Investigating Register Cache Behavior: Implications for CUDA and Tensor Core Workloads on GPUs

Vahid Geraeinejad, Qiran Qian, and Masoumeh Ebrahimi, *Senior Member, IEEE*

***Abstract*— GPUs are extensively employed as the primary devices for running a broad spectrum of applications, covering general-purpose applications as well as Artificial Intelligence (AI) applications. Register file, as the largest SRAM on the GPU die, accounts for over 20% of the total GPU energy consumption. Register cache has been introduced to reduce traffic from the register file and thus decrease total energy consumption when CUDA cores are utilized. However, the utilization of register cache has not been thoroughly investigated for Tensor Cores which are integrated into recent GPU architectures to meet AI workload demands. In this paper, we study the usage of register cache in both CUDA and Tensor Cores and conduct a thorough examination of their pros and cons. We have developed an open-source analytical simulator, called RFC-sim, to model and measure the energy consumption of both the register file and register cache. Our results show that while the register cache can reduce energy consumption by up to 40% in CUDA cores, it results in increased energy consumption by up to 23% in Tensor Cores. The main reason lies in the limited space of the register cache, which is not sufficient for the demand of Tensor cores to capture locality.**

***Index Terms*— GPGPU, CUDA core, tensor core, register file, register cache, GEMM, energy efficiency, cache traffic.**

I. Introduction

GPUs are widely utilized as primary devices for executing a diverse range of applications, encompassing general-purpose tasks and Artificial Intelligence (AI) applications powered by Deep Neural Networks (DNN) [1]. This capability stems from the inherent parallel processing power enabled by simultaneously executing multiple instances of the same instruction with different data, a form of parallel processing commonly referred to as Single Instruction Multiple Data (SIMD).

Conventionally, the GPU architecture, as illustrated in Figure 1a, comprises multiple processing units called Streaming Multiprocessors (SMs) capable of data exchange with higher levels of the memory hierarchy, including the L2 cache and global memory, through the utilization of the Network on Chip (NoC) and memory controllers. Each SM, as depicted in Figure 1b, features multiple processing blocks, its own exclusive L1 cache, and shared memory. These processing blocks, outlined in Figure 1c, are equipped with multiple CUDA cores to execute general-purpose computing workloads and Tensor Cores (TCs) [2] to execute matrix multiplication workloads as well as a private main Register File (RF), collectively forming the Single Instruction Multiple Thread (SIMT) architecture. In order to refine the focus of our research, we narrow our attention specifically to NVIDIA terminology and GPU microarchitecture.

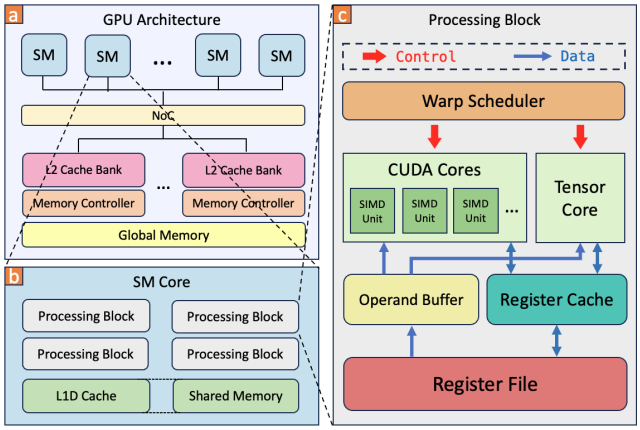


Fig. 1. Representative GPU architecture. (a) GPU top-level architecture; (b) SM core organization; (c) processing block architecture illustrating the presence of CUDA cores, Tensor Core, and register cache.

Energy efficiency has emerged a significant focal point of research due to its pivotal role in optimizing GPU performance and conserving energy resources, particularly within the realm of data center computing [3]. According to [4], the RF accounts for more than 20% of the entire GPU's energy consumption. To address the significant energy consumption attributed to the RF, a technique known as Register Caching (RC) [5] has been introduced. Borrowed from the CPU realm by Gebhart et al. [6], RC in GPUs entails adding a secondary but smaller and faster register file to the SM cores. This aims to mitigate the limitations of the RF by reducing data traffic and enhancing energy efficiency. While the implementation of RC has been speculated in previous GPU generations by previous works [7], including Fermi [8] and its successors such as Volta [9] and Turing [10], its examination concerning configuration, impact on the RF, and energy efficiency remains to be explored. Furthermore, with the recent integration of TCs in GPUs, the benefits or drawbacks of register cache within the GPU TCs remain largely unexplored.

