is normally kept large enough to store execution contexts of massive threads.

Inspired by the research of drowsy techniques in conventional CPUs, we propose

to apply the drowsy technique to reduce the RF leakage energy consumption. A drowsy policy for GPU register file which is similar to Drowsy-IS has been studied in [7]. This method will simply awake a register from the drowsy mode when it is accessed. The register will go back to the drowsy mode when the requested data has been provided or the writeback stage has been fulfilled. For the applications with a long register file re-access interval, the leakage energy can be significantly reduced with negligible performance degradation. In the case of a great number of short re-access intervals, however, the performance overhead may grow substantially due to the additional cycles generated by re-activating drowsy registers.

To mitigate the performance and potential energy overheads, we propose Drowsy-TA, letting the register wait for a fix number of cycles (i.e., the awake interval) before putting registers into the drowsy mode. Accessing registers during this awake interval is free of re-activation latency, therefore performance overhead can be alleviated. The main role of Drowsy-TA is to trade a small amount of the RF energy reduction for overall performance improvement. The worst case in Drowsy-TA happens when a register is only accessed beyond the duration of the awake interval and hurts the energy efficiency without any performance gain.

To strike the balance between performance and energy reduction for the drowsy RF, we also propose the Drowsy-RI, aiming at predicting the re-access intervals for each register and apply different drowsy strategies to registers in terms of the length of predicted awake interval. In case of a short re-access interval, a register is supposed to stay awake for a short period to receive the next access. On the other hand, since it is inefficient to wait for an access which will not come during the awake interval, a register is put back into the drowsy mode immediately if the re-access distance is longer than the awake interval.

4.2 Related Work

The drowsy technique was first proposed for CPUs to reduce the leakage energy of caches [49]. Large caches of CPUs utilized to cover long memory access latencies consume a large fraction of the total CPU energy, making it important to use cache leakage management techniques such as drowsy caches. In GPUs, however, caches are typically small while RF is much larger, thus we propose to exploit drowsy RF to solve severe energy problem in GPUs. There is a prior study on drowsy RF in GPUs [7]. This work employed a simple drowsy policy, which is putting registers into the drowsy mode immediately after a register read or write to save RF leakage energy. However, we find that GPGPU benchmarks have quite different RF access intervals, and a simply policy as used in [7] may result in moderate or even significant performance overheads, which can also affect the overall GPU energy consumption. Thus, we study and evaluate other drowsy policies to try to reduce the performance degradation while maximizing the RF and/or GPU energy savings. Another work exploits drowsy technique to save RF energy proposed in [50]. They divide each bank in the RF into several pieces, for instance 4 partitions. The first partition in each bank is assumed to be frequently accessed and only features drowsy mode to ensure quick response, while the rest partitions are power gated at the beginning of the execution and will only be activated when they are informed by wake up instructions inserted during compiler time. Their schemes achieve significant RF leakage energy saving.

The drowsy technique are also studied to reduce the leakage energy of GPU caches [51, 52, 53]. [51] proposed the periodic drowsy method which periodically puts entire cache into the drowsy mode to achieve the leakage energy saving of the GPU hybrid SPM-cache. The technique proposed in [52, 53] reduces static power of L1 data cache by placing cache blocks into the the drowsy mode immediately after each

access.

4.3 Leakage Energy Reduction Using Drowsy RF

4.3.1 Drowsy-IS: An Aggressive Drowsy Policy

Power gating unallocated registers: As aforementioned in Section 4.1, a segment of RF may stay idle and never be accessed during run-time for most applications, offering an opportunity to saving leakage energy. By analyzing register allocation information retrieved from the compile time, the unallocated registers can then be located at the beginning of execution and it is safe to power gate these registers for energy saving.

Immediately putting register in the drowsy mode: Drowsy-IS implements a similar policy with [7] to place registers in the drowsy state immediately after a register access for maximum leakage energy saving. The timing of waking up a register is when it is accessed (write or read). After the writeback stage is completed or the operands are provided, the target register returns to the drowsy mode instantly. Drowsy-IS is supported by the tri-modal switch proposed in [54], which allows a register to keep in one of the three states: operational, power gated, and drowsy. The power gated registers consume only negligible leakage energy. The downside of power gating, however, is that a register will lose all maintained data. As compared to power gated mode, more energy are needed to keep registers in the drowsy state, although the leakage energy consumption is still very low (about 10% to 20% of operational state [49, 7]) and the data are completely reserved. By using the tri-modal switch, the unallocated registers are power gated at the beginning of the execution and during run-time, registers are leaving or returning to the drowsy state according to the signal generated by the RF arbitrator. After receiving requests from the operand collector

units, the RF arbitrator decides registers to be written or read and informs the tri-modal switches to wake up corresponding registers. In addition, each access to register in the drowsy mode experiences a re-activation delay of certain cycles (one or two cycles [49, 7]).

4.3.2 Drowsy-TA: Temporal Awake for a Fixed Interval

By observing that a large amount of re-accesses occur within a short time period, Drowsy-TA forces the registers to stay in the operational state for a preset number of cycles in order to avoid repeated re-activation penalty. Obviously, using a long-bit counter for each warp register can result in a significant hardware overhead. This problem can be resolved by using a hierarchical counter mechanism [55] where a single global cycle counter is set up to provide the ticks for much smaller warp register counters. The global counter is usually hardware free, because most processors are already equipped with fine-grained counters for common usage. Moreover, the hardware overhead of additional warp register counters is minimal, because they tick at a much coarser level than the global counter and only need one or at most a few bits. In our scheme, a 1-bit counter for a warp register is sufficient and in total 1024 1-bit counters are added to the 128 KB RF with 1024 warp registers. Each time a register is accessed, the corresponding counter is reset to zero. The counter is incremented to one at the preset number of cycles. At the same time, the associated register is placed into the drowsy mode. In the Drowsy-TA scheme, the re-activation penalty is successfully avoid if a register accessed again before it returns to drowsy mode. In case of a re-access interval beyond the preset number of cycles, the re-activation penalty paid. This, however, is expected to be rare because for most benchmarks, the re-access intervals are mainly below 512 or even 128 cycles.

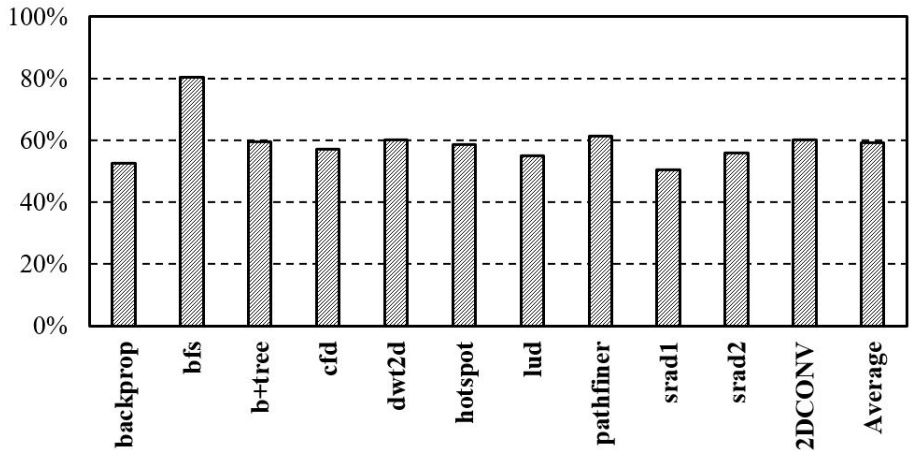


Fig. 26. The prediction accuracy of Drowsy-RI

4.3.3 Drowsy-RI: Balancing between Energy and Performance

Drowsy-RI manages registers with different drowsy strategies according to their respective length of re-access intervals. If a register is re-accessed very soon, Drowsy-RI keeps it awake to receive next accessing by avoiding the activation latency. The leakage energy spending on waiting is also small due to the short re-access interval. On the other hand, if a register will not be accessed for a long time, Drowsy-RI will put it into the drowsy mode instantaneously. Drowsy-RI manages to predict the re-access interval by adding a 1-bit counter and an indicating bit to each warp register. The indicating bit represents the prediction for the re-access interval. "1" indicates that the next accessing comes within a preset number of cycles. "0" stands for that the next accessing arrives after the preset period expires. The 1-bit counter is incremented in case of a miss-prediction and decremented if the prediction is correct. The indicating bit reverses whenever the counter overflows. The fundamental idea of Drowsy-RI is to change prediction at the moment of two consecutive miss-predictions. Figure 26 describes the prediction accuracy of Drowsy-RI. On average, Drowsy-RI succeeds to predict 59.3% of re-access intervals and the accurate rate is over 80.4% for `bfs`.

4.4 Evaluation Results

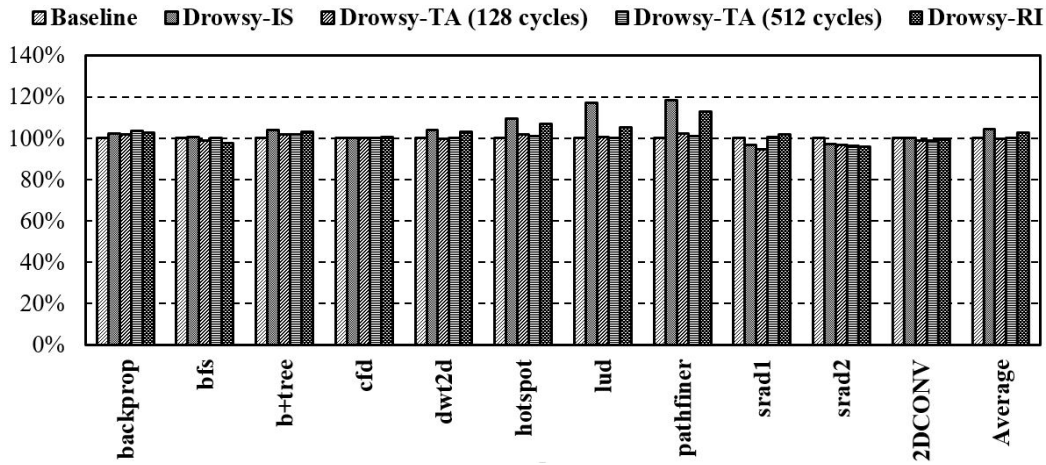
4.4.1 Performance Overhead

We evaluated three drowsy policies studied in this work and compare them with the baseline (w/o drowsy RF). Drowsy-IS focuses on maximizing the leakage energy reduction. Drowsy-TA attempts to reduce the performance overhead significantly. Drowsy-RI adapts the re-access intervals to balance RF leakage reduction and performance degradation.

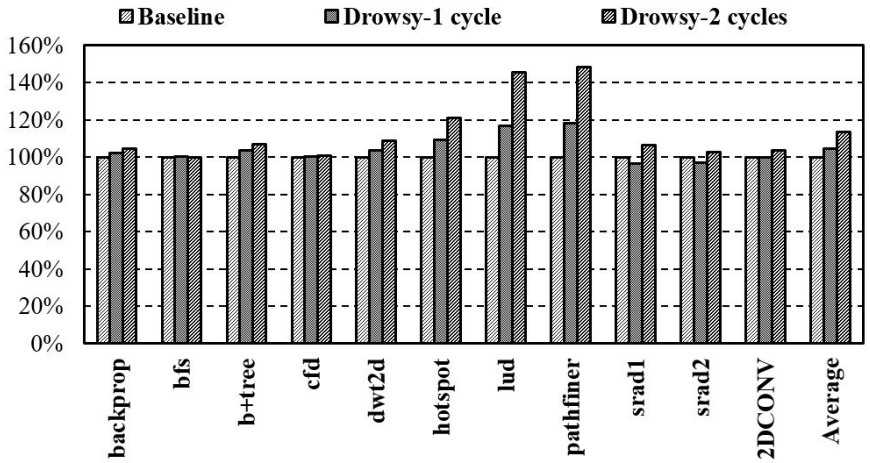
As shown in Figure 27a, Drowsy-IS results in 4.4% additional execution cycles on average as compared to baseline. Two of the evaluated benchmarks, `lud` and `pathfiner` exhibit severe performance degradation. The reason is that the frequencies of register accesses in those benchmarks are much higher than other benchmarks. The large amount of latencies generated by re-activating registers from the drowsy state can not be completely hidden and thus the performance is degraded significantly. Figure 27c depicts that `lud` and `pathfiner` have more register accesses per cycle than the rest of benchmarks.

Figure 27b shows that the performance degradation with 1 (default) and 2 cycles re-activation latency. As we can see, when the re-activation latency increases to 2 cycles, the average performance degradation of the Drowsy-IS is increased dramatically to 13.6% on average, which can also lead to significant energy overhead. Although GPUs typically can hide latencies by switching between massive executable threads, frequently waking registers from drowsy state could lead to a great number of delay cycles and applications will suffer from performance degradation if latencies can not be fully hidden.

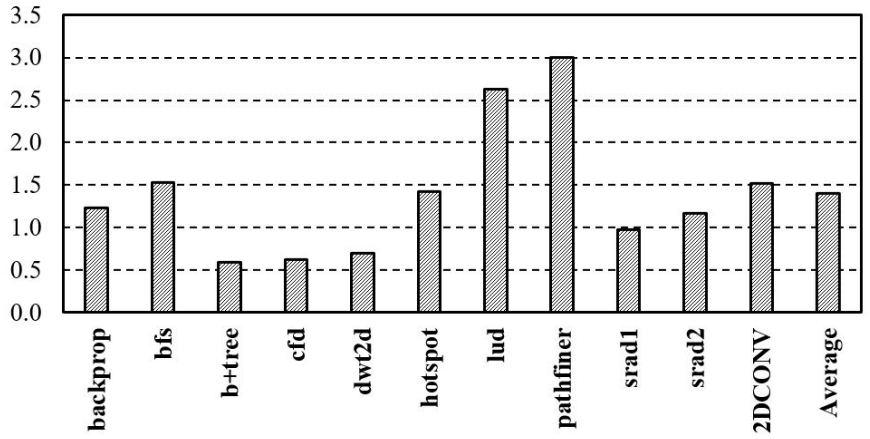
Luckily, the performance overhead becomes negligible when Drowsy-TA is applied with 128 and 512 awaking cycles. Ideally, Drowsy-RI can eliminate as many re-



(a) Normalized performance of three policies



(b) Performance degradation with 1 and 2 cycles re-activation latency



(c) Register accessing times per cycle

Fig. 27. Performance results for different drowsy policies

activation latencies as Drowsy-TA does. However, due to mispredictions, Drowsy-RI is only able to partially identify short re-access intervals, and thus results in medium performance overhead of 2.7% on average, which is 1.8% lower than Drowsy-IS but 2.5% higher than Drowsy-TA.

4.4.2 Leakage Energy Reduction

Figure 28 shows the RF leakage energy reduction of the three policies. Two aspects contribute to the total RF leakage energy saving. One is the power gating of unallocated registers at the beginning of the execution and another is to place registers into the drowsy state at run-time. As expected, Drowsy-IS achieves the most RF leakage energy reduction among the three policies, which is up to 94.0% and 91.7% on average.

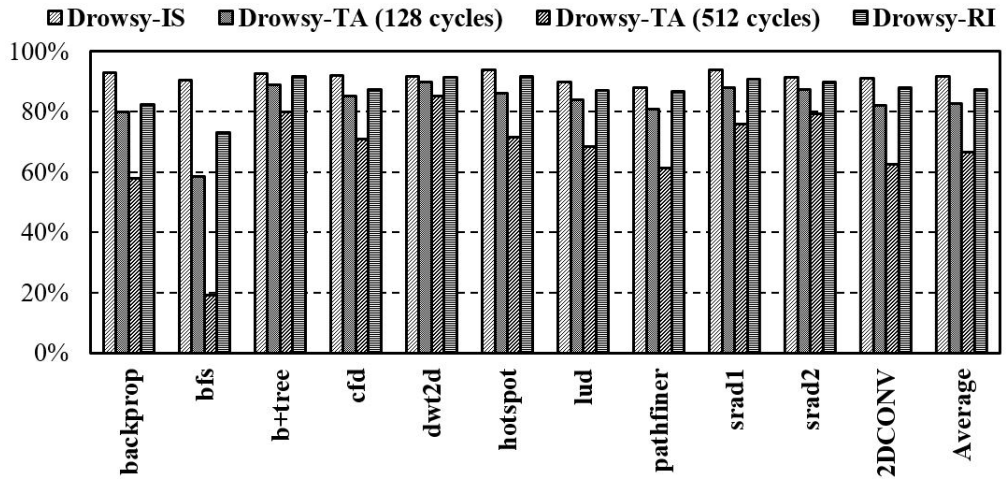


Fig. 28. RF leakage energy reduction

For Drowsy-TA, since part of the opportunities to save leakage energy is wasted by keeping each register awake for a fixed period whether the register is re-accessed or not, it leads to less RF leakage reduction. Drowsy-TA with an awake interval of 128 cycles reduces leakage energy up to 90.9% and 82.8% on average. The reduction

rate decreases to 66.6% on average if the awake interval is increased to 512 cycles. In particular, Drowsy-TA (512 cycles) dramatically weakens the ability of energy reducing for `bfs`. By examining re-access pattern of `bfs`, we find that 99.8% of register re-accesses in `bfs` arrive within 512 cycles, thus almost all allocated registers are put into the operational mode during the entire execution. In this case, power gating unallocated register becomes the only source of energy saving for `bfs`.

As expected, Drowsy-RI results in a RF leakage reduction between the Drowsy-IS and Drowsy-TA schemes. More specifically, Drowsy-RI saves 4.3% more leakage energy than Drowsy-TA (128 cycles) and is worse than Drowsy-IS (4.5% less energy saving). Furthermore, Drowsy-RI decreases the performance overhead compared to Drowsy-IS as described in Section 4.4.1.