To sustain the computational demand for AI/HPC workloads involving intensive Matrix Multiply Accumulate (MMA)

Manuscript received 17 March 2024; revised 5 June 2024 and 21 July 2024; accepted 26 July 2024. Date of publication 5 August 2024; date of current version 13 September 2024. This work was supported by the STINT under Project MG20188007. This article was recommended by Guest Editor J. Kim. *(Vahid Geraeinejad and Qiran Qian contributed equally to this work.) (Corresponding author: Vahid Geraeinejad.)*

The authors are with the KTH Royal Institute of Technology, 164 40 Stockholm, Sweden (e-mail: vahidg@kth.se).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/JETCAS.2024.3439193.

Digital Object Identifier 10.1109/JETCAS.2024.3439193

computations, NVIDIA has introduced TC as a heterogeneous computing paradigm since Volta architecture [9]. TC hardware comprises SIMD units where the lanes collectively conduct an MMA [11] operation. TC instructions intrinsically perform MMAs of specific sizes. MMA instructions utilize multiple General Purpose Registers (GPR) to store matrix fragments in their operands. As a result, managing operands amidst heavy RF traffic poses a challenge, potentially resulting in increased latency and energy consumption during data movement.

Our objective in this study is to assess the traffic and energy efficiency of both RF and RC during the execution of general-purpose and MMA-intensive workloads utilizing CUDA cores and TCs, respectively. To the best of our knowledge, this study marks the first comprehensive examination of the mentioned workload types employing RC in GPGPUs. The main contributions of this paper are as follows:

- We have conducted an extensive analysis of the RF and RC, evaluating their traffic and energy consumption across diverse scenarios. This analysis involves utilizing general-purpose workloads executed on CUDA cores. Furthermore, we thoroughly explored the usage of RC for MMA-intensive workloads executed on TCs, uncovering new insights. To achieve this, we assessed the RC design using various metrics, including cache associativity, allocation policy (write-back and compiler-aided), and destination register operand mapping.
- We have developed an analytical simulator from the ground up, called RFC-sim, to profile the behaviour of the RF and RC and extract our statistics based on the memory configuration and the input code. The source codes of RFC-sim are available on https://github.com/BrianQian1999/RFC-sim.
- Our approach demonstrates that the optimal RC configuration yields a significant improvement in RF energy efficiency, achieving up to a 40% reduction compared to the baseline GPU architecture in general-purpose workloads executed on CUDA cores. Surprisingly, in the case of MMA-intensive workloads on TCs, the use of RC not only fails to conserve energy but also leads to a notable increase in power consumption of up to 23%, attributed to its incapacity to capture locality. This adverse impact persists across all examined RC configurations. Lastly, we offer fresh insights and suggest directions for future research to optimize RC utilization in TCs.

II. Related Work

Extensive research has focused on enhancing GPGPU performance and energy efficiency. This section delves into relevant studies, emphasizing GPGPU architecture modeling and register file optimization, which are closely related to this study's subject matter.

A. GPGPU Modeling

For computer architecture researchers, possessing precise, fast, and scalable analytic/timing/power models for their hardware of interest is necessary. Relying on simulations is the prevailing approach in the field of computer architecture [12]. It enables us to assess the performance of designs without needing to delve into the intricacies of circuit-level analysis.

As for the GPGPU models, Bakhoda et al. [13] developed a detailed cycle-accurate simulator called GPGPU-Sim targeting the CUDA workloads based on an NVIDIA Fermi-like model. Expanding upon prior research, GPGPU-Sim is further developed and verified to accommodate the latest GPGPU architectures and TC architecture in [14]. However, GPGPU-Sim can only accept the PTX (Parallel Thread eXecution) and an enhanced instruction set called PTXAS as the input instead of SASS (Streaming ASSembler), as elaborated later. This limitation makes it relatively inaccurate and could barely support the TC workloads using in-house libraries including cuBLAS and cuDNN [2]. To create a workaround, Mahmoud et al. further extended the original GPGPU-Sim to a framework called Accel-Sim [15], which natively supports SASS by utilizing the instrumentation tool called NVBit [16] provided by NVIDIA. NVBit traces all machine instructions executed by the GPU at runtime and dumps the traces, enhancing SASS code visibility for common users.

Both GPGPU-Sim and Accel-Sim are associated with a power model called GPUWattch [4]. GPUWattch power model is highly configurable and enables potential energy optimizations by providing a cycle-level energy consumption evaluation and validation against the real hardware. Besides, it considers dynamic voltage and frequency scaling (DVFS) and clock gating. AccelWattch [17] is the upgraded version of GPUWattch and is embedded into the Accel-Sim framework.

B. GPGPU Register File Optimization

GPU register file is the largest SRAM structure on the die and consumes significant energy. Numerous studies have proposed optimization methods for the register file architecture to enhance performance and energy efficiency in GPGPUs.

Gebhart et al. [18] proposed to adopt a multi-level register file hierarchy to migrate the idea of register caching from the CPU realm to the GPU realm. The register cache exploits the temporal locality of register accesses to mitigate the traffic to the main register file. Additionally, the author suggests enhancing this approach with compiler assistance, utilizing a bit in the instruction encoding to direct the de-allocation of unused register values within the register cache instead of writing them back to the main register file. Rather than focusing on improving single-thread IPC similar to CPU implementation [6], the 2-level register cache is designed to reduce energy consumption while preserving performance.

Jeon et al. [19] borrowed the insight of virtualization from the design of the memory system and utilized the technique to optimize the register file. Firstly, it proposes to use register renaming to elaborate a virtualized layer on top of the physical registers. Further, compiler-aided allocation and de-allocation of register values could enhance the method by actively de-allocating dead values and re-allocating the register resource to other warps. Such shrinkage of the physical registers file reduces both the dynamic and static energy consumption, while not harming performance.

Khorasani et al. [20] proposed sharing register file resources by logically separating the physical register file into a basic

register set and an extended register set via the compiler. The basic register set is private to each warp, while the extended register set is time-shared by warps, as determined at compile time. This compiler-aided architecture allows registers to be visible and easily manipulated at the assembly level, enabling techniques to control the allocation and de-allocation of register resources. These techniques improve hardware utilization, reduce area, and enhance energy efficiency.

Abdel-Majeed et al. [21] propose a partitioned GPU register file design employing near-threshold voltage (NTV) technology, capitalizing on the observation that applications often use a small subset of registers. This design divides registers into two partitions: highly accessed registers stored in a small RF switch between high and low power modes, while the rest are stored in a large RF partition consistently operating at NTV. This approach substantially reduces power consumption while minimally impacting performance.

While prior approaches reduce register access energy by using a smaller RC or a compiler-managed scratchpad [18], Bailey et al. [22] propose a hybrid design that combines both, leveraging runtime and compile-time information to capture a wider range of register accesses. Their approach manages to significantly reduces register energy by up to 38.7% of the baseline, surpassing RC or a register scratchpad alone.

Sadrosadati et al. [23] address challenges related to large register files in GPUs, such as scalability due to long access latency and power consumption. They introduce the Latency-Tolerant Register File (LTRF) architecture, aimed at mitigating these challenges by prefetching register sets from the main register file to the cache, thus reducing latency. Their results demonstrate the effectiveness of LTRF, enabling optimizations like utilizing high-density memory for the main register file and significantly enhancing GPU performance.

Guan et al. [24] introduce PresCount, a novel register allocation method designed to efficiently mitigate bank conflicts. By enhancing the Register Conflict Graph (RCG) coloring strategy and incorporating a bank pressure tracking mechanism, PresCount significantly reduces bank conflicts on platforms with rich register banks and limited budgets. Additionally, a subgroup splitting technique facilitates register allocation under the bank-subgroup register file design, achieving a 99.85% reduction in bank conflicts for specific kernel domains such as AI computing.

Introduced by Shoushtary et al. [25] Malekeh is a low-cost, hierarchical register file cache mechanism that repurposes Operand Collector Units (OCUs) to cache registers and capture temporal reuses with minimal overhead. By keeping the number of OCUs low, Malekeh avoids significant overheads while managing effective allocation, replacement, and write policies through compiler-guided strategies. This approach maximizes energy savings and performance by leveraging register reuse distance and a dynamic algorithm that enhances hit ratio and efficiency based on runtime behavior.

While drawing on previous insights, the simulator in this study independently explores the intriguing impact of register caching on NVIDIA GPUs by evaluating workloads executed on both CUDA cores and TCs.

III. Background and Motivation

In this chapter, we embark on an exploration of the fundamental aspects of GPU applications, encompassing workloads executed on both CUDA cores and TCs, and examining their pivotal roles in parallel processing. Subsequently, we delve into the TC General Matrix Multiply (GEMM) Paradigm, concentrating on its utilization in matrix multiplication. We discuss the associated data flow and instruction structure, shedding light on its challenges, and examine its potential impact on register file traffic and energy efficiency. Finally, we present an energy consumption breakdown including both general-purpose and MMA-intensive workloads, underscoring the significant contribution of the RF to the GPU's overall energy consumption. This analysis serves as a key motivator for the research conducted in this study.