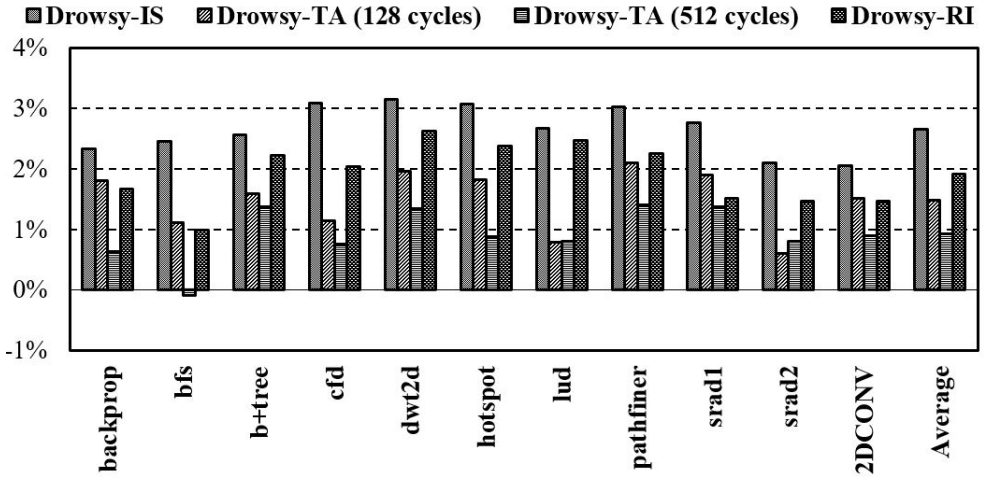


Fig. 29. RF dynamic energy increasing

The main overhead of drowsy technique contains two parts, the additional dynamic energy and execution cycles spent on re-activating registers from the drowsy mode. As shown in Figure 29, Drowsy-IS leads to 2.7% more RF dynamic energy on average. Referring to the normalized number of accesses to registers in the drowsy mode shown in Figure 30, Drowsy-TA and Drowsy-RI decrease the number of accesses

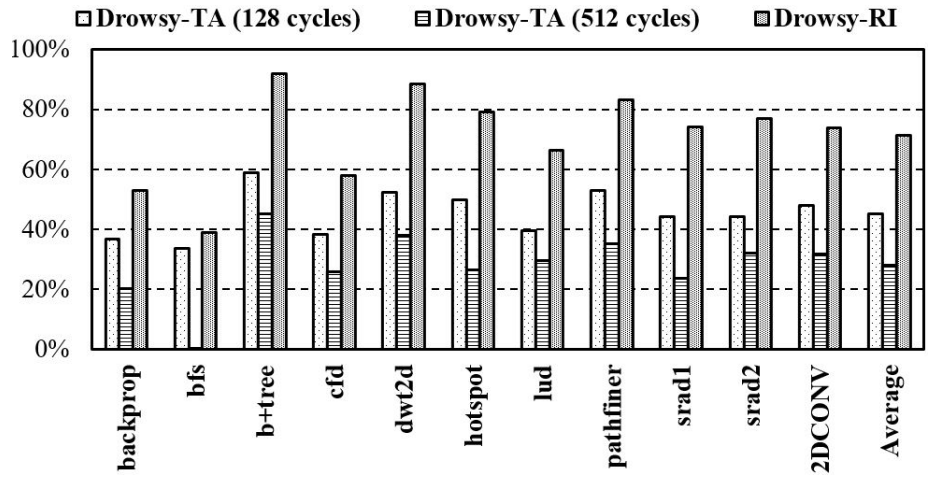


Fig. 30. The normalized number of accesses to registers in drowsy mode

to the drowsy mode registers. Therefore, the dynamic energy overhead is reduced to less than 2% for Drowsy-TA (128 cycles) and Drowsy-RI and 1% for Drowsy-TA (512 cycles). Even though Drowsy-IS shows a slightly higher dynamic RF energy increasing than other policies, the overwhelming power of RF leakage energy reduction makes it the best choice to reduce total RF energy (49.5% on average shown in Figure 31).

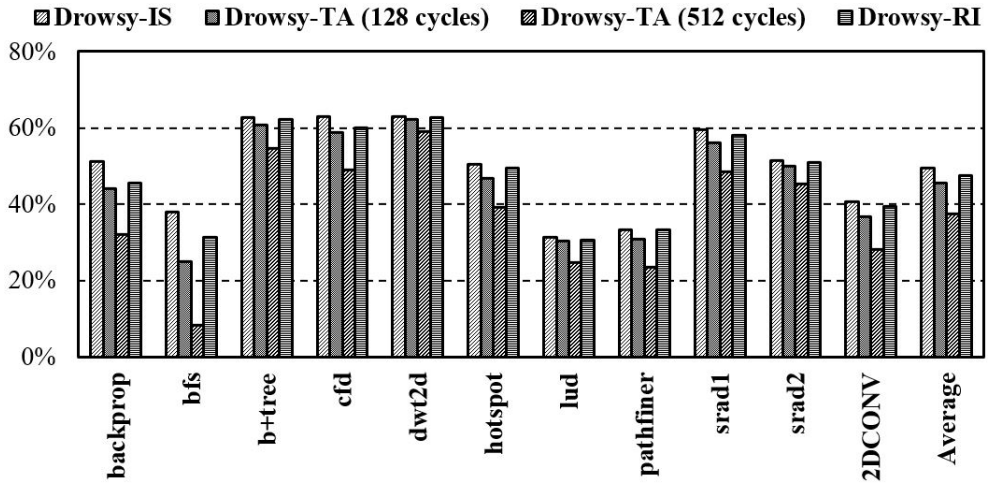


Fig. 31. RF energy reduction

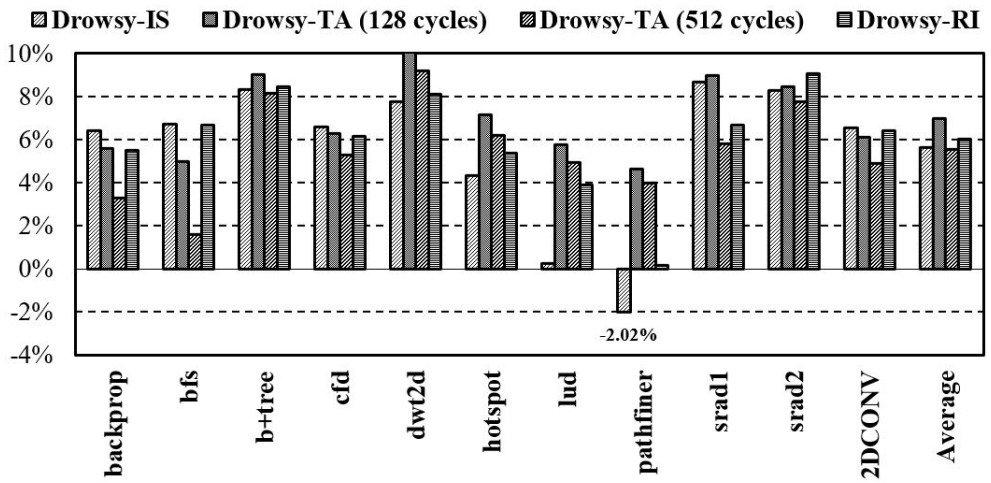


Fig. 32. GPU energy reduction

The performance overhead can also negatively affect the energy reduction of entire GPU (due to the increased execution time). To reduce the total GPU energy, the RF energy reduction must be larger than the energy overheads due to the increased execution time. Figure 32 shows the total GPU energy reduction. For most benchmarks, Drowsy-TA (128 cycles) is the best in reducing the total GPU energy due to its low performance overhead and decent RF energy reduction. Although Drowsy-TA (512 cycles) has even lower performance overhead, its poor ability in RF energy reducing limits the overall GPU energy saving. For benchmarks which are experiencing severe performance degradation under Drowsy-IS, such as `lud` and `pathfinder`, the total energy consumption of GPU is sharply increased. On the other hand, for benchmarks whose performance barely changes under different drowsy policies like `bfs`, Drowsy-IS is able to effectively improve GPU energy efficiency via the RF energy reduction. On average, Drowsy-TA (128 cycles), Drowsy-RI and Drowsy-IS reduce the GPU energy by 7.0%, 6.0% and 5.6%, respectively.

CHAPTER 5

GPU EXECUTION UNITS POWER-GATING STRATEGIES

5.1 Introduction

We examine the underutilization of three execution units in 10 GPGPU benchmarks. The results show a great waste of execution resources. As shown in Figure 33, the integer units stay idle during around 61% of the total execution time and the floating point units are unused in 92% of execution cycles. SFUs are free in 75% of the execution period. The remarkable leakage energy consumption and the low utilization of execution units convince us to power-gate spared execution units intelligently and help GPUs to relieve the serious energy dissipation issue. We propose to uncover the inherent opportunities for the power-gating on the energy-efficiency enhancement without involving any re-schedule techniques. Researches in [7, 56] depend on re-scheduling instructions to reconstruct the instruction sequences and intentionally generate long idle durations. Unlike actively utilizing re-schedule techniques in the previous studies, this work focuses on analyzing the existing idleness of the execution units with respect to the default instruction sequences. According to the inherent idleness, we explore appropriate power-gating strategies to save GPGPU leakage energy. This work is to guide the future GPU energy studies regarding the execution units power-gating. Instead of involving additional microarchitectures for complex re-scheduling logic, only basic counters are required for the proposed method which is then applicable with low hardware overhead. Besides the power-gating strategies, the idleness pattern also inspires us to increase the number of integer units in an SM for performance and energy efficiency improvement.

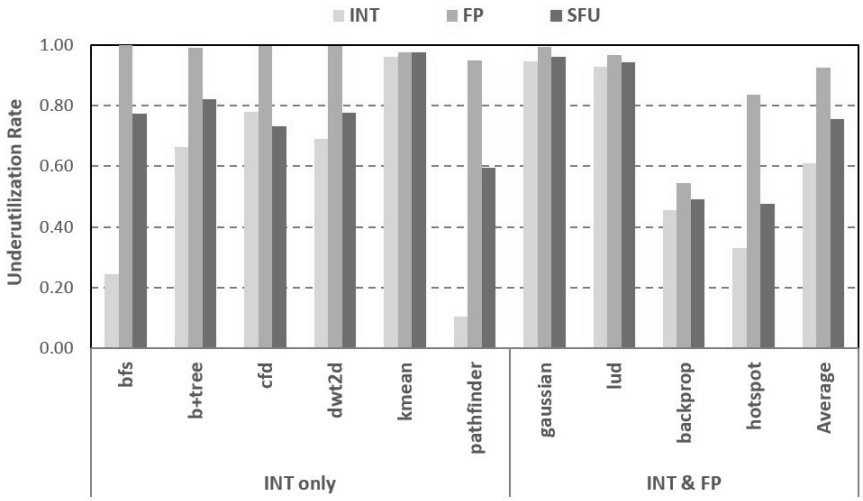


Fig. 33. The Underutilization rate of execution units

5.2 Related Work

GPGPUs are becoming promising platforms for accommodating parallelizable compute-intensive applications. Indeed, the support of a massive number of execution units and abundant bandwidth enable GPGPUs to provide an extremely high throughput by concurrently executing thousands of threads; however, the energy efficiency can become an urgent concern if the GPGPUs attempt to further improve the performance. Moreover, the great amount of execution units are responsible to a considerable portion of the total energy consumption and they unfortunately produce many idle cycles resulting energy wasting. Many researchers have studied GPGPU's energy-efficiency [57, 7, 9, 58, 34, 59, 60, 56]. Most of the researches discovered the underutilization of various GPU components such as register files and execution units and proposed to power gating the idle resources for energy saving. [57] propose to shut down unused fragments or putting them into the low power modes to reduce leakage energy consumption of GPU register files. [58] implements the power-gating at the SM level granularity. They monitor the activity of entire SMs and shut down

the unoccupied SMs to improve energy-efficiency. Some other solutions operate the power-gating at a much finer granularity [7, 56]. They explore the idleness of the execution units and design strategies to apply power-gating down to per SIMT lane and minimize the leakage energy wasting. By comparison, this work studies the inherent idleness of the integer and floating point units of SP as well as the SFU. To reduce GPU leakage energy dissipation, we have studied two different policies operating the power-gating adaptively and efficiently on different execution units.

5.3 Power-gating Strategies

Power-gating technique has been widely demonstrated to be effective on the leakage energy reduction. In this work, we use the traditional power-gating strategies to save execution units leakage energy on GPGPUs. We operate the power-gating at a fine granularity and shut down the integer unit and floating point unit per SIMT lane individually. The four SFUs in an SM are also able be power-gated separately. t_{break_even}, t_{wakeup} and t_{idle_detect} are three parameters related to the execution units power-gating technique. The t_{break_even} represents the least number of cycles the power-gated mode should last. The leakage energy saving during t_{break_even} is equal to the energy overhead of turning off and on the execution units. As long as the power-gated mode lasts longer than t_{break_even}, the leakage energy reduction is attained; otherwise, the energy overhead of the power-gating exceeds the insufficient saving and negatively affects the energy efficiency instead. The t_{wakeup} is the number of cycles spent on waking up execution units. These extra cycles can lead to negative performance impact. Both t_{break_even} and t_{wakeup} can be calculated by using the formulas in [61] and vary with parameters of circuit components. The typical value of t_{break_even} lies between 9 to 19 cycles and t_{wakeup} is from 3 to 9 cycles [61]. In this work, the evaluation starts with t_{break_even} of 10 cycles and t_{wakeup} of 3 cycles and

continues with other t_{break_even} (5 and 20 cycles).

The power-gating decisions are made according to the threshold t_{idle_detect}. We can tune the t_{idle_detect} to achieve the balance between the leakage energy reduction and performance loss. A large t_{idle_detect} filters the short idleness of execution units to avoid performance degradation, but consequently misses opportunities of saving ever more leakage energy. We first evaluate with t_{idle_detect} of 0 cycles called immediate power-gating (IPG). The IPG put execution units into power-gated mode immediately as soon as they are not occupied. The IPG is capable to maximize the leakage energy reduction if most of the idleness is longer than t_{break_even}; otherwise, the IPG could hurt both energy efficiency and performance if a lot of short term idleness shows up. Due to the various idleness patterns, the optimal t_{idle_detect} can be varied for different execution units. In order to achieve further leakage energy saving, we examine the variation of energy reduction with different t_{idle_detect} (2 and 5 cycles), called idle detect power-gating (ID-PG). Based on how busy a certain type of execution unit is, the length distribution of the execution unit idleness is different and t_{idle_detect} varies accordingly.

Due to that the power-gating strategies studied in this work can be completely supported by several simple counters and avoid complex microarchitecture and logic, the hardware overhead is negligible. Moreover, the performance overhead that is mostly caused by the wake-up latency t_{wakeup} is negligible as well. An issued instruction is not going to be executed immediately. Instead, several cycles are taken to request operands of the instruction from the register file and collect them in the operand collector. The instruction is not forwarded to the execution unit until all operands are ready in the operand collector. By noticing the time gap between the issues and execution stage, the SIMT lanes that will be used to execute the instruction can be woken up in advance right after the instruction has been issued. Getting operand

ready can typically take up to 10 cycles, which is longer than t_{wakeup}. Therefore, t_{wakeup} successes to overlap itself with the period of fetching and collecting operands and thus is not expected to have a negative impact on performance. Moreover, since there is no warps and instructions rescheduling in our methods, the performance is not influenced by any rescheduling policy. Overall, both hardware and performance overheads of the power-gating strategies are negligible, which is also confirmed in our experiments.

5.4 Evaluation Results

5.4.1 Leakage energy reduction for different types of execution units

Figure 34 shows that the leakage energy of integer units can be reduced up to 93% and 24.9% on average, when $t_{break\_even} = 10$ and $t_{idle\_detect} = 0$. For benchmarks such as `bfs`, `pathfinder`, `backprop` and `hotspot`, the setting of $t_{idle\_detect} = 0$ leads to more leakage energy dissipation from integer units and thus hurts the energy efficiency. This is because there are too many short-term idleness generating inevitable energy overheads.

The portion of the idleness over 10 cycles for integer units is shown in Figure 35. Compare to the other benchmarks, `bfs`, `pathfinder`, `backprop` and `hotspot` show a smaller portion of idleness lying above 10 cycles (9.0%, 2.9%, 7.4% and 11.5% respectively). `bfs`, `pathfinder`, `backprop` and `hotspot` fail to contribute to the energy efficiency due to the lack of long-term idleness. What's worse, the execution units are frequently switching on and off resulting in the overwhelming energy overhead and the degradation of the energy efficiency. As shown in Figure 34, integer units leakage energy reduction is -28%, -22%, -44% and -22% for `bfs`, `pathfinder`, `backprop` and `hotspot` respectively.