A. GPU Applications and CUDA Cores

Generally, as shown in Figure 2b, GPU applications comprise compute kernels containing one or several executable thread blocks (TBs). These TBs, defined by the software programmer, are each assigned to an SM core. Then they are divided into groups of threads (typically 32, based on the GPU architecture) called warps by the GPU scheduler. Finally, warp scheduler assign warps to be executed concurrently on CUDA cores (SIMD units). CUDA cores are the fundamental processing units responsible for executing parallel tasks within SM processing blocks.

For instance, as depicted in Figure 2a, a kernel may perform an element-wise addition of two arrays (with a length of 1024 in this example). The kernel is organized into thread blocks defined by the programmer, where each block contains 128 threads (specific to this example). Finally, the thread blocks are further subdivided into warps, typically comprising 32 threads on a GPU. Each thread is tasked with adding two elements from each array and storing the resulting sum. Threads are then allocated to CUDA cores for parallel execution. CUDA cores are specialized for different types of computations including FP64 (double-precision floating-point), FP32 (single-precision floating-point), and INT32 (integer) arithmetic tasks.

B. Tensor Core GEMM Paradigm and Data Flow

Found in modern NVIDIA GPU architectures, TCs are specialized processing units crafted to accelerate deep learning and AI workloads by optimizng MMA operations used mainly in neural network training and inference phases. Initially introduced by NVIDIA in their Volta architecture and perpetuated in continued generations, TC hardware consists of SIMD units where thread lanes collectively execute MMA operations, demonstrating its proficiency in boosting throughput and enhancing energy efficiency. The hierarchical scheme of a GEMM routine on TC is illustrated in Figure 2c. The operation that is also referred to as blocked GEMM is split into thread block tiles where each of them is assigned to an SM core for execution. Meanwhile, thread block data is moved from global memory to the local shared memory of their respective SM. A thread block tile is further broken into warp tiles, where each

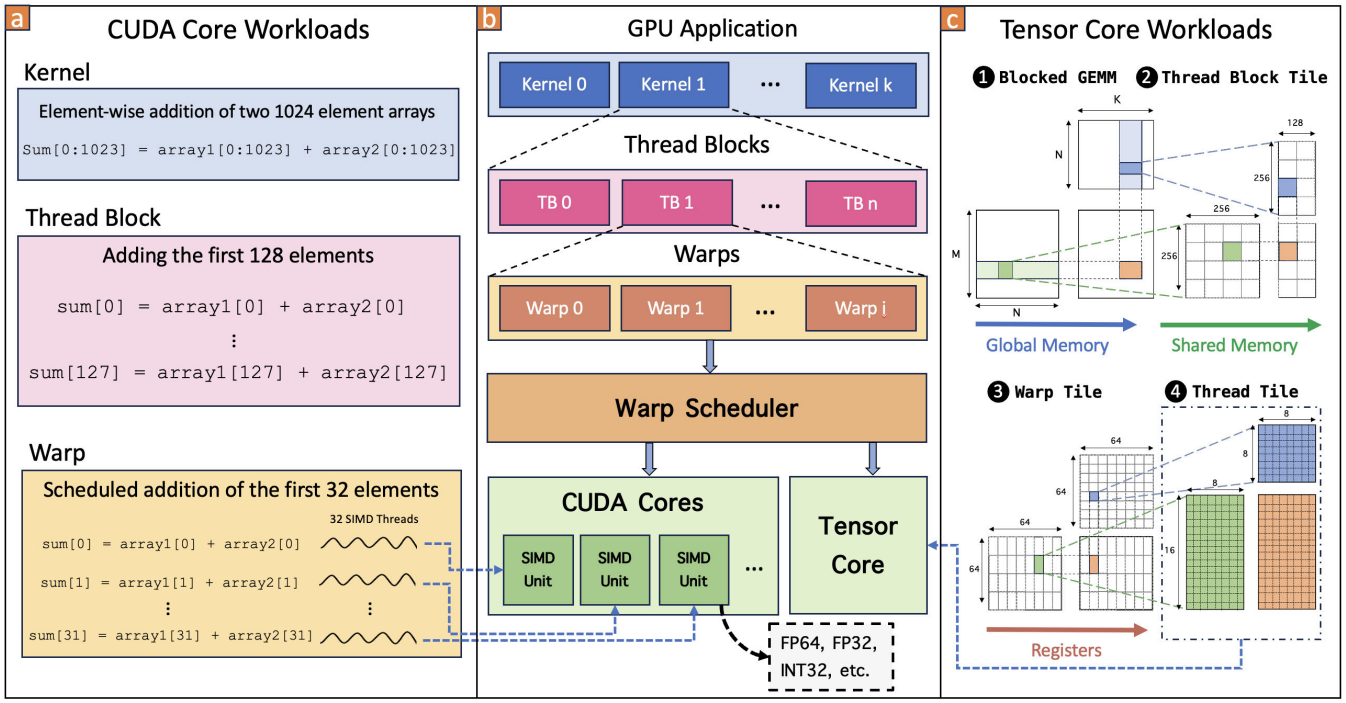


Fig. 2. GPU application execution (a) CUDA core application breakdown; (b) Top-level GPU application breakdown; (c) Tensor Core application breakdown.