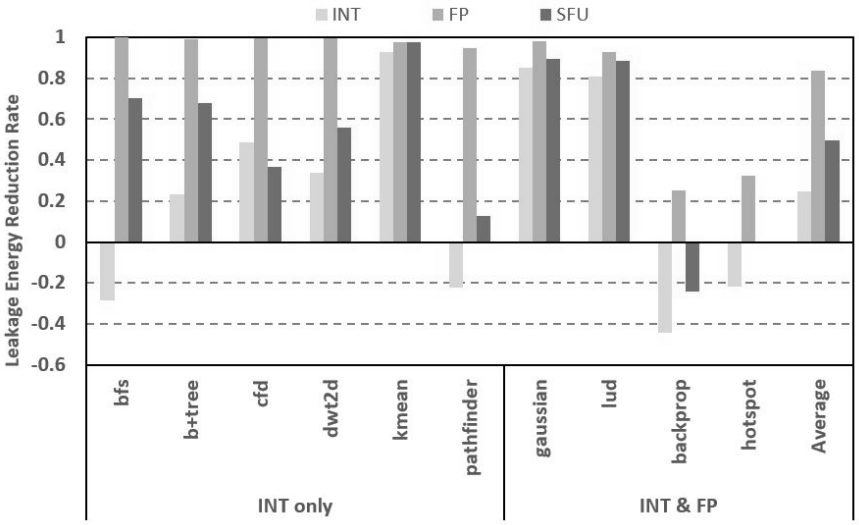


Fig. 34. Leakage energy reduction for different types of execution units (t_{break_even} = 10, t_{idle_detect} = 0)

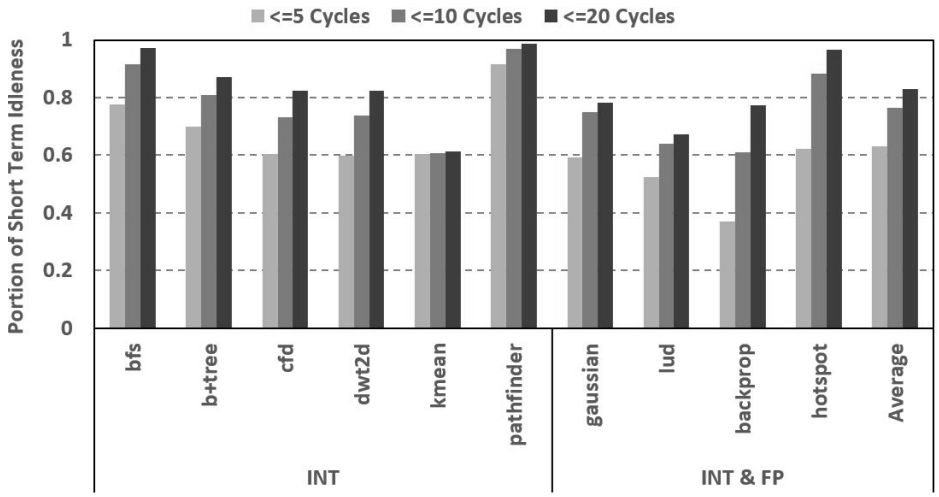
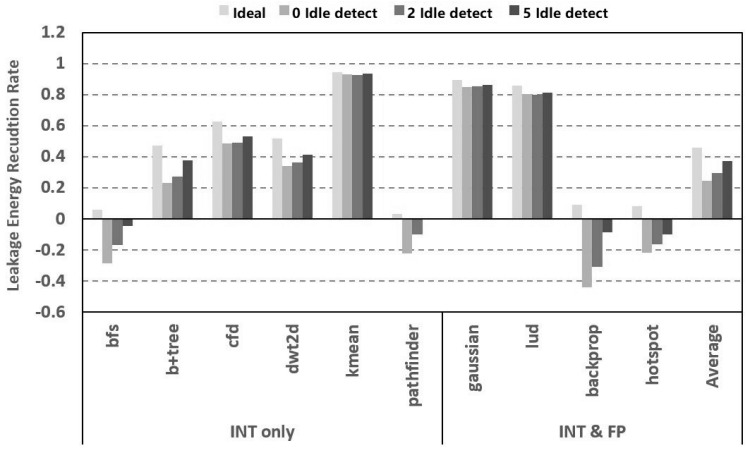
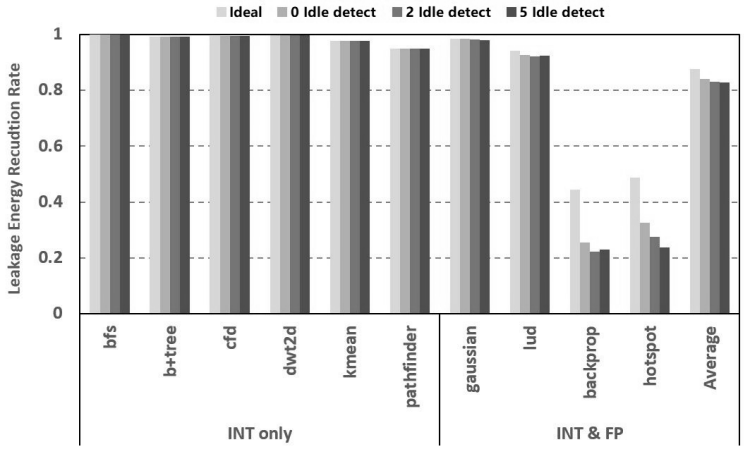


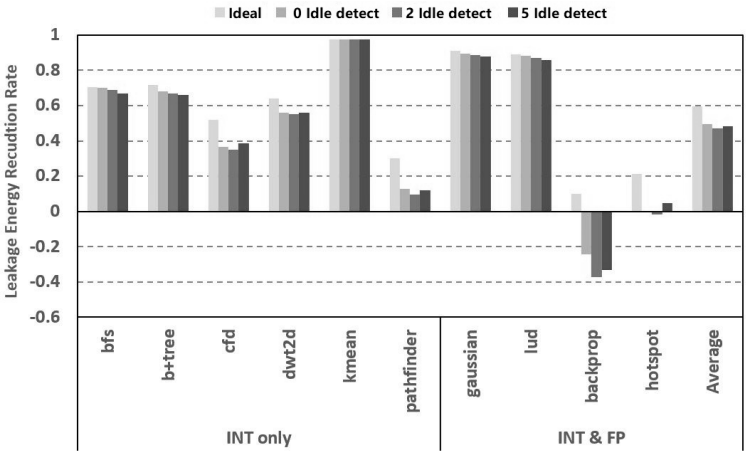
Fig. 35. The portion of the idleness below 5, 10 and 20 cycles for integer units (t_{break_even} = 10, t_{idle_detect} = 0)



(a) Integer units leakage reduction rate



(b) Floating point units leakage reduction rate



(c) SFUs leakage reduction rate

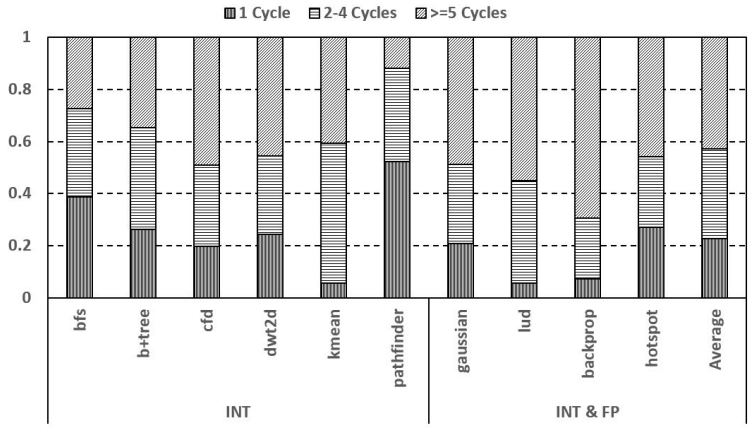
Fig. 36. Execution units leakage energy reduction rate for different idle detect time (t_{idle_detect} = 0, 2, 5)

As compared to the integer units, the power-gating works much better on reducing the leakage energy of floating point units and SFUs (83.9% and 49.5% leakage energy saving for floating point units and SFUs are reduced respectively). Since 93% of the idleness of SFUs is shorter than 10 cycles for `backprop`, the very high occurrences of short-term idle periods causes the leakage energy increasing of SFUs for `backprop`.

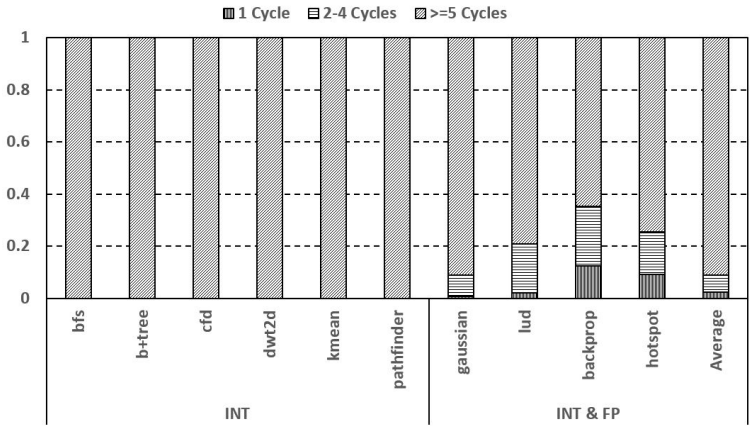
5.4.2 Leakage energy reduction for different t_{idle_detect}

We next evaluate the power-gating strategies with different idle detect time (t_{idle_detect} = 0, 2, 5 cycles). Figure 36a depicts that increasing t_{idle_detect} can improve the integer units leakage energy saving, as more and more short-term idle intervals are filtered by longer t_{idle_detect} and the energy overhead is compromised. When increasing t_{idle_detect} from 0 to 5 cycles, the average leakage energy reduction boosts from 24.9% to 37.2%. The breakdown of idleness for integer units is shown in Figure 37a. When t_{idle_detect} increases from 0 to 2 cycles, the 2-cycle threshold can rejects to power-gate as encountering 1 cycle idleness. The overhead can be remarkably removed owing to that 23% of total idle durations on average is only 1 cycle. If further increasing t_{idle_detect} to 5 cycles, even more short-term idleness with intervals of 2, 3 or 4 cycles can be filtered. As shown in Figure 37a, 34% of total idleness on average is 2, 3 or 4 cycles and therefore the energy overhead can be further reduced.

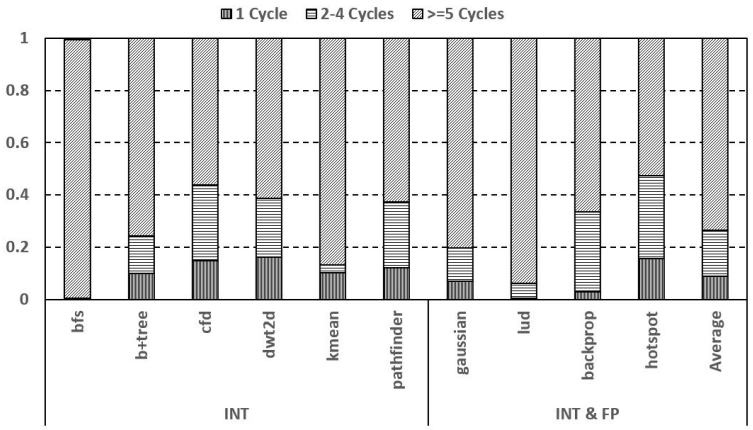
Figure 36b and Figure 36c show that operating the power-gating with shorter t_{idle_detect} can lead to better energy-efficiency for floating point units and SFUs. Figure 37b and Figure 37c explain that there are only limited number of short idle intervals can be gated by t_{idle_detect} when it becomes longer. Specifically, setting t_{idle_detect} to 5 cycles can only remove the power-gating overhead from 2.5% (1 cycle) and 6.5% (2-4 cycles) of the total idleness for floating point units. And, for SFUs, 26% more



(a) Idleness breakdown of integer units



(b) Idleness breakdown of floating point units

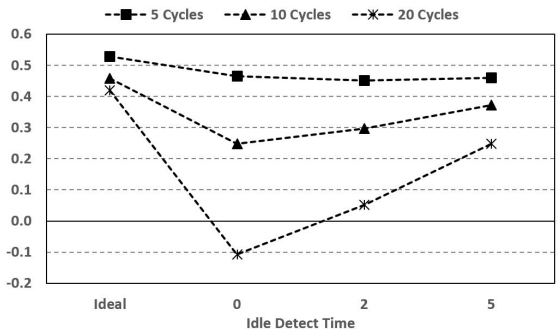


(c) Idleness breakdown of SFU

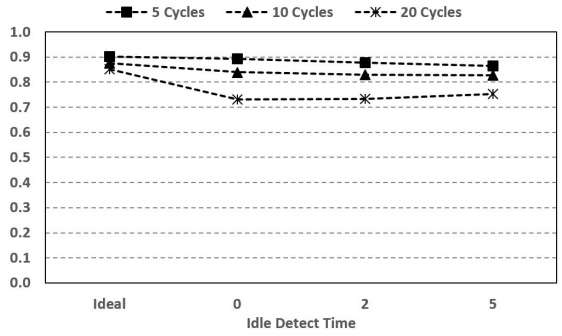
Fig. 37. Idleness breakdown

idleness can be found in total (9% 1 cycle idleness and 17% 2-4 cycles idleness). Once the longer t_{idle_detect} is unable to dig more enough short-term idleness to optimize the power-gating strategies, it wastes additional cycles on waiting the power-gating decision rather than power-gates immediately to enhance the energy efficiency.

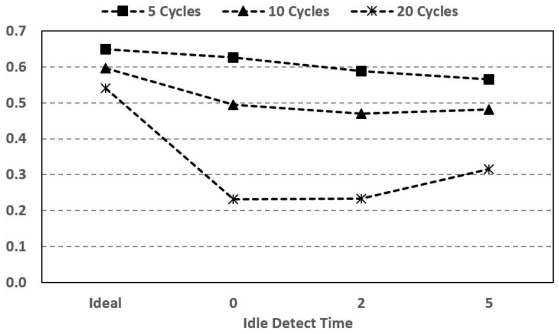
5.4.3 Sensitivity analysis of t_{break_even}



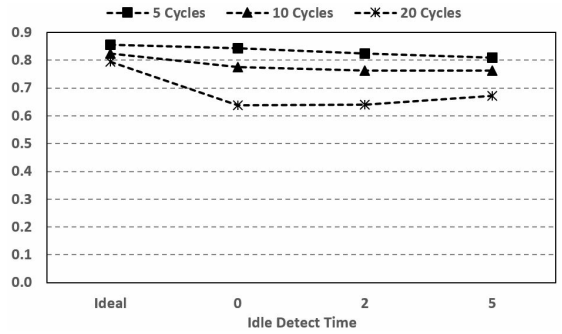
(a) Integer units leakage energy reduction with different t_{break_even}



(b) Floating point units leakage energy reduction with different t_{break_even}



(c) SFUs leakage energy reduction with different t_{break_even}



(d) Overall leakage energy reduction of all execution units with different t_{break_even}

Fig. 38. Leakage energy reduction with different t_{break_even}

As shown in Figure 38, the longer idle detect time becomes the optimal choice for all execution units, when t_{break_even} goes up from 5 cycles to 20 cycles. Especially, the power-gating technique achieves the most reduction of the leakage energy of floating point units when 5 cycles idle detect time is chosen instead of 0 cycle. As raising t_{break_even}, the energy budget of power-gating execution units increases. In this case,

the energy overhead reduction from avoiding short-term idle periods exceeds the extra energy paid on reaching the idle detect time.

Based on the results shown in Figure 38b, the evaluated power-gating schemes are able to approach a satisfactory leakage energy saving of floating point units. The average saving is very close to the ideal case. However, the increasing of the break even time weakens the power of the power-gating and pulls the leakage energy saving against the best case. Not surprisingly, the overall leakage energy saving of all execution units follows a trend similar with the floating point leakage energy reduction as represented in Figure 38d. As compared to the ideal case (85.5%, 82.4% and 79.3%), 84.3%, 77.5% and 63.1% reduction of execution units leakage energy can be reached for the break even time of 5, 10 and 20 cycles, respectively.

5.4.4 Add Green SP(s) for enhancement of both performance and energy efficiency

According to the experimental results in Figure 33, the integer units are overall demanding resources. Therefore, increasing the number of integer units in an SM can relieve the pressure of integer computation and potentially improve the performance. Moreover, by adding Green SPs which are dedicated to integer computations, more normal SPs can yield to floating point computation. For the applications including both integer and floating point computation, the waiting line of floating point units might be cut down and the performance could be boosted. The energy overhead of introducing more integer units can be negligible as only 0.1% of the total GPU energy is consumed by the integer units. In this work, we evaluate the performance when one or two Green SPs are added to an SM. Figure 39 shows the results. For the applications working on integer computation only, the performance improvement is slight. However, `bfs` and `pathfinder` enjoy a little benefit on the performance (1.8%

and 5.2% improvement respectively) due to their extreme demand on integer units. As shown in Figure 33, integer units are unoccupied during only 24.5% and 10.6% of the total execution time for `bfs` and `pathfinder` respectively. For applications including both integer and floating point computation, extra integer units can be more helpful on boosting performance as the Green SPs can free the normal SPs to the floating point computation and the floating point instructions that are stalled for lack of normal SPs then can acquire the resources earlier. For example, extra integer units eventually achieve 5.6% and 14.4% performance improvement for `backprop` and `hotspot` that rely on both integer and floating point units.

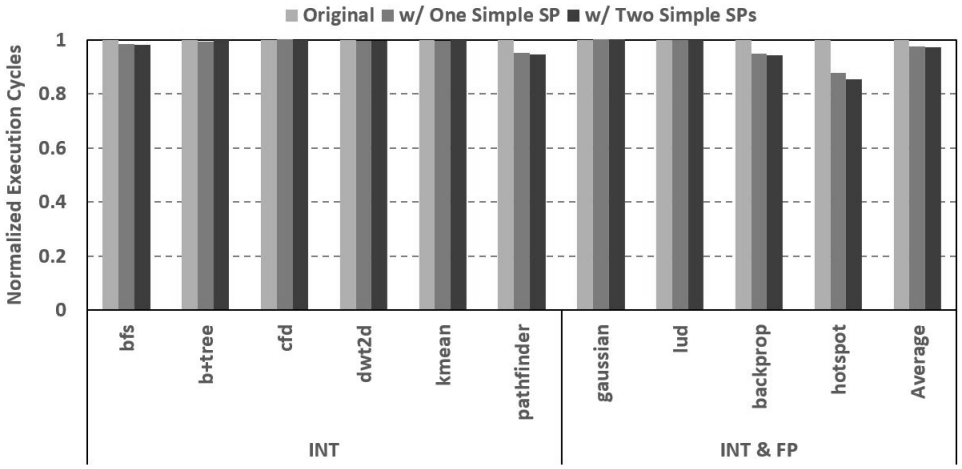


Fig. 39. Normalized execution cycles w/ simple SP(s)

CHAPTER 6

GPU SIDE-CHANNEL ATTACKS

6.1 Introduction

6.1.1 Brief Description of pSCA and pacSCA

Recent researches have found that the SCAs are practical and the secret information can be revealed on GPUs [62, 63, 64, 65, 66, 67]. All these SCAs adopt a correlation analysis based on a certain kind of relationship between the key-related information (e.g. the number of unique cache lines in [66]) and observed matrix (e.g the execution time in [66]) to recover the AES key. As one of the most typical SCAs on GPUs, [66] successfully establishes the relationship between the execution time of an AES encryption kernel and the AES encryption key by observing that the kernel execution time is linearly proportional to the number of unique cache line requests that is highly dependent on the encryption key. They then construct a side-channel timing attack to recover all the 16 bytes AES secret key on a specific NVIDIA GPU by using a statistical analysis [66]. To thwart this SCA, the work in [68] proposes the Randomized Coalescing (RCoal) techniques to effectively protect GPUs with varied performance overheads. To hide the relationship utilized by this SCA, the memory coalescing is randomized in several ways including the number of subwarps, the threads assigned to each subwarp, or a combination of both. The introduced randomness is shown to improve GPU security by making it much harder to discover the correlation via the statistical analysis.

In this work, we propose two novel side-channel attacks that information leaked

from GPU performance profiling can be extracted when executing on an SIMT-based GPU to fully recover the encryption secret key. Unlike power and timing side-channel attacks, instead of bridging the number of unique cache line requests with the key via a differential of power consumption or execution time, our side-channel attacks accomplish straightforward key recovery procedure by sampling the exact number of unique cache line requests during the run-time and simply checking all 256 possibilities for each byte of all 16-byte AES key to determine the correct answer. The number of samples and time needed for recovering the key are dramatically reduced while the accuracy can be always guaranteed. Moreover, the profiling-based side-channel attack has great scalability, as the profiling time only increases slightly with no additional samples required when the length of the AES key scales up.