TABLE I
CUDA AND TENSOR CORE SASS INSTRUCTION DESCRIPTION

Instruction	Description	Src regs	Dest regs	Reg size	Execution unit
R4 IADD3 R1,R2,R3	3-input integer addition	3 {R1,R2,R3}	1 {R4}	4-byte	INT32 CUDA core
R40 HMMA.1688.FP32 R10,R20,R40	16x16x8 half-precision matrix multiply and accumulate	7 {R10,R11} {R20} {R40,R41,R42,R43}	4 {R40,R41, R42,R43}	32-lane 4-byte	Tensor Core

warp tile is executed by a specific warp execution flow. Each one of the thread tiles which composes a warp tile is defined by a TC MMA instruction and executed by the cooperative threads in the TC unit.

C. GPU SASS Instructions

As the lower level to the virtual ISA (i.e., PTX), SASS is the lowest level of the assembly, whose codes are compiled into binary and executed natively on NVIDIA GPU hardware.

SASS codes contain both instruction information and scheduling control information to guide the hardware data path and control logic. These instructions typically include a descriptive opcode, followed by several destination and source operands that specify the input and output registers. The operations are dissected via reverse engineering using instruction traces from NVBit [16]. Here are two examples of CUDA core and TC SASS instructions.

1) CUDA Core Example: $R4$ $IADD3$ $R1, R2, R3$, as shown in Table I, denotes a CUDA Core SASS instruction executed by a thread utilizing CUDA core SIMD units. In this instruction, I stands for integer, and it performs a 3-input addition on three source operands, $R1$, $R2$, $R3$, with the resulting sum written back to the destination operand $R4$. Typically, operand registers in CUDA core instructions are 4-byte data words that occupy an entry in the RF or RC.

2) Tensor Core Example: TC instructions intrinsically perform MMAs of specific sizes such as $R40$ $HMMA.1688.FP32$ $R10, R20, R40$, as shown in Table I. This SASS instruction represents an MMA operation of $A_{16\times8} \times B_{8\times8} + C_{16\times8} = D_{16\times8}$, or simply the $m16n8k8$ (1688) MMA operation. In the instruction, H implies half-precision (16-bit) for the input matrices (i.e., A and B) while $FP32$ represents a full-precision format for the accumulator matrix C and the resulting matrix D.

A TC MMA may require a number of GPRs to store the matrix fragments. Matrix fragments should first be loaded from the shared memory to RF using dedicated load instructions. For example, the instruction $HMMA.1688.FP32$ in the Turing architecture that accumulates on FP16 inputs and FP32 accumulator matrices, 2 GPRs hold $A_{16\times8}$, 1 GPRs holds $B_{8\times8}$, and 4 GPRs are required to hold each matrix $C_{16\times8}$ and $D_{16\times8}$. Generally, MMA GPRs (used in TC specific instructions) are 32-lane registers capable of storing 32 FP32 values or 64 FP16 values. Assuming row-major A and column-major B, the data element layout of specific lanes corresponding to the fragments is illustrated in Figure 3. Distinct colors in the figure denote the GPRs containing the data, where each index number corresponds to the lane number within the 32-lane GPR (two FP16 values or one FP32 value stored in one 4-byte data word). Unlike the conventional CUDA core instructions where the operands are explicitly

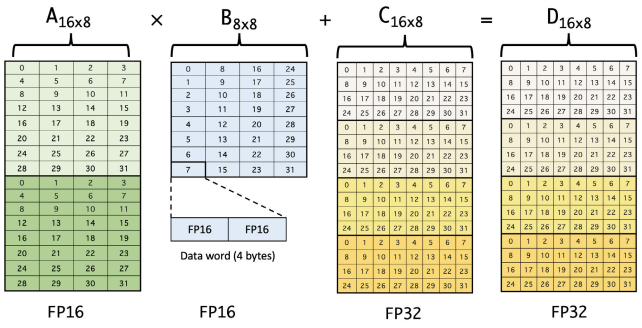
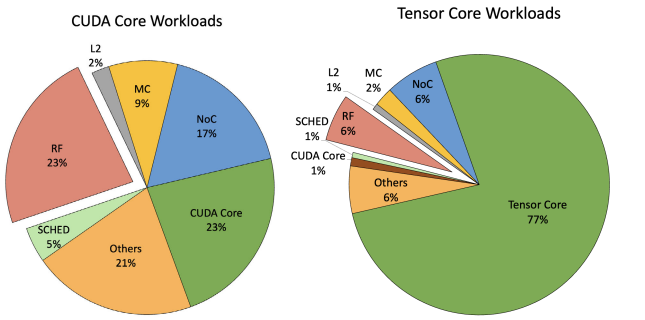