We first propose a profiling-based side-channel attack (pSCA). Compared to the prior work in [69] of GPU side-channel attack, for which only GPUs supporting the execution of multiple kernels from different programs are vulnerable since the spy program needs to implant malicious kernels to probe victim program; creating contention between spy and victim programs is unnecessary for the attack proposed in this work, which can be more generally applied to GPUs. Furthermore, instead of profiling coarse-grained information or collecting only one sample for each kernel executed, we propose a fine-grained profiling strategy which can break the limitation of sample frequency set up by the profiling APIs and get more details during the kernel execution for an efficient and accurate key recovery. We have implemented a run-time profiling tool, which can sample two performance matrices, the number of the memory load and memory store requests. Based on the two performance matrices, the number of unique memory load requests in the last round encryption for each byte except the first byte then can be determined. The 16-bytes AES key can be recovered byte by byte. By calculating the number of unique memory load requests with 256

possibilities of a single byte of the AES key and comparing to the profiled number, the number of possibilities of corresponding byte key can be narrowed down. The number of possibilities can be further decreased to 1 by repeating the same procedure with different input data for several times. We further enhance pSCA to overcome RCoal which can protect the GPUs from our original pSCA. Due to the two reasons: 1) the possible number can be reversely computed with a guessed key value, a reverse table, and the cipher-output which are all known by the attacker and 2) the exact number of unique cache line requests can be sampled, simply checking all 256 possibilities for each byte of all 16-byte AES key will determine the correct answer. For GPUs protected by the RCoal techniques, indeed, the pSCA alone cannot crack the right AES key directly because the sampled number has been obfuscated by the randomness of the memory coalescing. However, we propose a boundary check mechanism to be used with the pSCA to effectively crack the AES key with RCoal. To this end, the RCoal guarded GPUs will still be unable to guarantee the security completely. Moreover, the RCoal techniques increase the number of subwarps to improve the security, which comes at the cost of the performance. Since the boundary-check pSCA can successfully attack RCoal techniques with various number of subwarps, including 2, 4, and 8, except for 16 though. Thus, to fully protect the system against the pSCA with the boundary checking, the hardware designers will be forced to pick the 16 subwaprs configuration, resulting in much larger performance loss for GPUs.

We then propose a Profiling-Assisted Correlationbased side-channel attack (pac-SCA). Existing correlation-based side-channel attacks take advantage of the correlation between the number of unique memory load requests and the execution time [66] or power consumption [67] to recover the secure key. The weakness of these side-channel attacks is that a large number of samples are needed to observe the weak correlation making them fragile to defense strategies. As comparison, our profiling-

assisted mechanism utilize a much stronger relationship which connects the number of memory loads in the last round to the total amount of memory loads. The exploited relationship delivers significant correlation and interacts with minimum noises. Naturally, the demand on the number of samples for reconstructing the key is remarkably reduced. Additionally, the scalability of the pacSCA is satisfactory as there is only a slight increase on the number of samples with respect to scaling up the length of the AES key. A run-time profiling tool has been developed to sample the total number of memory load requests accurately. Through the correlation analysis on the profiled traces, the secret information can then be extracted by the attacker. Since there is no statistical analysis involved in the pSCA, the attack performance is not affected by any noise and thus the success rate of the AES key recovery can be always guaranteed to be 100%. By comparison, although the success rate can converge to 100% in case of enough samples, the correlation analysis prevents the pacSCA to achieve the perfect success rate. Moreover, the pSCA requires much smaller number of samples than the pacSCA (9 samples for the pSCA **vs.** 10000 samples for the pacSCA). The downside of the pSCA is that the detailed number of unique memory load requests for each load instruction in the last round have to be profiled. Owing to the poor sample resolution, the profiling tool needs to profile the same encryption case multiple times until all details to form a sample are captured. Unlike the pSCA, the pacSCA only profiles the total number of memory load requests which can be easily sampled at the end of each encryption kernel. Consequently, the pSCA spends much longer time on getting a sample than the pacSCA (1 sample every 4 seconds for pSCA **vs.** 10000 sample every 5 seconds for the pacSCA). Besides this, the pacSCA also outperforms the pSCA on the total key recovery time (5 seconds for pacSCA **vs.** 4 seconds $\times$ 9 = 36 seconds for the pSCA). Furthermore, the complexity and detailing of the profiling strategy makes the pSCA fragile to both hardware and software defense mechanisms that can

interrupt the profiling. On the other hand, due to the simplicity of the profiling tool and the significant correlation, the pacSCA is able to survive from state-of-the-art countermeasures such as the Rcoal [68].

6.1.2 AES GPU Implementation

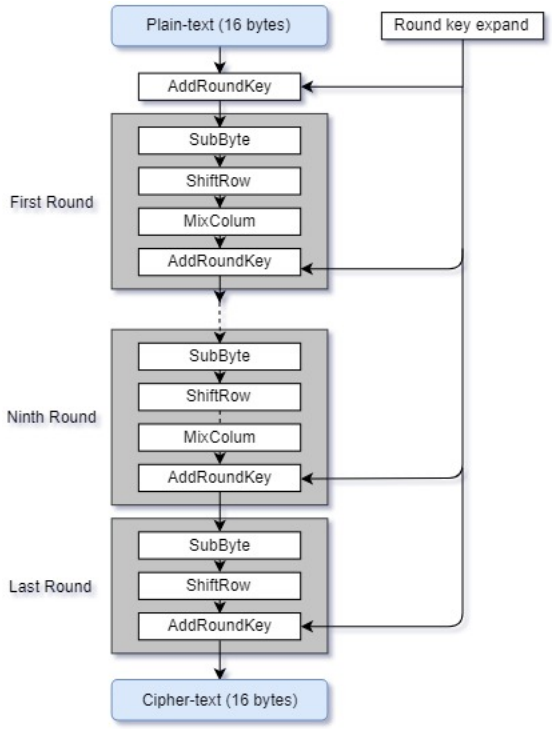


Fig. 40. The operations of encryption rounds in AES.

The size of the basic encryption block in the AES algorithm is fixed to 128 bits. According to the security requirement, the encryption key length can be tuned from 128 to 196 and 256 bits and the total encryption round is 10, 12, and 14 respectively. In this study, we focus on the 128-bit AES which processes 10 encryption rounds. Initially, the original 16-byte key is expanded into 160-byte round keys for 10 rounds of operations. In the first round, the first round key is XORed with the 16-byte plain-text to generate the initial state. The same operations composed of SubByte,

ShiftRow, MixColumn, and AddRoundKey are executed in each encryption round except the last round. The 10th (i.e. last) round, which skips MixColumn, only contains SubByte, ShiftRow and AddRoundKey operations. Figure 40 shows the operations of encryption rounds in AES. The LUT (Look Up Table) based AES is one of the most classical implementations on many general-purpose computing platforms, in which the SubByte, ShiftRow, MixColumn operations are completed by four lookups to four different tables. Due to skipping the MixColumn operation, the last round only visits one special T-table instead. To complete the encryption round, the AddRoundKey operation is XORed with the corresponding round key at the end of each round after table lookups. To take advantage of the high throughput, the GPU implementation of the AES algorithm assigns one 16-byte block of the plain-text to each thread for independent encryption. In this way, a warp consisting of 32 threads can work on 32 different blocks concurrently. Our GPU implementation of the AES is the same as that used in [66]. Figure 41 depicts the AES algorithm executing on a GPU.

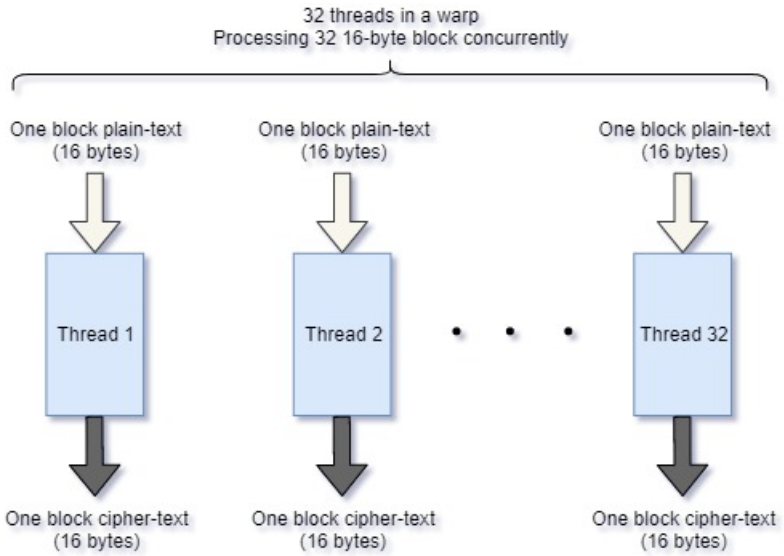


Fig. 41. The GPU based AES implementation.

6.1.3 The RCoal Techniques

To defend GPUs against SCAs successfully demonstrated in [62, 63, 64, 65, 66, 67], Kadam et al. [68] propose the RCoal techniques to enhance security at the cost of the performance overhead that is tunable. The RCoal techniques randomize the coalescing logic, making it hard for the attacker to guess the correct number of coalesced accesses. Specifically, the size of the subwarp and the threads assigned to each subwarp are randomized. Accordingly, they propose three coalescing randomization strategies: fixed sized subwarp (FSS), random-sized subwarp (RSS), and random threaded subwarp (RTS).

FSS separates a warp into several subwarps and applies the memory coalescing independently within each subwarp. From the attacker's point of view, the number of subwarps is always 1. By hiding the real number of subwarps from the attacker, FSS makes it impossible to directly profile or compute the correct number of coalesced memory accesses and therefore mitigates the GPU's vulnerabilities. Although FSS increases the security, the subwarps with shrunken size waste the opportunities to optimize the memory accesses, resulting in an increased number of memory accesses and hence performance loss. What's worse, the number of subwarps is easy to be guessed or measured by the attacker and in this case, FSS offers no security improvement other than performance degradation.

To overcome the flaw of FSS, RSS is proposed to further randomize the subwarp size. Unlike FSS that breaks up a warp into subwarps of a fixed size, RSS assigns various number of threads to each subwarp and uses the skewed distribution to randomly determine the subwarp sizes. Thus, the attacker cannot explore the correct number of coalesced memory accesses without the details of both the number and sizes of subwarps, which are difficult to be revealed.

To cooperate with FSS and RSS and involve even more randomness, the assignment of threads to each subwarp is randomized in RTS. To be more specific, RTS randomly reorders the threads in the original warp and reassigns them to each subwarp. Applying RTS with FSS or RSS (i.e., FSS+RTS and RSS+RTS) makes the number of coalesced memory accesses even harder to be calculated by the attacker.

Kadam et al.'s research [68] shows that the RSS along with RTS outperforms other three coalescing randomization schemes (FSS, RSS, and FSS+RTS) on security enhancement and are demonstrated to enable 24 to 961 times improvement in the security against the correlation timing attacks with 5% to 28% performance degradation.

6.2 Related Work

Jiang et al. [66, 63, 64] proposed several GPU side-channel attacks based on correlation analysis which leverage the correlations between the execution time and run-time state transactions of GPU microarchitectures such as caches and shared memory to recover the last round key. The last round key leaked by the side-channel attack then can be reversed to obtain the original AES key. During the last round encryption of the T-Table based AES algorithm, table lookup actions which complete the data transformation result in different memory access patterns decided by the last round key leading to different execution cycles. The adversary bridges the execution time and the last round key with the differences in memory access addresses to leak useful information for AES key recovery. [63] presents a side-channel timing attack depending on the shared memory bank conflicts. According to the SIMT execution model, when loading data from shared memory, 32 threads in a warp generate 32 table lookups pointing to different shared memory addresses locating in different banks. Consequently, the shared memory bank conflicts happen when accesses going

to the same bank to affect the execution time. In the AES algorithm, the shared memory address accessed by each thread depends on the last round key resulting in diversity on the execution cycles of a load operation. The overall execution time of the AES program also depends on the shared memory bank conflicts due to the T-table accessing. This correlation of total execution time and the last round key is the critical measurement to guess the original key. In [66], instead of shared memory bank conflicts, they propose to use the number of unique memory requests after coalescing to build the correlation, while [64] demonstrate a similar attack which focus on the bank conflicts of GPU cache. Luo et al. [67] propose a power side-channel attack on GPU to crack AES key. They implement a power estimation model to trace the power consumption and remove the source of noises due to the asynchronous execution between warps and SMs. The power traces then can be used to analyze the correlation between the power consumption and the AES key bytes. However, all these methods require much more samples and time than the profiling-based SCA.

6.3 Profiling-Based Side-Channel Attack

6.3.1 The Original pSCA

First, we find that due to no MixColumn operation, the last round can be simply performed by one table lookup (i.e., load instructions), XORing the round key (i.e., logic instructions) and result write-back (i.e., store instructions). Among these instructions, we only concern about load and store instructions, as they are relevant to information leaking. In particular, the execution order and data dependencies of load and store instructions provide the basis for our profiling based attack. To generate and preserve one cipher-text byte, one load and one store instruction are required to read the table and write back the byte generated. To generate the entire

16 bytes block cipher-text, 16 load and 16 store instructions are executed alternately and hence the store instruction is dependent on the previous load instruction because it cannot be executed to store the cipher-text byte which has not yet been generated by executing the load instruction preceding it.

Second, the memory request coalescing which is considered as an effective method for GPU to reduce the number of memory accesses and enhance the throughput can be a channel to leak key related information to attackers. In the GPU-based AES, a warp with 32 threads can process 32 16-byte blocks simultaneously and a table lookup initiating a memory load instruction generates 32 memory requests to fetch the data from different addresses within the memory space where the table is stored. The size of the table in the last round is 1024 bytes with 256 4-byte elements. A cache line of the L1 cache in the GPU is typically 64 bytes and therefore 32 memory requests generated by one load instruction can result in memory accesses varying from 1 (if all 32 threads request data locating in the same cache line) up to 16 (i.e., 16 consecutive cache lines totally are necessary to serve the entire table) after coalescing. The number of unique memory requests of one load instruction of 32 threads in a warp is highly dependent on the table indexes which can be calculated with cipher-text byte and corresponding round key using an inverse lookup table.

Third, we find that there are in total 66 memory store transactions when 32 threads in a warp process 32 16-byte blocks. Specifically, 2 of 66 memory store transactions are caused by system calls at the end of the program. The remaining 64 transactions are used to store 32*16 bytes data for 32 threads and storing 32*1 bytes costs 4 transactions. Therefore, the number of unique memory store request that can be profiled ranges from 0 to 66.

We use the CUDA Profiling Tools Interface (CUPTI) to implement our own profiling tool. The Event API of CUPIT allows to access multiple performance counters

at the same time and therefore at one profiling point we can get two parameters (the number of the memory load requests and the number of the memory store requests) forming a pair. Due to the poor profiling rate, only one profiling point can be found for a single run of the kernel. The profiling point may happen anywhere and offers both valid and invalid information. To increase the profiling resolution, we keep sending the same plain-text to the victim program so that it is able to profile the same case under the same circumstance multiple times until the details to generate a sample are all captured.

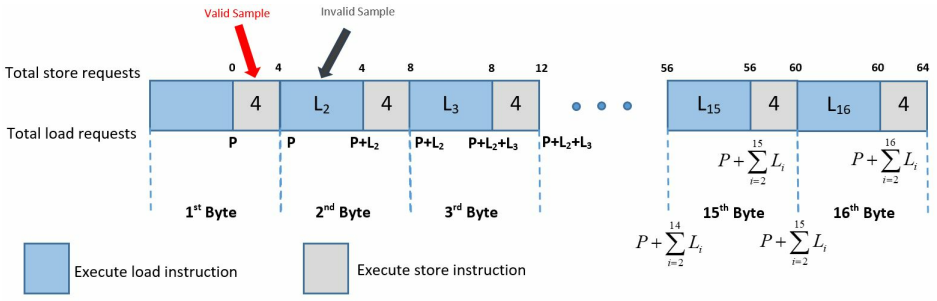


Fig. 42. The overview of the profiling strategy.

Figure 42 describes the scenario to determine the validity of a profiling point. P denotes the total number of unique memory load requests before generating the second cipher-text byte in the last round. $\{L_2, L_3, ..., L_{16}\}$ denotes a sample composed of the numbers of unique load requests for $2nd$ to $16th$ load instructions in the last encryption round. The profiling point that happens during the execution of the store instruction (i.e., the gray regions) is valid, providing the total number of unique memory load requests of previous load instructions. In the gray regions, the number of memory store requests may change due to the ongoing store instruction. However, the number of unique memory load requests remain constant because there is no load instruction executing simultaneously. This is because the load instruction before the store instruction has already been finished and the next load instruction

should not start before the completion of the current store instruction. For example, profiling in the first gray region pointed by the red arrow in Figure 42 can get a performance counter pair (3200, 2), in which 3200 is the total number of unique memory load requests generated by the load instructions in previous encryption rounds and the first load instruction in the last round whereas 2 is the number of unique memory store requests initiated by the first store instruction. As aforementioned, each store instruction costs 4 transactions and therefore, if only 1, 2 or 3 of 4 have been completed, the store instruction is still under execution. In this case, P equals to 3200. To summarize, for a performance counter pair (num_load, num_store), if $num\_store = a*4+b$ and $b = 1, 2$ *or* 3, $P+\sum_{i=2}^{a+1} L_i = num\_load$. By comparison, the profiling point locating in the blue region is invalid since a load instruction is during execution. We then can filter the performance counter pair (num_load, num_store), if $num\_store = 4 * a$. For two invalid performance counter pair, even they have the same num_store, num_load may vary according to the profiling timing during the execution of the same load instruction. By collecting all 16 valid numbers at the right profiling points (i.e., $\{P, P + L_2, P + L_2 + L_3, ..., P + \sum_{i=2}^{15} L_i, P + \sum_{i=2}^{16} L_i\}$), the numbers of unique memory load requests for all 16 load instructions except for the first one then can be recovered by calculating the differences between consecutive numbers in $\{P, P + L_2, P + L_2 + L_3, ..., P + \sum_{i=2}^{15} L_i, P + \sum_{i=2}^{16} L_i\}$. For example, $L_{16} = (P + \sum_{i=2}^{16} L_i) - (P + \sum_{i=2}^{15} L_i)$. However, the number of unique memory load requests for the first cipher-text byte cannot be calculated directly based on profiling because there is no store instruction preceding it as the boundary.

Figure 43 depicts the procedure to recover the entire 16-byte AES key. The exact number of unique memory load requests in the last round encryption for generating every cipher-text byte except the first byte then is already known by the attacker, based on which the corresponding AES key bytes can be recovered one by one. To

recover one key byte, using an inverse lookup table, the attacker checks all 256 guesses of a single byte of the AES key by calculating and comparing to the profiled number. The number of possibilities of the corresponding key byte can be narrowed down. The number of possibilities can be further decreased to 1 case by repeating the same procedure with different input data for several times, which must be the true key byte. The profiling-based method can only recover the last 15 bytes of the AES key. However, the attack can reveal the first key byte by checking at most 256 cases with the same input if the later 15 key bytes are already known from profiling.

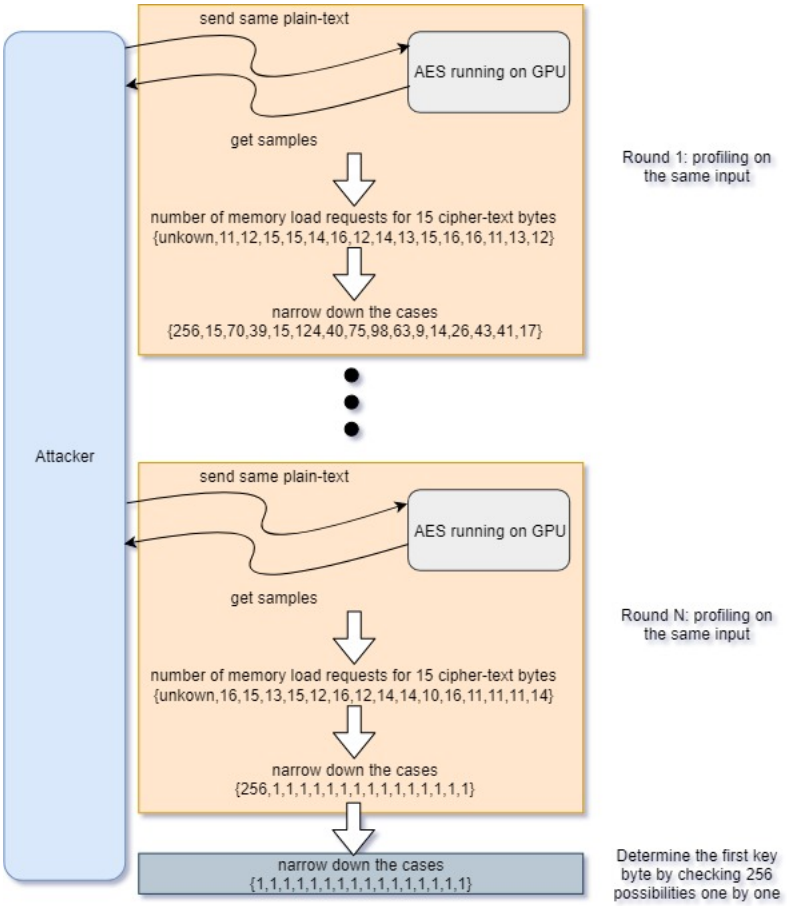


Fig. 43. Procedures to recover the entire 16-byte AES key

6.3.2 The Boundary Check pSCA

The RCoal technique cannot only thwart prior GPU based SCA attacks [62, 63, 64, 65, 66, 67], but also the original pSCA. This is because with RCoal, the exact number of loads is hard to be derived due to the memory coalescing randomization, making it impossible for the orginal pSCA to recover the AES key. To enhance the pSCA (and hence to motivate the study of stronger protection mechanisms than RCoal), we propose to determine the upper bound and lower bound of the number of unique memory accesses to form a range that the actual number should locate in. The range can be calculated by a possible value of a key byte and the encrypted output. For one key byte, there are 256 cases and thus 256 ranges can be computed. Moreover, by checking whether the profiled number is within the ranges or not, the wrong guesses of this key byte can be eliminated. Since the range calculation is related to the number of subwarps that is still unknown, we need to examine all the cases including 2, 4, 8, and 16. It is unnecessary to check 1 and 32 because 1 subwarp is the same with the normal problem without RCoal and 32 subwarps means that the coalescing mechanism is completely disabled, thus the number of unique memory requests is always 32.

Assuming the number of subwarps is 4 (we can use the same method to calculate the range for other numbers of subwarps such as 2, or 8, but not 16), if the number of memory requests accessing the same cache line is equal to or less than 4, these requests can be allocated to different subwarps to maximize the total number of memory requests. If the number of memory requests accessing the same cache line is larger than 4, at least one memory request has to be assigned to a subwarp containing the memory request that can be merged together. Furthermore, more than 4 memory requests accessing the same cache line can only result in 4 unique memory requests

always. With this strategy, we can calculate the upper bound.

To calculate the lower bound, the idea is to allocate as many memory requests as possible, which can be coalesced into the same subwarp. The range derived from a real key can guarantee to cover the profiled number, while the profile number may lie out of bounds calculated with a false guess, which then is eliminated. Indeed, with a single sample, a false guess may succeed to form a range holding the profiled number of memory requests and remains to be considered a candidate of the real key; however, it cannot guarantee this all the time for every sample. Therefore, with enough samples and corresponding boundary checks, a false guess will be identified and then filtered sooner or later. There are 256 guesses for each key byte and 256*16 guesses in total for the 16-byte AES key. The boundary check pSCA is responsible for reducing 256*16 guesses to only 256 cases and the brute-force method exhausts them to a single case, which is the correct AES key. Based on the experimental results, the boundary check pSCA is capable to successfully exclude all other guesses and result in 256 possibilities in the case of sufficient samples. In addition, according to the results, the profiled number is never smaller than the lower bound, therefore the calculation of lower bound can be skipped for simplification.

The boundary check pSCA follows two assumptions to calculate the upper bound with a fixed number of subwarps (e.g. 2, 4 or 8): 1) the subwarp size is flexible in order to hold as many as possible memory requests that cannot be coalesced to approach the worst memory coalescing case; 2) a thread is free to be allocated to any subwarp in order to avoid memory requests that can be merged together to achieve the worst memory coalescing case. The flexibility on the subwarp size and thread allocation leads to an ideal case that can be achieved to increase the number of unique memory requests and hence weaken the memory coalescing mechanism. The RCoal techniques including FSS, RSS, FSS+RTS and RSS+RTS, involve different kinds

of memory coalescing randomness and increase various numbers of unique memory requests. However, none of them is possible to generate the number of unique memory requests exceeding the ideal case calculated with the corresponding key byte. In other word, if any RCoal technique results in a number that is larger than the ideal case, this guess of the corresponding key byte used to compute the ideal case is not the real answer. Therefore, the boundary check mechanism is effective for all RCoal techniques and hence it is unnecessary for the boundary check pSCA to know which one is exactly chosen to protect GPUs.

6.4 Profiling-Assisted Correlation-based Side-Channel Attack

The GPU architectural vulnerability to side-channel attacks. The SIMT execution style and the memory coalescing mechanism contribute to the performance enhancement, however, negatively cause the vulnerability to side-channel attacks. Following the SIMT execution manner, concurrent threads are able to execute an unique instruction in parallel and to access different memory addresses. In order to conserve the memory bandwidth, a coalescing unit is equipped to merge memory requests accessing to the same memory addresses. The SIMT and memory coalescing are exploited by malicious attackers to threaten the security. In the AES algorithm, the number of coalesced memory accesses initiated by a memory load instruction is highly related to the secure key. The attacker then can extract the confidential information from the variation of the number of coalesced memory accesses.

The correlation analysis. Although the variation of the number of coalesced memory accesses implies the secure key, the attacker cannot observe it directly. Instead, the attacker successes to perceive this variation through a correlation-based side-channel, which exposes it through the variation of the observable metric, i.e., the total number of memory accesses in this work. To be specific, the number of

coalesced accesses of a single load instruction makes contribution to the total number of memory accesses of the entire program. Therefore the attacker can detect the variation statistically by a correlation analysis that constructs the relationship between the number of memory accesses from a load instruction and the whole program.

Profiling tool assistance. The profiling tool we implemented in this work can accurately capture the total number of memory accesses which is required to conduct the correlation analysis. The CUDA Profiling Tools Interface (CUPTI) provides various APIs to build the profiling tool. Since the detailed number of coalesced memory accesses for each load instruction is unnecessary for the pacSCA, the profiling tool don't have to struggle for a high sample resolution. Instead, our profiling tool can easily provide the total number of memory accesses accurately by taking a sample right after the cypher-text is returned. To get enough traces, we make the AES program contiguously working on different plain-texts.

Figure 44 shows the flow chart for the Profiling-Assisted Correlation-based Side-Channel Attack. The size of the original and 10 round keys are fixed to 16 bytes in the 128-bit AES algorithm. For one possible value (e.g., 128) of a key byte (e.g., key byte 0), a correlation coefficient b can be computed via the regression analysis model as $y = b * x + c$, where y is the profiled number of unique memory load requests for the entire AES program, x is the guessed number of unique memory load requests in the last round for the key byte 0 with guessed value 128, and c is a constant. While the profiled number of unique memory load requests for the entire AES program is obtained by the profiling tool, the corresponding guessed number of unique memory load requests for key byte 0 in the last round can be computed with the output (corresponding byte in the cipher-text, 0th byte in this case), inverse table (a constant table known by the attacker), and the 128 (one possible case of key byte 0). Since there are in total 256 cases (from 0 to 255), 256 correlation coefficients will

be calculated. By collecting enough number of samples, the real key byte value will stand out other 255 cases by showing the largest correlation coefficients. Therefore, the pacSCA is capable of recovering the 16-byte AES key byte by byte independently in the same way.

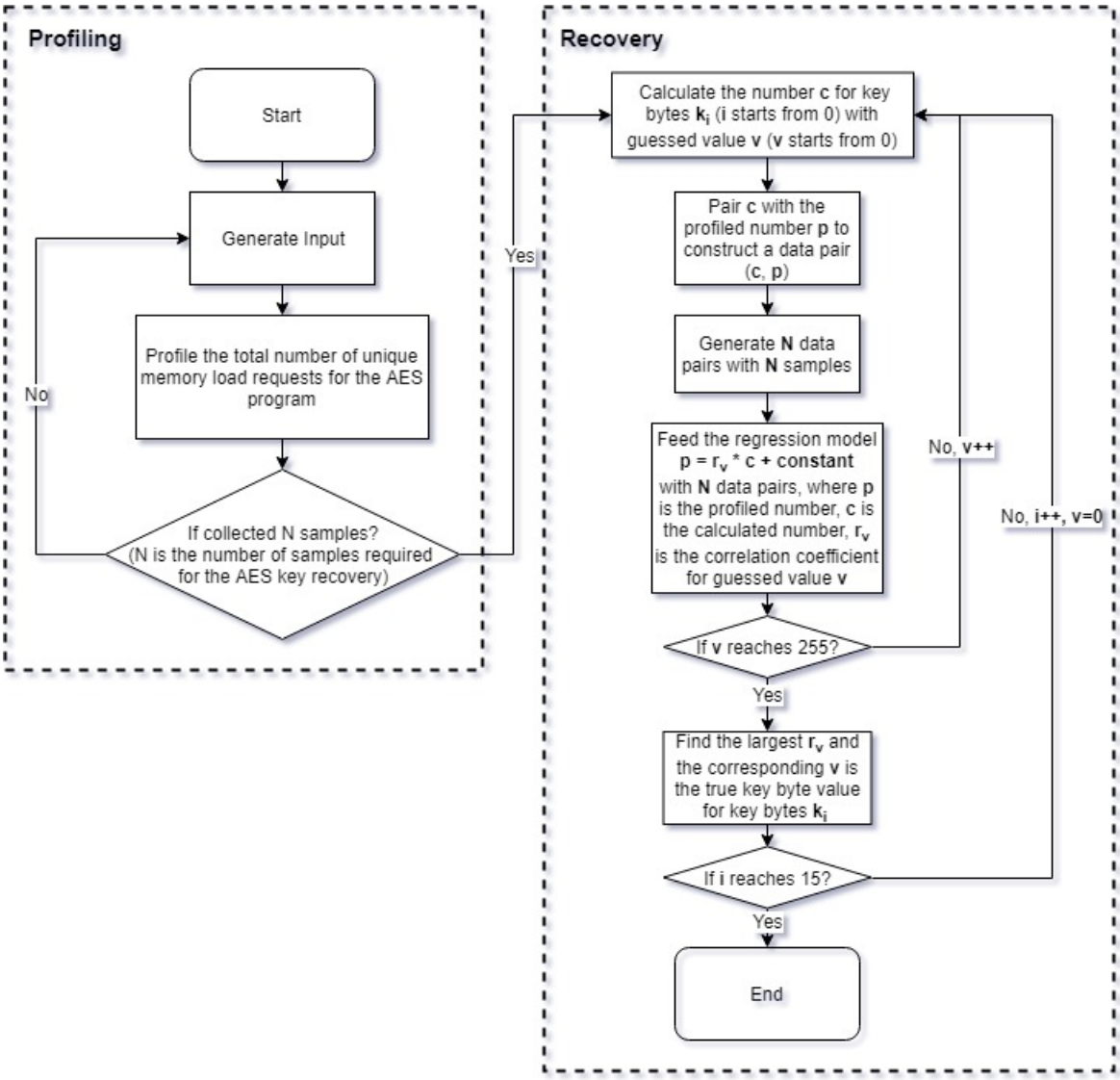


Fig. 44. The flow chart for pacSCA.

Table 3. Experimental results of pSCA

	Tesla C2075	Quadro 2000
Key recovery done	success:100	success:100
Recovery success rate	100%	100%
Profiling done	success:676	success:656, fail:1
Profiling accuracy	100%	99.85%
The average profiling time	13814.99 ms	29837.02 ms
The average recovery time	93389.37 ms	196029.20 ms
The average inputs profiled	6.76	6.57

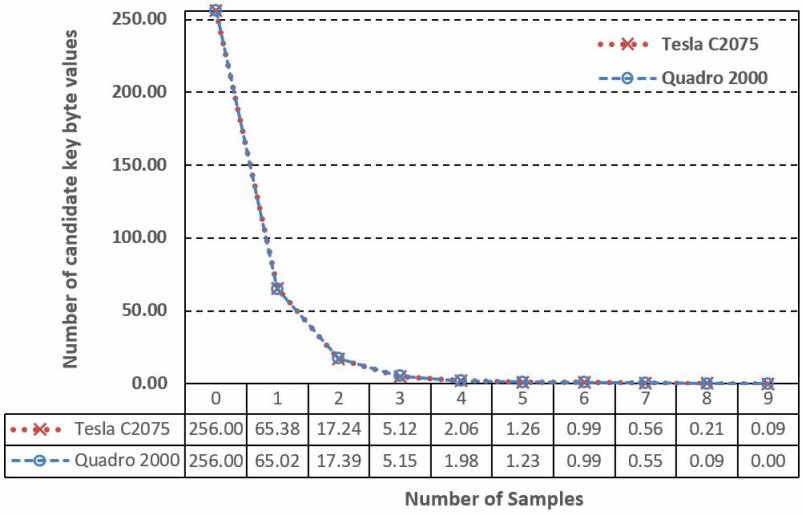
6.5 Evaluation Results of pSCA

6.5.1 Evaluation Results of Original pSCA

We implement a python script to control and complete the AES key cracking procedure automatically and repeatedly which can also monitor the procedure and finally output the results. Using this script, we have done systematic experiments and verification. To simulate a attack as real as possible, we let the AES program keep running and waiting data from outside. The adversary program keep sending data to AES program for encryption and meanwhile launches the GPU profiling tool to get samples. We have evaluated the profiling-based AES key recover on two GPU platforms: NVIDIA Quadro 2000 and NVIDIA Tesla C2075.

To test the accuracy of the profiling tool and success rate of key recovering, we recover the key 100 with different inputs. Table 3 shows the experimental results for both GPU cards. Both recovery success rate and profiling accuracy are 100% for NVIDIA Tesla C2075 while NVIDIA Quadro 2000 fails to get the correct number of memory load requests only once. However, the success on the rest of 656 inputs profiled results in 99.85% profiling accuracy. Although the profiling accuracy is not 100%, the recovery success rate can still achieve 100% since the inaccurate samples can always be filtered by the profiling tool noticing the abnormal pattern. The rea-

son for the profiling failure on NVIDIA Quadro 2000 is that this GPU card serves for both general-purpose and graphics computing. Consequently, the profiling tool may be misled by graphic computation tasks. This profiling failure can be easily avoided by prohibiting other tasks running on GPU when initiating the profiling based side-channel attack. NVIDIA Tesla C2075, which is dedicated to general-purpose computing, can always achieve 100% on both recovery success rate and profiling accuracy.



	0	1	2	3	4	5	6	7	8	9
Tesla C2075	256.00	65.38	17.24	5.12	2.06	1.26	0.99	0.56	0.21	0.09
Quadro 2000	256.00	65.02	17.39	5.15	1.98	1.23	0.99	0.55	0.09	0.00

Fig. 45. The convergence speed of the number of possible key byte cases with the increasing number of samples.

Figure 45 shows the convergence speed of the number of possible key byte cases with respect to the increasing number of samples. Both GPU cards present the same convergence trend that the number of cases drops from 256 to around 65 after profiling one input and the convergence speed slows down afterward. Figure 46 presents the distribution of the number of samples required to recover the entire AES key for 100 times. In most cases, information retrieved from 6 or 7 samples is enough to recover all the key bytes on both GPU cards. On Tesla C2075, the largest number of samples is 9 for a complete key recovery, while it is 8 on Quadro 2000. The average number of samples is 6.76 and 6.57 for Tesla C2075 and Quadro 2000 respectively, which is

much smaller than the amounts needed by existing GPU side-channel attacks [66, 64, 63, 67].