Fig. 3. Matrix fragment layout of TC thread tile MMA ($HMMA.1688.FP32$ Instruction).



(a) CUDA workloads (b) Tensor Core workloads

Fig. 4. GPU components average energy consumption breakdown using (a) CUDA and (b) Tensor Core workloads.

visible in the assembly, e.g., $R4$ $IADD3$ $R1, R2, R3$, the operands involved in a TC instruction are implicitly implied in the semantics. For instance, the TC instruction $R40$ $HMMA.1688.FP32$ $R10, R20, R40$ appears to expose $R40$ as the destination, and $R10, R20, R40$ as the sources, while it actually indicates a result matrix held by $\{R40, R41, R42, R43\}$, and similarly the source matrices held by $\{R10, R11\}, \{R20\}$, $\{R40, R41, R42, R43\}$ respectively. That is, the instruction exposes 4 *destination registers* and 7 *source registers*.

Given the characteristics of TC MMA, it is challenging to deliver a large number of instruction operands, specifically source operands, from the register file. This may cause a high traffic load to the register file which results in increased latency and energy consumption. In this research, compiler-aided allocation policy is utilized to further capture the locality among source operands within register caching, aiming to boost the energy efficiency of the primary register file by efficiently managing its traffic flow.

D. GPU Applications Energy Consumption Breakdown

To comprehensively analyze the overall GPU energy consumption across various workloads and GPU components, while also assessing the impact of the RF, we conducted a thorough energy consumption breakdown analysis encompassing two main categories: 1) general-purpose workloads executed on CUDA cores and 2) MMA-intensive workloads executed on TCs. For the general-purpose workloads, we selected eight benchmarks from Rodinia 3.1 benchmark suite [26], as detailed in Table IV. These benchmarks represent a diverse range of computational tasks commonly encountered in GPU computing. Additionally, for the MMA-intensive workloads, we focused on the squared GEMM operation, incorporating two different data formats: half-precision (FP16 inputs) and INT8, covering diverse computational demands and data types. This experiment is simulated using AccelWattch [17] on a RTX 2060 GPU architecture. The findings from our analysis of energy consumption, as depicted in Figure 4, show the average breakdown of energy usage across different GPU components over two benchmark scenarios. Our experiments demonstrate that, on average, the RF contributes approximately 23% to the total GPU energy consumption in general-purpose workloads, consistent with previous research findings [4]. However, in MMA-intensive workloads, RF energy consumption accounts for only 6% of the total energy consumption.

While the average energy consumption of the RF in MMA-intensive workloads may appear relatively smaller compared to general-purpose workloads, it is crucial to acknowledge that TCs are frequently recognized for their trade-offs, emphasizing high throughput and performance over energy efficiency, as also reflected in our findings. For example, our MMA benchmarks, averaging 44.8 W, were nearly four times more energy-intensive than the general-purpose workloads, which averaged 12.2 W. Consequently, the total energy consumption by the RF was nearly equivalent across both types of workloads, averaging 2.8 W for MMA and 2.6 W for general-purpose workloads. Given that RF is a significant contributing factor to overall GPU energy consumption, we are highly motivated to investigate how the integration of RC can affect RF in terms of both traffic and energy efficiency.

IV. Register Cache Model

In this section, we detail our implementation of register caching on the GPU, along with the incorporation of compiler-aided cache allocation policy based on [7]. While register caching is not a novel concept in CPUs, its proposal seeks to alleviate bank conflicts in the RF and enhance overall energy efficiency. According to Gebhart et al. [6], the introduction of register caching in GPUs has the potential to reduce RF energy consumption by 36%, achieved by capturing temporal locality among register operands in CPUs.

As depicted in Figure 1c, the RC reserves data in a small SRAM near the ALUs, bypassing the route from RF, operand buffer, and the interconnect to both CUDA cores and TCs. RC's location helps mitigate expensive RF transactions. The following sections describe the principal cache concepts and expound on the implementation of the RC model in this study.

A. Cache Associativity

Cache associativity refers to the relationship between cache entries and memory addresses and determines how memory locations are mapped to cache entries. In a set-associative cache, the cache is divided into multiple sets, with the associativity level indicating how many cache entries are in each set. Figure 5 shows RC utilization when executing the MMA instruction $R40$ $HMMA.1688.FP32$ $R10, R20, R40$. In this

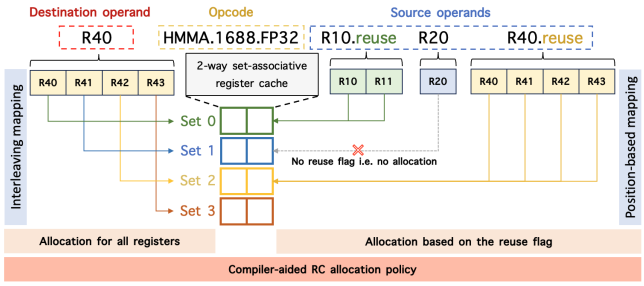


Fig. 5. Register cache associativity, operand allocation policy, and cache set mapping depicted with an MMA instruction.

example, a 2-way set-associative RC, where each set consists of 2 cache entries is depicted. In this particular scenario, the corresponding RC encompasses 8 cache entries. The total number of sets can be calculated as the number of cache entries divided by the associativity level. Generally, higher cache associativity leads to a lower cache miss rate, as it reduces conflict misses. However, it accounts for timing penalties and higher energy consumption. Investigating the energy efficiency of RC, it is essential to consider the cache associativity and characterize the trade-offs between the cache hit-rate and the energy cost of cache accesses. As for the realistic scenario of NVIDIA GPUs, there are 2 alternatives regarding the cache associativity. For older GPU architectures [6], a fully associative cache is utilized to lift the hit-rate by offering maximum flexibility while requiring complex searching algorithms. However, newer GPUs [7] adopt a 2-way associative cache that maps source register operands to cache sets based on operand position. Both models are considered in this study to explore the advantages and disadvantages of this evolution when implementing the RC in GPUs.

B. Cache Allocation

In the realm of memory cache, the cache allocation policy dictates whether data should be fetched from the memory and occupy a cache line in the event of a cache miss. One such allocation policy is *read-allocate*, where data is written to cache in the event of a cache read-miss. As shown in Table II, under this policy only source operands could be cached. Another common allocation policy is *write-allocate*, where data is written to cache during a cache write-miss. This policy leads to only the destination operands being cached. There is another allocation policy which is a blend of both, commonly known as *read & write-allocate*, where both source and destination operands could be cached.

This concept extends to RC, but due to limited space and low locality among source operands, previous approaches have often used write-allocate policy to load only destination operands. However, to harness the high locality of source operands, particularly in MMA-intensive workloads, reuse flags have been incorporated to determine which source operands to load in case of a cache miss, referred to as the *compiler-aided* allocation policy. Speculated in post-Volta NVIDIA GPU architectures, this policy utilizes software-side information and embeds 4-bit reuse flags (one for each source operand) into the 128-bit ISA to leverage source register

TABLE II
RC ALLOCATION POLICIES FOR SOURCE AND DESTINATION OPERANDS

Allocation policy	Source operands	Destination operands
Read-allocate	Allocate all	-
Write-allocate	-	Allocate all
Read & Write	Allocate all	Allocate all
Compiler-aided	Reuse allocate	Allocate all

operand reuse [7]. We observed reuse flags across both CUDA and Tensor operations using NVBit [16] instruction traces.

In the MMA instruction example illustrated in Figure 5, the compiler-aided allocation policy is adopted where cache entries are allocated to all the registers associated with the destination operand $R40$, in case of cache write-miss. On the other side, as the compiler has assigned reuse flags to two of the source operands, namely $R10$ and $R40$, cache entries are only allocated to these registers, in case of cache read-miss.

C. Cache Set Mapping

While cache allocation policy decides whether the data should occupy a cache entry, cache set mapping in RC determines 1) which cache set is a destination register mapped to if it is allocating a cache line, and 2) which cache set is a source register mapped to, if it is allocating a cache line. The RC implementation in this study employs distinct cache set mapping schemes for both source and destination registers.

1) Source registers: For the cache set mapping of source operands, the position-based scheme is adopted as indicated by the SASS instruction encoding, which can be expressed by:

$$SetIndex = (Position\ of\ Reg) \mod (\#\ of\ Cache\ Sets) \tag{1}$$

where the position can be one of $\{0, 1, 2, 3\}$ determined by the register's position in the instruction. This mapping scheme is depicted in Figure 5, where each source operand is assigned to an RC set based on its position in the instruction (the first operand to Set 0, the second to Set 1, and so forth).