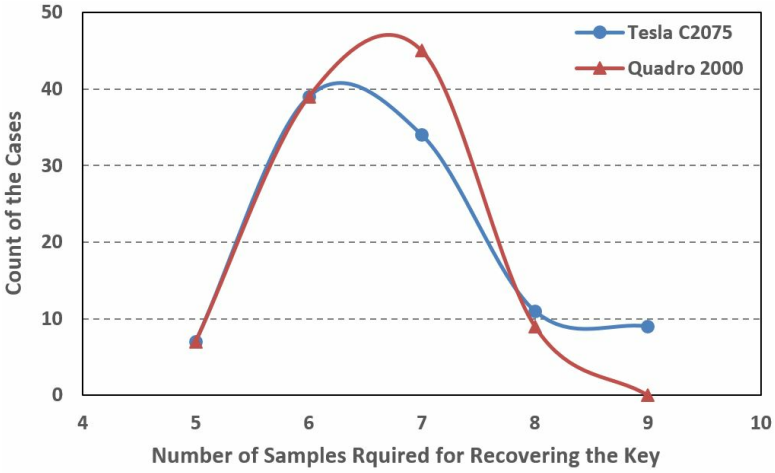


Fig. 46. The distribution of the number of samples required to recover an entire AES key.

We also evaluate the performance of the proposed attack as the length of the AES key is increased to 192-bit and 256-bit as shown in Table 4. The extended AES key does not increase the number of samples for the key recovery; however, the time to profile one sample increases a little bit since it takes longer time to execute the encryption kernel due to the additional encryption rounds. The scalability of the profiling-based AES key recovery method is very good as the profiling time appear to increase linearly with respect to extending the length of the AES key.

6.5.2 Evaluation Results of Boundary Check pSCA

The Boundary Analysis. We first record the boundaries calculated during the AES key recovery. Figure 47 shows the average boundary as well as the profiled number of memory loads for four RCoal techniques (FSS, RSS, FSS+RTS and RSS+RTS) with respect to the number of subwarps. FSS and FSS+RTS result in a similar num-

Table 4. Scalability with longer AES keys

	Profiling one sample	Average samples
Tesla 128-bit	13814.99 ms	6.76
Tesla 192-bit	15011.62 ms	6.68
Tesla 256-bit	16825.12 ms	6.71
Quadro 128-bit	29837.02 ms	6.57
Quadro 192-bit	32521.45 ms	6.6
Quadro 256-bit	34651.12 ms	6.57

ber of memory loads on average, and so do RSS and RSS+RTS. Moreover, Figure 47 also depicts that the true number is closer to the boundary for FSS and FSS+RTS than for RSS and RSS+RTS, implying that the false cases in FSS and FSS+RTS are more likely to cross the boundary and thus are easier to identify. Therefore, FSS and FSS+RTS may need a less number of samples to recover the AES key than the RSS and RSS+RTS. In addition, with a larger number of subwarps, the upper bound goes up sharply, implying that it becomes harder to reach the boundary.

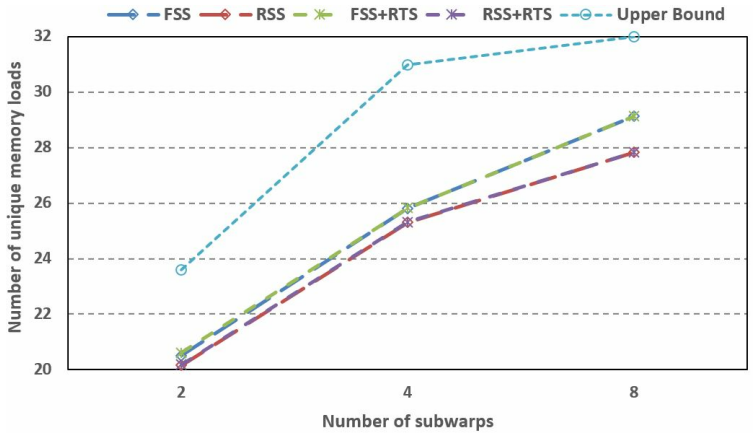
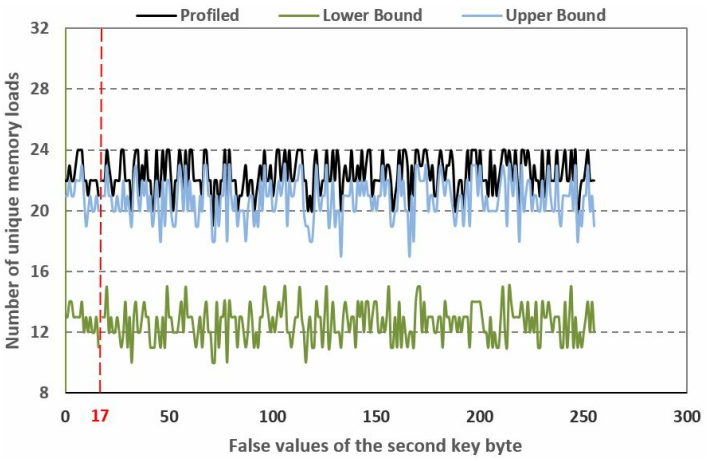


Fig. 47. The average boundary and the profiled number of memory loads for four RCoal techniques.

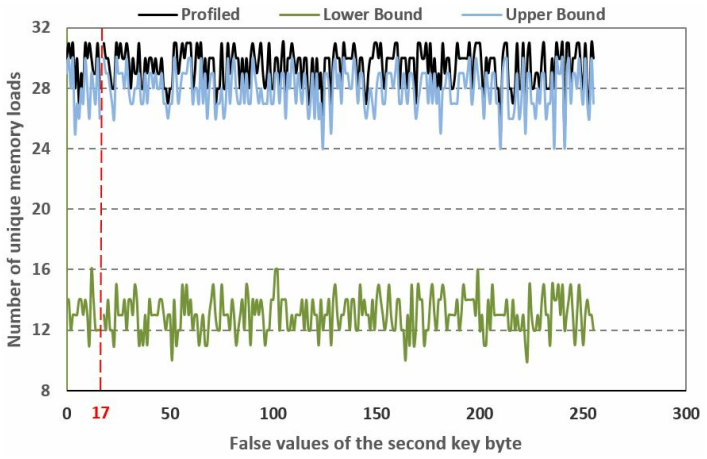
In order to confirm if all 255 false guesses for a key byte have been identified and removed during rebuilding the AES key, we record boundaries coming from false guesses and the profiled number of memory load requests when false guesses fail the

boundary check. Whenever a guess fails the boundary check, it is then excluded and will not be checked again. Therefore, the failure of the boundary check only happens once for each false guess. As shown in Figure 48 (where the X-axle represents the possible values of the second key byte except for the true value, e.g. 17 in this case. The black line draws the profiled number of memory load requests and the blue and green line record the upper bound and lower bound computed with the corresponding wrong guesses.), the black line is always above the blue line for RSS+RTS with 2, 4, and 8 subwarps, indicating that all 255 wrong guesses that result in upper bounds even smaller than the real number of memory load requests fail the boundary check. The real key byte value highlighted with the red dash line in Figure 48 is void in this record of the boundary check failure, indicating that it survives every boundary check and stands out of other wrong cases. Both the upper bound and the profiled number keep approaching 32 when the number of subwarps increases from 2 to 8, which again demonstrates that the wrong cases are harder to be eliminated through boundary check with a large number of subwarps and hence the security is enhanced accordingly (though not being able to thrwart our attack). As aforementioned and also illustrated in Figure 48, the profiled number of memory loads has never exceeded the lower bound, which therefore is unnecessary to be considered in the boundary check mechanism.

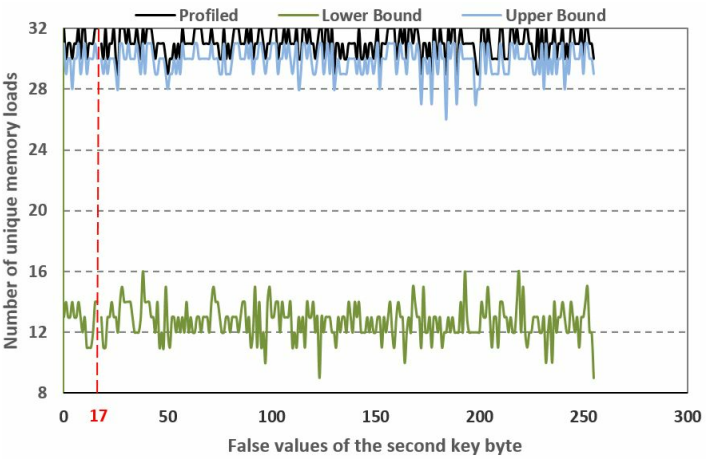
The Boundary Check Mechanism . We evaluate the boundary check mechanism and Figure 49 shows the number of samples needed to recover the AES key when RCoal is enabled. RSS (along with RTS) can offer the best security as compared to FSS and FSS+RTS, according to the number of samples needed for the AES key recovery. Indeed, the RCoal techniques can defend against the original pSCA and the number of samples increases exponentially with respect as the number of subwarps increases from 2 to 8. However, through collecting more samples, the AES key can



(a) RSS+RTS with 2 subwarps



(b) RSS+RTS with 4 subwarps



(c) RSS+RTS with 8 subwarps

Fig. 48. RSS+RTS eliminates the false guesses of the second key byte through boundary check.

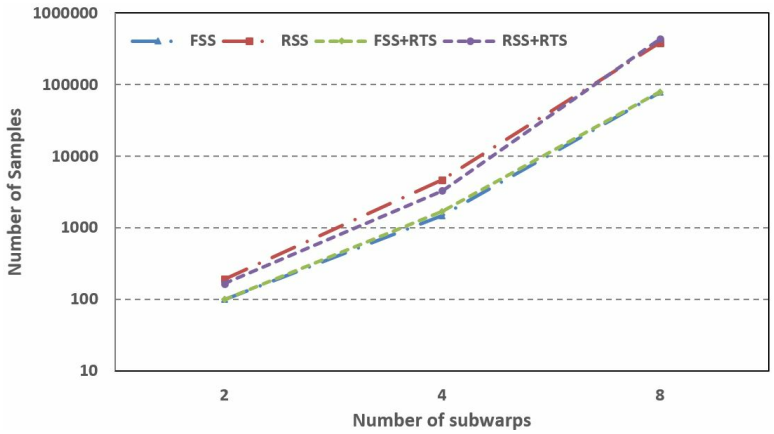


Fig. 49. The number of samples needed to recover the AES key when RCoal is enabled.

still be recovered by the boundary check pSCA (e.g., For RSS+RTS with 8 subwarps, the AES key can be successfully rebuilt using around 430000 samples). The drawback of the RCoal is that the performance overhead can be unaffordable. For 16 subwarps, the upper bound can be less than 32 if more than half of the access pointing to the same cache line and then it is still possible that the profiled number is large that the upper bound to filter the corresponding false guesses. Hence, theoretically, the boundary check pSCA still works for 16 subwarps in case of large enough sample size. However, based on the evaluation of 16 subwarps with up to 1000000 samples, the upper bound is always 32 for every guess of a key byte , showing that the condition that more than 16 of 32 accesses request the same cache line is extremely rare. The profiled number is then impossible to fall out of upper bound to identify false guesses. Therefore, we aggressively assume that Rcoal with 16 subwarps which offers the best security can completely protect the GPU from the pSCA; however, it will result in huge performance overhead (26% for plain-text with 32 lines and over 200% for plain-text with 1024 lines as reported in [68]).

Based on the experimental results, the most secure RCoal method RSS, when used with 2, 4, or 8 subwarps, can be cracked by the boundary check pSCA by col-

lecting enough samples. To be more specific, our experiments show that around 150, 3000, and 430000 samples are needed for RSS with 2, 4, and 8 subwarps respectively. The typical time spending on collecting one sample is 5 seconds and thus the boundary check pSCA can recover the 16-byte AES key with around 12 minutes, 4 hours, and 25 days when the AES algorithm is running on a RSS-enabled GPU with 2, 4, and 8 subwarps respectively. Compared to the brute-force method which may take 1 billion years to crack the AES key even with a supercomputer [70], the boundary check pSCA manages to significantly reduce the effort.

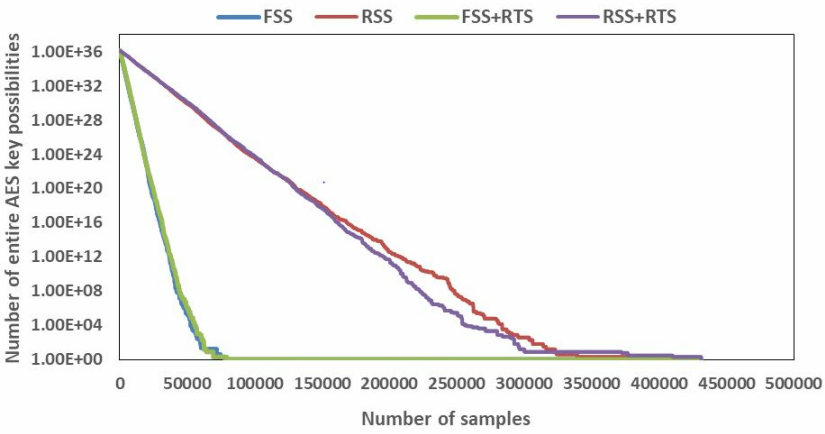


Fig. 50. The convergence of the number of entire AES key possibilities.

We further propose a two-stage recovery method to achieve even shorter recovery duration. The boundary check pSCA can reveal the last 15 bytes of the AES key and in other words, the number of possible cases starting from 2^{8*15} can be reduced to 1. Instead of examining every possibility in a brute-force way, the boundary check pSCA works on revealing each key byte value independently in parallel and therefore is much more effective on excluding wrong cases. However, as shown in Figure 50, for RSS, the number of possibilities is exponentially reduced using the first 320000 samples, while the rest 110000 samples only eliminate a quite limited number of wrong guesses.

By observing this, we divide the entire key recovery procedure into two stages. **Stage 1** applies the boundary check pSCA to exclude most of the false guesses effectively and **Stage 2** uses a brute-force method to finally reach the correct AES key. The switching point between these two stages is determined by comparing the effectiveness on excluding wrong cases of two methods (the boundary check pSCA to brute-force). The effectiveness can be quantitatively measured for the boundary check pSCA, which is represented as the number of possibilities excluded per second (PEPS). The PEPS of the boundary check pSCA is calculated every 10000 samples using the formula: $num\_excluded/(10000 * sample\_time)$, where $num\_excluded$ denotes the number of entire AES key possibilities excluded by the last 10000 samples and $sample\_time$ represents the time to get one sample. On the other hand, verifying one case costs 5 milliseconds and PEPS is 200 for the brute-force method. Whenever PEPS of the brute-force method is larger than the boundary check pSCA, the key recovery procedure enters **Stage 2**. According to the results, the time spending on recovering the AES can be further reduced from about 25 days to less than 14 days by the two-stage strategy.

6.6 Evaluation Results of pacSCA

The pacSCA is evaluated on two GPU platforms: NVIDIA Quadro 2000 and NVIDIA Tesla C2075.

According to the correlation results shown in Figure 51, the correct values of all 16 key bytes (highlighted with the red circles) stand out of other 255 possibilities. To determine the accuracy of recovering the AES key, we collect the success rates with respect to different amounts of samples. A success rate is calculated through averaging over 100 trials with certain number of samples. Figure 52 shows the success rates of recovering the $0th$ key byte, when the AES program is running at $-O3$ and

$-O0$ optimization level respectively. For both optimization levels, the success rate converges to 100% at around 10000 samples. By comparison, the SCA proposed in [66] can only achieve 100% success rate till the number of samples raises up to 700000. As the high optimization level reorganizes the CUDA instructions to avoid data dependency stalls and hide memory access latency, the relationship between the execution time and the number of unique memory accesses is obscured. Consequently, the performance of the SCA in [66] that leverages this relationship to constructs the side-channel, is dramatically affected by compiler optimization levels. On the other hand, since different optimization levels result in the same number of memory accesses as long as the AES program is working on the same plain-text, the pacSCA delivers similar performance in spite of the optimization levels as shown in Figure 52.

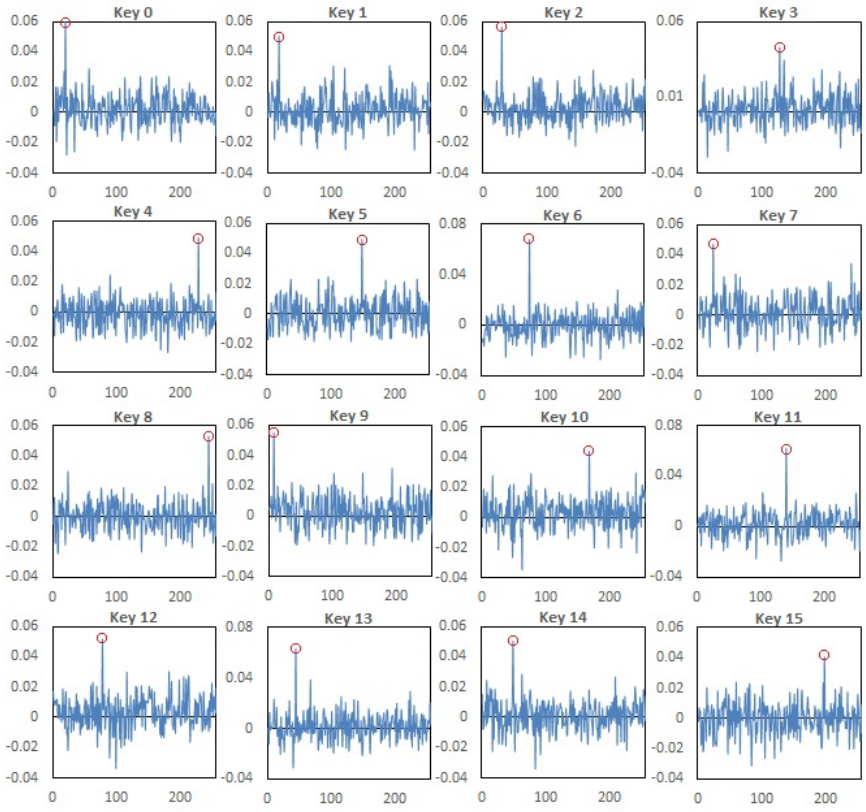


Fig. 51. Correlation analysis result of 10000 samples.

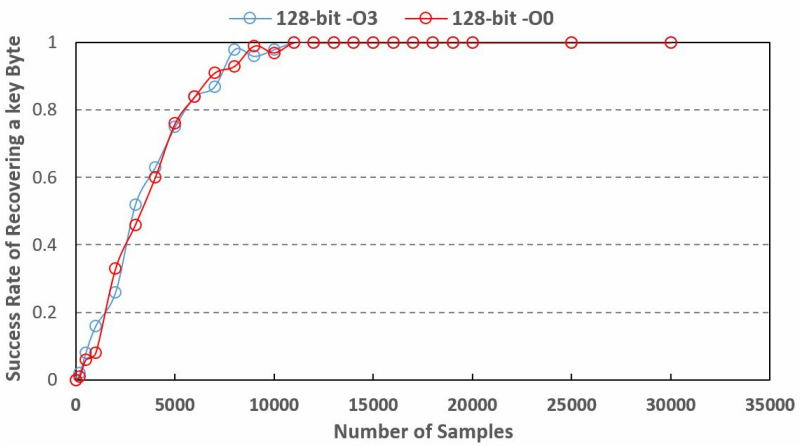


Fig. 52. Success rate of 0^{th} key byte.

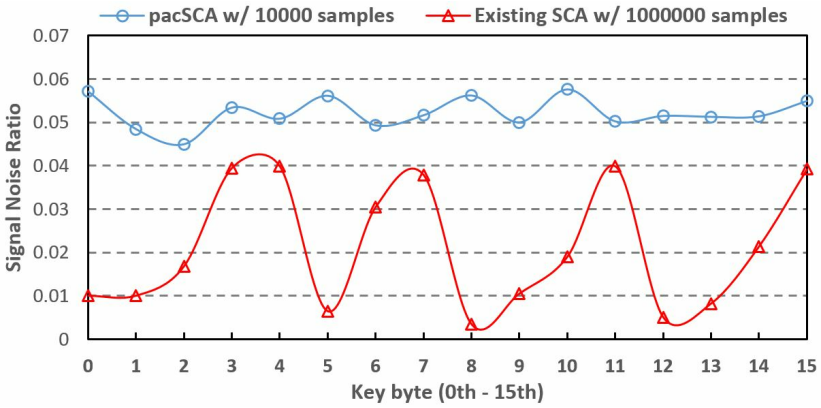


Fig. 53. The Signal to Noise Ratio (SNR) for the pacSCA and the existing SCA in [66].

As the success rate directly depends on the Signal to Noise (SNR) value and a linear relationship exists between the SNR and the correlation, the efforts paid on recovering a certain key byte can be indicated by the SNR value. As shown in Figure 53, with only 10000 samples, SNR values converge to 0.05 for all 16 key bytes in the pacSCA. Moreover, only a slight variation are found among SNR values of different key bytes. In contrast, the existing SCA [66] still results in much smaller SNR values with huge variation spreading from 0.0064 to 0.0399 with up to 1000000 samples. This

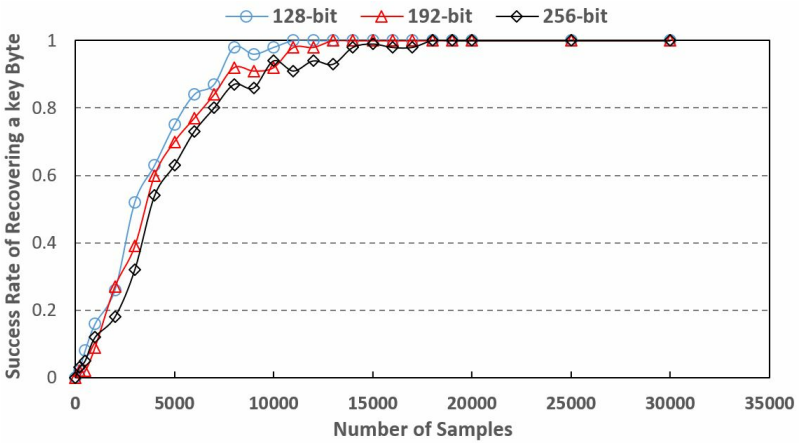


Fig. 54. Success rate of 0^{th} key byte for 128-bit, 192-bit and 256-bit AES key.

result illustrates that the pacSCA needs a much smaller number of samples to reveal the secure key than the existing SCAs as shown in Table 5. Due to the slight variation of the SNR values, the pacSCA needs similar number of samples to recover each key byte. The existing SCA, on the other hand, requires much larger number of samples for some of the key bytes, while much smaller number for others. The downside is that the overall number of samples depends on the worse case and a part of samples are redundant for other cases. Most of existing SCAs utilize the execution time as a measurable metric to observe the variation of the number of memory accesses. Since the execution time is determined by comprehensive hardware and software factors, the performance of these SCAs can be negatively affected by noises introduced by various factors such as the optimization level, cache miss rates, the number of cache bank conflicts and others that could vary the execution time. By comparison, the pacSCA observes the variation of the total number of the memory accesses rather than that of the execution time and thus it is immune to the noises and factors mentioned above. Naturally, the pacSCA outperforms the existing SCAs. Enough samples (10000 for the pacSCA) can be collected by the profiling tool in 5 seconds.

The correlation analysis takes less than 1 second. Overall, the entire procedure of revealing the AES key costs as short as 6 seconds.

Table 5. Comparison of existing GPU side-channel attacks

SCA	Type	Correlation	Number of Samples
pacSCA	Profiling	the total number of unique memory requests relies on the number of unique memory requests in the last round	Only **10000** samples is required due to the strong relationship.
pSCA	Profiling	No correlation analysis is required by this profiling-based SCA	**Less than 9** samples is required to shrink down 256 possibilities to a single case for all 16 key bytes values.
[66]	Timing	Execution time relies on the number of unique memory requests after coalescing	The collected samples contain 32-block encrypted messages and associated timings. The number of unique memory requests [66], cache bank conflicts [64] and shared memory bank conflicts [63] which are associated with the execution time can then be computed with the encrypted data and corresponding round keys using an inverse lookup table. The correlation coefficient which represents the strength of this relationship is calculated by a correlation analysis model for all 256 possible cases of a single key byte. The value with the maximum correlation coefficient is the right key. The correlation analysis model needs a huge amount of the samples in order to overcome noises and derive the right key with the largest correlation coefficient. The necessary amount of the samples to overcome noises varies for different timing SCAs due to their differences on Signal to Noise Ratios (SNRs). In particular, SCAs in [66], [64] and [63] need 10^6, 10^5 and 10^7 samples, respectively.
[64]	Timing	Execution time relies on cache access conflicts decided by the accessed address of each thread	
[63]	Timing	Execution time relies on shared memory conflicts decided by the accessed address of each thread	
[67]	Power	Power consumption depends on the Hamming distance which is calculated by the power model with the corresponding key value	With 10,000 power consumption traces (each contains the encryption of 49,152 blocks of plain-text), the correct key byte value stands out with the maximum correlation coefficient [67]. To extract the whole last round key, the same procedure needs to be repeated 16 times and **160,000** traces are necessary in total.

We also evaluate the performance of the pacSCA on revealing 192-bit and 256-bit AES key. As shown in Figure 54, the success rates approach 100% with only 13000 and 17000 samples for 192-bit and 256-bit key, respectively. The slight increase in the number of samples shows the good scalability of the pasSCA.

We also evaluate the pacSCA when the Rcoal is enabled. Figure 55 shows the number of samples needed to recover the 128-bit AES key when RCoal is enabled. Indeed, the RCoal techniques can defend against the pacSCA and the number of samples increases exponentially with respect as the number of subwarps increases from 1 to 16. However, through collecting more samples, the AES key can still be revealed by

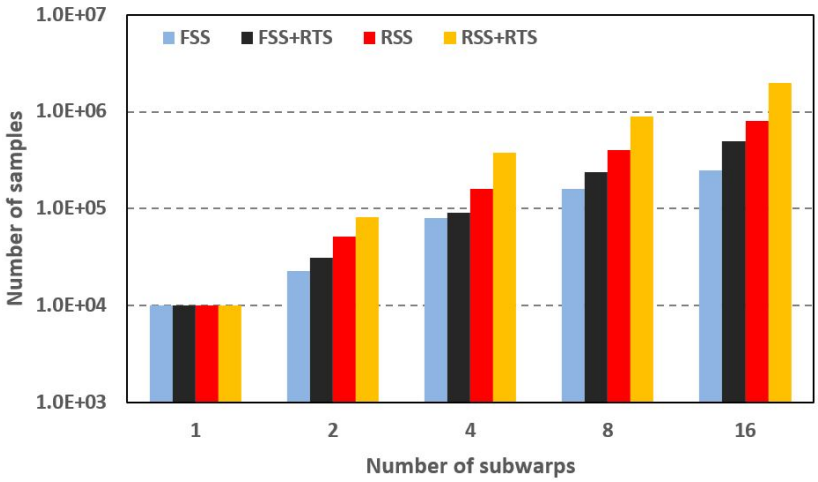


Fig. 55. The number of samples needed to reconstruct the AES key when RCoal is enabled.

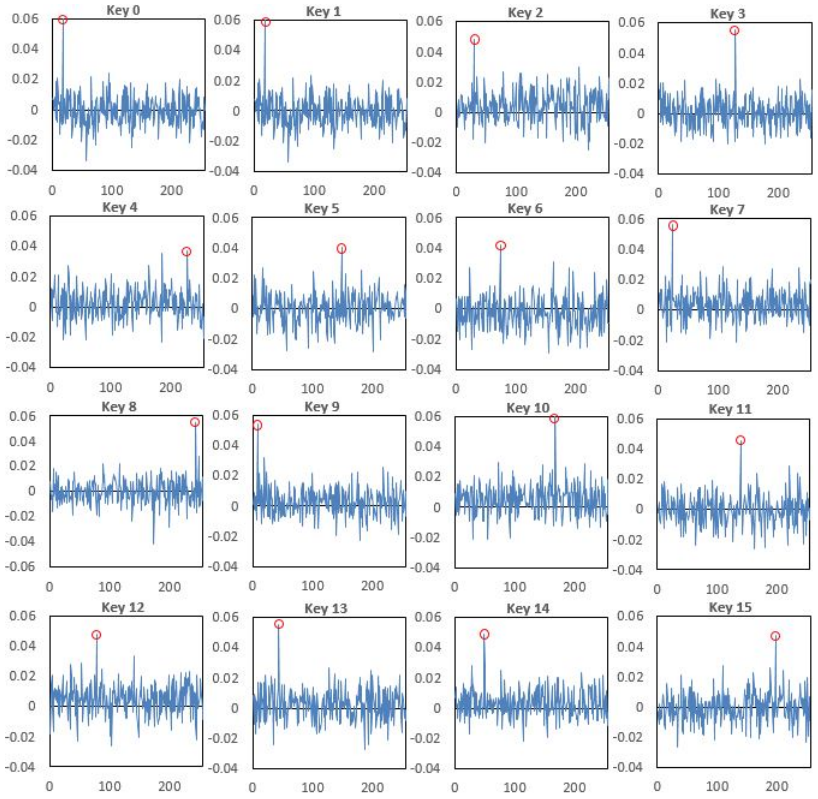


Fig. 56. Correlation analysis result of 2000000 samples for the pacSCA w/ RSS+RTS defense.

the pacSCA (e.g., For RSS+RTS with 16 subwarps, the AES key can be successfully rebuilt using around 2000000 samples as shown in Figure 56). It is unnecessary to check 32, since setting 32 subwarps is equivalent to disable the coalescing mechanism resulting in 32 unique memory requests always. Based on the experimental results, RSS+RTS can offer the best security as compared to FSS, FSS+RTS and RSS. To be more specific, our experiments show that around 82000, 380000, 900000 and 2000000 samples are needed for RSS+RTS with 2, 4, 8 and 16 subwarps respectively. As compared to the pacSCA without any defense mechanisms, RSS+RTS increases its effort on collecting samples by 8.2-, 38-, 90- and 200-times. As aforementioned, our profiling tool can collect 10000 samples every 5 seconds. The pacSCA is able to recovery the AES key in around 17 minutes (200×5 seconds$\div 60 \approx 17$ minutes) on the GPU protected by the most secure Rcoal defense mechnism (RSS+RTS with 16 subwarps). Our evaluation states that the GPU systems are still facing fatal side-channel threats, although the state-of-the-art countermeasures such as the Rcoal techniques are deployed.

6.7 Countermeasures

A straightforward way to completely hide the variation on the number of memory load requests and close the opportunity for the propsoed attack to thieve the secure key is to disable the memory coalescing mechanism, which however, may result in substantial performance degradation. Kadam et al. [68] propose the Rcoal techniques which can offer the security at the cost of the tuned performance overhead. However,

1. This technique is implemented by additional micro-architecture in GPUs leading to hardware overhead;

2. Moreover, since it is hardware-based, the randomized memory coalescing has to

be effective for all load instructions and may consequently result in significant performance overhead;

3. Furthermore, the randomized coalescing technique can only improve GPU security instead of completely eliminating the threat of side-channel attacks.

We have demonstrate that the Rcoal can only degrade the performance of boundary check pSCA and pacSCA. The GPUs are not completely protected by the Rcoal from our two SCAs. Our future work will focus on developing more effective countermeasures that introduce as low as possible hardware and performance overheads and as well are able to offer flexible deployment and firm security.

CHAPTER 7

CONCLUSIONS

This dissertation is dedicated to improving the performance, energy efficiency, and security of GPUs. We have introduced multiple strategies designed to enable GPUs to operate in an energy-efficient manner, while also offering opportunities for additional performance enhancements. Furthermore, with the intention of spotlighting the susceptibility of GPUs to side-channel attacks, we conducted research on AES algorithms deployed on GPUs and examined countermeasures. Altogether, this dissertation comprises four projects.

7.1 GPU Register File Narrow-Width Operands Packing

By running with massive thread-level parallelism(TLP), GPUs achieve high throughput as well as memory latency hiding. As a result, a large size of register file is required to enable fast and low cost context switching between tens of thousands of active threads. However, RF resources are still insufficient to enable all thread level parallelism and the lack of register file resources can hurt performance by limiting occupancy of GPU threads. Inspired by observing that a large percentage of computed results have fewer significant bits compared to the full width of a 32-bit register for many GPGPU applications. We propose OWAR, a GPU register packing scheme to dynamically exploit narrow-width operands and pack multiple operands into a single full width register. OWAR utilizes the additional registers saved by register packing to improve energy efficiency through power gating and thread overrun. The experimental results shows that OWAR can reduce the GPGPU's total energy up to 29.6%

and 9.5% on average. In addition, OWAR achieves performance improvement up to 1.97X and 1.18X on average.

7.2 GPU Register File Drowsy Management

We study three RF leakage energy management techniques for GPUs. To save as much leakage energy as possible, Drowsy-IS places a register into the drowsy mode immediately after an access. The drawback is that the additional cycles are required to re-active registers from the drowsy mode for each re-access, which may lead to performance degradation. To mitigate the re-activation penalty, we propose the Drowsy-TA technique to hold the register in the active mode for a fixed period of time before putting it into the drowsy mode. However, the leakage energy may be wasted if no re-access occurs during the awake interval. Finally, we propose Drowsy-RI, an adaptive policy to predict re-access interval and manage RF leakage accordingly at run-time.

Our experimental results show that compared to the baseline RF, Drowsy-IS achieves 91.7% RF leakage energy reduction on average at the cost of 4.4% performance degradation. Drowsy-TA reduces RF leakage energy by 82.8% with negligible performance degradation. Drowsy-RI saves more RF leakage energy (87.3%) than Drowsy-TA along with less performance degradation (2.7%) than Drowsy-IS. In terms of the total energy reduction, we find that Drowsy-TA (128 cycles) outperforms all other three schemes, leading to 7.0% reduction of the total energy dissipation with negligible performance overhead.

7.3 GPU Execution Units Power-Gating Strategies

The execution units power-gating strategies evaluated in this work are demonstrated to achieve considerable saving on leakage energy dissipation and improve

the GPU energy-efficiency. No additional microarchitecture with complex control logic is required and both hardware and performance overheads are negligible due to the simplicity. Through evaluating different parameter settings of the power-gating strategies, we find that an idle detect time should be involved to identify and filter short term idleness. On the other hand, the execution units can be put in the power-gated mode as long as they are unoccupied to maximize the leakage energy saving, if the overhead is affordable. For break even time down to 5 cycles, the immediate power-gating can achieve 84.3% execution units leakage energy reduction, which is almost the same with the ideal case (85.5%). For break even time up to 20 cycles, the idle detect power-gating can reduce the execution units leakage energy by 63.1%.

7.4 GPU Side-Channel Attacks

In this work, we propose a novel profiling-based side-channel attack on GPUs which leaks critical information to an adversary to fully recover the encryption secret key. The profiling-based side-channel attack accomplishes the high resolution profiling and is well scalable. The number of samples for recovering the key is dramatically reduced and much smaller than all the existing GPU side-channel attacks. We demonstrate this profiling-based side-channel attack on two Nvidia GPUs to recover 16-byte AES keys in as short as 30 seconds with very high accuracy (approaching 100%).

We further propose to use a boundary check Profiling-based Side-Channel Attack (pSCA) to overcome RCoal and to achieve the AES key recovery in a reasonable time. For the RCoal protected GPUs, the original strategy of the pSCA cannot recovery the AES key due to the randomness of memory coalescing. However, we apply a boundary check mechanism to successfully enable the pSCA to rebuild the AES key for RCoal methods with 2 to 8 subwarps. Consequently, in order to completely protect the GPU against the pSCA, the hardware designers will have to utilize the subwarp

configuration providing highest level security (i.e., 16 subwarps), which unfortunately will lead to significant GPU performance loss. According to the results, the bounday check pSCA achieves to rebuild the full 16-byte AES key in less than 14 day even though the GPU is protected by the RSS scheme that is supposed to offer the best security.

We finally propose a Profiling-Assisted Correlation-based Side-Channel Attack on GPUs. Owing to the significant correlation exploited and the immunity to noises, the number of samples is substantially reduced and is much smaller than all existing GPU SCAs. By checking the effectiveness of the pacSCA on revealing 192-bit and 256-bit AES keys, the pacSCA is proven to be well scalable. We further prove that the pacSCA can accomplish the information leakage even though a state-of-the-art defense mechanisms has been deployed. The experimental results show that the pacSCA can reveal the 128-bit AES key within 6 seconds. Under the protection of the Rcoal techniques, the pacSCA can obtain the secure key in less than 17 minutes. The purpose of this study is to arouse further researches on GPU side-channel threats and corresponding countermeasures.

7.5 Future Work

GPU's extensive threading capability can lead to a significant acceleration of machine learning algorithms. Various domains, including social media, automotive vehicles, medicine, and consumer electronics, have embraced machine learning to extract valuable insights from vast databases and abstract information. The scale of these databases has expanded exponentially, driven by the advancement of the internet, information technologies, and sensors. As a result of this growth, machine learning algorithms, characterized by their high computational complexity and data intensity, often experience extended execution times when running on CPUs. CPUs, being

more suited for general-purpose tasks, are gradually being supplemented by GPGPU computing platforms to leverage their performance-enhancing potential. The inherent ability of GPGPUs to handle numerous threads, coupled with their abundant floating-point operation units and high memory bandwidth, renders them well-suited for accommodating machine learning algorithms that heavily involve matrix multiplications.

The widespread use of GPUs in AI computing platforms has sparked a significant demand for further exploration into the GPU architectures specifically tailored for machine learning algorithms. Our future research endeavors will be centered around establishing a high-performance, energy-efficient, and highly secure GPU computing environment for AI applications. Specifically, although we have demonstrated that the register packing is still a convincing method to improve the energy-efficiency for AI applications, it is unknown that how effective the drowsy register file scheme and the computing units power-gating mechanism can be on reducing the energy consumption of AI applications. One of our future work will focus on evaluating these two works with AI applications and potentially making tailored adjustments. We will also extend the drowsy register file research by exploring advanced strategies to improve the prediction accuracy. Since the GPU cache locality of AI applications can be quite different with other applications, the existing cache replacement policies may not necessarily yield good results. As another part of our future work, we target analyzing the pattern of accesses to different cache hierarchies for AI applications and discovering specific policies to enhance the role of caches in improving the performance of AI applications. Our studies in this dissertation have already proved the vulnerabilities of the GPUs to the side-channel attacks and evaluated the limitation of the existing countermeasures. In the future, we will work on novel countermeasures which can deliver high security and low overhead so that the applications running on

GPUs can be well protected.

REFERENCES

[1] Y Drew. "A closer look at GPUs". In: *Communications of the ACM* 51.10 (2008).

[2] Shuai Che et al. "A performance study of general-purpose applications on graphics processors using CUDA". In: *Journal of parallel and distributed computing* 68.10 (2008), pp. 1370–1380.

[3] Pushpak Karnick. "GPGPU: General Purpose Computing on Graphics Hardware". In: *Pushpak's Home Page.[Online][Cited: November 19, 2008.] http://www. public. asu. edu/˜ pkarnic/portfolio/papers/IntraVis2006. pdf* ().

[4] CUDA Nvidia. *Programming guide*. 2008.

[5] Aaftab Munshi. "The opencl specification". In: *Hot Chips 21 Symposium (HCS), 2009 IEEE*. IEEE. 2009, pp. 1–314.

[6] Minsoo Rhu and Mattan Erez. "Maximizing SIMD resource utilization in GPGPUs with SIMD lane permutation". In: *ACM SIGARCH Computer Architecture News*. Vol. 41. 3. ACM. 2013, pp. 356–367.

[7] Mohammad Abdel-Majeed and Murali Annavaram. "Warped register file: A power efficient register file for GPGPUs". In: *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*. IEEE. 2013, pp. 412–423.

[8] Jiayuan Meng, David Tarjan, and Kevin Skadron. "Dynamic warp subdivision for integrated branch and memory divergence tolerance". In: *ACM SIGARCH Computer Architecture News*. Vol. 38. 3. ACM. 2010, pp. 235–246.

[9] Mark Gebhart et al. “Energy-efficient mechanisms for managing thread context in throughput processors”. In: *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*. IEEE. 2011, pp. 235–246.

[10] Wilson WL Fung and Tor M Aamodt. “Thread block compaction for efficient SIMT control flow”. In: (2011).

[11] Wilson WL Fung et al. “Dynamic warp formation and scheduling for efficient GPU control flow”. In: *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society. 2007, pp. 407–420.

[12] Aaron E Cohen and Keshab K Parhi. “GPU accelerated elliptic curve cryptography in GF (2 m)”. In: *Circuits and Systems (MWSCAS), 2010 53rd IEEE International Midwest Symposium on*. IEEE. 2010, pp. 57–60.

[13] Keisuke Iwai, Takakazu Kurokawa, and Naoki Nisikawa. “AES encryption implementation on CUDA GPU and its analysis”. In: *2010 First International Conference on Networking and Computing*. IEEE. 2010, pp. 209–214.

[14] Andrea Di Biagio et al. “Design of a parallel AES for graphics hardware using the CUDA framework”. In: *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*. IEEE Computer Society. 2009, pp. 1–8.

[15] Robert Szerwinski and Tim Güneysu. “Exploiting the power of GPUs for asymmetric cryptography”. In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 2008, pp. 79–99.

[16] Svetlin A Manavski et al. "CUDA compatible GPU as an efficient hardware accelerator for AES cryptography". In: *Signal Processing and Communications* 2007 (2007).

[17] Deguang Le et al. "Parallel AES algorithm for fast data encryption on GPU". In: *Computer Engineering and Technology (ICCET), 2010 2nd International Conference on*. Vol. 6. IEEE. 2010, pp. V6–1.

[18] Roberto Di Pietro, Flavio Lombardi, and Antonio Villani. "CUDA leaks: Information leakage in GPU architectures". In: *arXiv preprint arXiv:1305.7383* (2013).

[19] Roberto Di Pietro, Flavio Lombardi, and Antonio Villani. "CUDA leaks: a detailed hack for CUDA and a (partial) fix". In: *ACM Transactions on Embedded Computing Systems (TECS)* 15.1 (2016), p. 15.

[20] Sangho Lee et al. "Stealing webpages rendered on your browser by exploiting GPU vulnerabilities". In: *2014 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2014, pp. 19–33.

[21] Michael Patterson. "Vulnerability analysis of GPU computing". In: (2013).

[22] Janis Danisevskis, Marta Piekarska, and Jean-Pierre Seifert. "Dark side of the shader: Mobile gpu-aided malware delivery". In: *International Conference on Information Security and Cryptology*. Springer. 2013, pp. 483–495.

[23] C Nvidia. "Nvidia's next generation cuda compute architecture: Fermi". In: *Comput. Syst* 26 (2009), pp. 63–72.

[24] Sunpyo Hong and Hyesoon Kim. "An integrated GPU power and performance model". In: *ACM SIGARCH Computer Architecture News*. Vol. 38. 3. ACM. 2010, pp. 280–289.

[25] Jingwen Leng et al. “GPUWattch: enabling energy optimizations in GPGPUs”. In: *ACM SIGARCH Computer Architecture News*. Vol. 41. 3. ACM. 2013, pp. 487–498.

[26] Jieun Lim et al. “Power modeling for GPU architectures using McPAT”. In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 19.3 (2014), p. 26.

[27] David Brooks and Margaret Martonosi. “Dynamically exploiting narrow width operands to improve processor power and performance”. In: *High-Performance Computer Architecture, 1999. Proceedings. Fifth International Symposium On*. IEEE. 1999, pp. 13–22.

[28] Oguz Ergin et al. “Register packing: Exploiting narrow-width operands for reducing register file pressure”. In: *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society. 2004, pp. 304–315.

[29] Masaaki Kondo and Hiroshi Nakamura. “A small, fast and low-power register file by bit-partitioning”. In: *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*. IEEE. 2005, pp. 40–49.

[30] Jie Hu, Shuai Wang, and Sotirios G Ziavras. “In-register duplication: Exploiting narrow-width value for improving register file reliability”. In: *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*. IEEE. 2006, pp. 281–290.

[31] Jie Hu, Shuai Wang, and Sotirios G Ziavras. “On the exploitation of narrow-width values for improving register file reliability”. In: *IEEE transactions on very large scale integration (VLSI) systems* 17.7 (2009), pp. 953–963.

[32] Sangpil Lee et al. “Warped-compression: Enabling power efficient gpus through register compression”. In: *ACM SIGARCH Computer Architecture News*. Vol. 43. 3. ACM. 2015, pp. 502–514.

[33] Jingweijia Tan, Zhi Li, and Xin Fu. “Soft-error reliability and power co-optimization for GPGPUS register file using resistive memory”. In: *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. EDA Consortium. 2015, pp. 369–374.

[34] Syed Zohaib Gilani, Nam Sung Kim, and Michael J Schulte. “Power-efficient computing for compute-intensive GPGPU applications”. In: *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*. IEEE. 2013, pp. 330–341.

[35] Daniel Wong, Nam Sung Kim, and Murali Annavaram. “Approximating warps with intra-warp operand value similarity”. In: *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*. IEEE. 2016, pp. 176–187.

[36] Hyeran Jeon et al. “GPU register file virtualization”. In: *Proceedings of the 48th International Symposium on Microarchitecture*. ACM. 2015, pp. 420–432.

[37] Sheng Li et al. “CACTI-P: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques”. In: *Computer-Aided Design (ICCAD), 2011 IEEE/ACM International Conference on*. IEEE. 2011, pp. 694–701.

[38] Steven JE Wilton and Norman P Jouppi. “CACTI: An enhanced cache access and cycle time model”. In: *IEEE Journal of Solid-State Circuits* 31.5 (1996), pp. 677–688.

[39] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems* 25 (2012).

[40] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint arXiv:1409.1556* (2014).

[41] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[42] Forrest N Iandola et al. "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and¡ 0.5 MB model size". In: *arXiv preprint arXiv:1602.07360* (2016).

[43] Baoyuan Liu et al. "Sparse convolutional neural networks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 806–814.

[44] Nilanjan Goswami, Bingyi Cao, and Tao Li. "Power-performance co-optimization of throughput core architecture using resistive memory". In: *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*. IEEE. 2013, pp. 342–353.

[45] Ehsan Atoofian. "Reducing shift penalty in domain wall memory through register locality". In: *Proceedings of the 2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. IEEE Press. 2015, pp. 177–186.

[46] Majid Namaki-Shoushtari et al. "ARGO: aging-aware GPGPU register file allocation". In: *Proceedings of the Ninth IEEE/ACM/IFIP International Con-*

ference on Hardware/Software Codesign and System Synthesis. IEEE Press. 2013, p. 30.

[47] Sparsh Mittal and Jeffrey S Vetter. "A survey of methods for analyzing and improving GPU energy efficiency". In: *ACM Computing Surveys (CSUR)* 47.2 (2015), p. 19.

[48] Nagesh B Lakshminarayana and Hyesoon Kim. "Spare register aware prefetching for graph algorithms on GPUs". In: *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*. IEEE. 2014, pp. 614–625.

[49] Krisztián Flautner et al. "Drowsy caches: simple techniques for reducing leakage power". In: *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*. IEEE. 2002, pp. 148–157.

[50] Chih-Chieh Hsiao, Slo-Li Chu, and Chiu-Cheng Hsieh. "An adaptive thread scheduling mechanism with low-power register file for mobile GPUs". In: *IEEE Transactions on Multimedia* 16.1 (2014), pp. 60–67.

[51] Hao Wen and Wei Zhang. "Reducing cache leakage energy for hybrid SPM-cache architectures". In: *Compilers, Architecture and Synthesis for Embedded Systems (CASES), 2014 International Conference on*. IEEE. 2014, pp. 1–9.

[52] Ehsan Atoofian and Ali Manzak. "Power-aware L1 and L2 caches for GPGPUs". In: *European Conference on Parallel Processing*. Springer. 2014, pp. 354–365.

[53] Ehsan Atoofian. "Reducing Static and Dynamic Power of L1 Data Caches in GPGPUs". In: *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*. IEEE. 2014, pp. 798–804.

[54] Ehsan Pakbaznia and Massoud Pedram. "Design and application of multimodal power gating structures". In: *Quality of Electronic Design, 2009. ISQED 2009. Quality Electronic Design*. IEEE. 2009, pp. 120–126.

[55] Stefanos Kaxiras, Zhigang Hu, and Margaret Martonosi. "Cache decay: exploiting generational behavior to reduce cache leakage power". In: *ACM SIGARCH Computer Architecture News* 29.2 (2001), pp. 240–251.

[56] Qiumin Xu and Murali Annavaram. "PATS: pattern aware scheduling and power gating for GPGPUs". In: *Proceedings of the 23rd international conference on Parallel architectures and compilation*. ACM. 2014, pp. 225–236.

[57] Mohammad Abdel-Majeed, Daniel Wong, and Murali Annavaram. "Warped gates: gating aware scheduling and power gating for GPGPUs". In: *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM. 2013, pp. 111–122.

[58] Po-Han Wang et al. "Power gating strategies on GPUs". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 8.3 (2011), p. 13.

[59] Minsoo Rhu et al. "A locality-aware memory hierarchy for energy-efficient GPU architectures". In: *Microarchitecture (MICRO), 2013 46th Annual IEEE/ACM International Symposium on*. IEEE. 2013, pp. 86–98.

[60] Yue Wang, Soumyaroop Roy, and Nagarajan Ranganathan. "Run-time power-gating in caches of GPUs for leakage energy savings". In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012*. IEEE. 2012, pp. 300–303.

[61] Zhigang Hu et al. “Microarchitectural techniques for power gating of execution units”. In: *Proceedings of the 2004 international symposium on Low power electronics and design*. ACM. 2004, pp. 32–37.

[62] Chao Luo et al. “Side-channel power analysis of a GPU AES implementation”. In: *2015 33rd IEEE International Conference on Computer Design (ICCD)*. IEEE. 2015, pp. 281–288.

[63] Zhen Hang Jiang, Yunsi Fei, and David Kaeli. “A novel side-channel timing attack on GPUs”. In: *Proceedings of the on Great Lakes Symposium on VLSI 2017*. ACM. 2017, pp. 167–172.

[64] Zhen Hang Jiang and Yunsi Fei. “A novel cache bank timing attack”. In: *Proceedings of the 36th International Conference on Computer-Aided Design*. IEEE Press. 2017, pp. 139–146.

[65] Yiwen Gao et al. “Cache-Collision Attacks on GPU-Based AES Implementation with Electro-Magnetic Leakages”. In: *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*. IEEE. 2018, pp. 300–306.

[66] Zhen Hang Jiang, Yunsi Fei, and David Kaeli. “A complete key recovery timing attack on a GPU”. In: *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*. IEEE. 2016, pp. 394–405.

[67] Chao Luo et al. “Power Analysis Attack of an AES GPU Implementation”. In: *Journal of Hardware and Systems Security* 2.1 (2018), pp. 69–82.

[68] Gurunath Kadam, Danfeng Zhang, and Adwait Jog. “RCoal: mitigating GPU timing attack via subwarp-based randomized coalescing techniques”. In: *2018*

IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE. 2018, pp. 156–167.

[69] Hoda Naghibijouybari et al. "Rendered Insecure: GPU Side Channel Attacks are Practical". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2018, pp. 2139–2153.

[70] Abdullah Al-Mamun et al. "Security analysis of AES and enhancing its security by modifying S-box with an additional byte". In: *International Journal of Computer Networks & Communications (IJCNC)* 9.2 (2017).

Appendix A

PUBLICATION

1. Xin Wang and Wei Zhang. "Reducing GPU Energy Consumption by Packing Narrow-Width Operands." In: *J. Comput. Sci. Eng.* 15.4 (2021), pp. 135–147.

2. Xin Wang and Wei Zhang. "pacSCA: A Profiling-Assisted Correlation-based Side-Channel Attack on GPUs". In: *2020 IEEE 38th International Conference on Computer Design (ICCD)*. IEEE. 2020, pp. 525–528.

3. Xin Wang and Wei Zhang. "Packing narrow-width operands to improve energy efficiency of general-purpose GPU computing". In: *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE. 2020, pp. 1–7.

4. Xin Wang and Wei Zhang. "GPGPU Functional Units Power Gating for Leakage Energy Reduction." In: *J. Comput. Sci. Eng.* 14.3 (2020), pp. 102–111.

5. Xin Wang and Wei Zhang. "Exploring Time-Predictable and High-Performance Last-Level Caches for Hard Real-Time Integrated CPU-GPU Processors". In: *Journal of Computing Science and Engineering* 14.3 (2020), pp. 89–101.

6. Xin Wang and Wei Zhang. "Execution Units Power-Gating to Improve Energy Efficiency of GPGPUs". In: *2019 International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. IEEE. 2019, pp. 711–718.

7. Xin Wang and Wei Zhang. "Cracking randomized coalescing techniques with

an efficient profiling-based side-channel attack to GPU". In: *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*. 2019, pp. 1–8.

8. Xin Wang and Wei Zhang. "An efficient profiling-based side-channel attack on graphics processing units". In: *National Cyber Summit*. Springer. 2019, pp. 126–139.

9. Xin Wang and Wei Zhang. "Energy-Efficient DNN Computing on GPUs Through Register File Management". In: *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE. 2018, pp. 1–7.

10. Xin Wang and Wei Zhang. "Packing Narrow-Width Operands to Improve GPU Performance". In: *Journal of Computing Science and Engineering* 12.2 (2018), pp. 37–49.

11. Xin Wang and Wei Zhang. "Improving CPU and GPU Performance through Sample-Based Dynamic LLC Bypassing". In: *Journal of Computing Science and Engineering* 12.2 (2018), pp. 50–62.

12. Xin Wang and Wei Zhang. "Drowsy Register Files for Reducing GPU Leakage Energy". In: *Parallel and Distributed Systems (ICPADS), 2017 IEEE 23rd International Conference on*. IEEE. 2017, pp. 632–639.

13. Xin Wang and Wei Zhang. "Gpu register packing: Dynamically exploiting narrow-width operands to improve performance". In: *2017 IEEE Trustcom/BigDataSE/ICESS*. IEEE. 2017, pp. 745–752.

14. Xin Wang and Wei Zhang. "A Sample-Based Dynamic CPU and GPU LLC Bypassing Method for Heterogeneous CPU-GPU Architectures". In: *2017 IEEE*

Trustcom/BigDataSE/ICESS. IEEE. 2017, pp. 753–760.

15. Xin Wang and Wei Zhang. "Cache locking vs. partitioning for real-time computing on integrated CPU-GPU processors". In: *2016 IEEE 35th International Performance Computing and Communications Conference (IPCCC)*. IEEE. 2016, pp. 1–8.

VITA

Xin Wang was born on June 15, 1985, in Wuhan, Hubei Province, China, and is a Chinese citizen. He graduated from No.3 Middle School, Wuhan, China in 2004. He received his Bachelor of Science degree in Electrical Engineering from Peking University, Beijing, China in 2008. He received a Master of Science degree in Underwater Acoustics Engineering from Naval University of Engineering, Wuhan, China in 2010. He was a Driver Development Engineer at Mitac Shanghai in 2012 and worked for Shanghai BIO-TAG Corp. as a R&D Engineer from 2013 to 2015. He now is a Firmware Engineer at Micron Technology.