2) Destination registers: The cache set mapping of destination registers is based on register indices. Since the register space in GPUs is often defined with an address range of $0-255$, the first mapping approach is to linearly separate the address space based on the number of cache sets, calculated by:

$$SetIndex = \lfloor \frac{(Index\ of\ Reg) \times (\#\ of\ Cache\ Sets)}{256} \rfloor \tag{2}$$

or alternatively, in an interleaving fashion, defined by:

$$SetIndex = (Index\ of\ Reg) \mod (\#\ of\ Cache\ Sets) \tag{3}$$

Both of the mapping schemes can be beneficial depending on the exact register accessing pattern exposed by the workloads.

The interleaving cache set mapping for destination registers is depicted in Figure 5. In this example, the registers associated with the operand $R40$, namely $R40$, $R41$, $R42$, $R43$, are mapped to consecutive sets in the RC because of their sequential index numbers. To clarify, the mapped set number is obtained by dividing the register index by the total number

of cache sets and finding the remainder (e.g., 42 mod 4 = 2). In another example, consider the utilization of the linear mapping approach in the instruction $R150$ $IADD3$ $R1, R2, R3$, which involves a destination register $R150$ and three source register operands $R1$, $R2$, and $R3$. While the source operand registers are mapped to RC entries based on their positions, RC Set 2 is allocated to $R150$ using the following calculation: $\lfloor \frac{150 \times 4}{256} \rfloor = 2$. This study employs a position-based mapping scheme for all workloads. However, both linear and interleaving schemes are adopted to map the destination registers to RC sets in general-purpose workloads. Our focus in MMA-intensive workloads lies on interleaving as the primary destination operand mapping scheme. This preference stems from the frequent occurrence of instructions with consecutive or closely positioned register indexes, which are highly prone to being grouped together in the same RC set by the linear mapping. Considering the small set sizes employed in RC, such grouping could significantly impair performance.

D. Cache Bank Organization

In memory design, a multi-bank organization minimizes bank conflicts, enhancing SRAM array performance and throughput. However, more banks increase power consumption due to additional read/write ports, posing a trade-off in the design. For a SIMT core with 32-lane SIMD execution, a 32-bank RC is conflict-free but energy-intensive. Conversely, a 4/8/16 bank design consumes less power but may face performance penalties due to duplicated port setups.

To mitigate bank conflicts, widening ports by combining neighboring lanes can be effective. For example, an 8-bank design with 128-bit ports per bank eliminates conflicts. However, it is essential to note that SIMD masks may reduce throughput by introducing redundant data fetches.

In this study, we experimented with both the 32-bank with 32-bit bitwidth and 8-bank with 128-bit bitwidth organization schemes. However, given the negligible performance disparity between the two schemes and the typically improved energy efficiency associated with the 8-bank with 128-bit bitwidth organization, this scheme has been selected as the main RC bank organization in our experiments.

E. Cache Replacement and Eviction Policies

A cache replacement policy determines which cache entry to evict when the cache is full. Common age-based strategies, including LRU and FIFO, are extensively employed in the realm of GPUs, offering acceptable performance. In scenarios with a compact RC, we prioritize streamlined control logic, choosing FIFO over LRU for implementation.

When a cache miss occurs, synchronization between the cache (RC) and memory (RF) is crucial for consistency, known as the dirty cache line eviction policy. Eviction policies like write-through and write-back bring different trade-offs. Write-through immediately updates lower-level memory (RF) but increases traffic, while write-back delays updates until cache (RC) eviction, reducing traffic but risking data inconsistency. Our RC implementations leverage write-back for its lower transaction and energy overhead, ensuring dirty cache lines are evicted only during replacement.

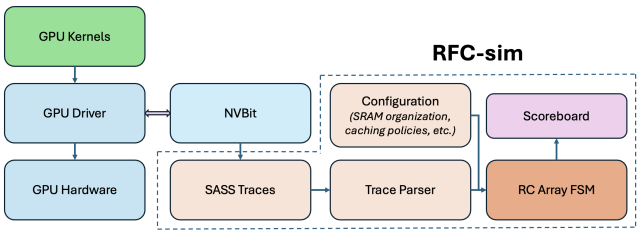


Fig. 6. RFC-sim top-level diagram.

V. RFC-Sim Simulator

In this study, we have developed an analytical simulator called *RFC-sim* to model RC design based on the behavior requirements outlined in Chapter IV. *RFC-sim* can simulate the behavior of the RC when executing GPU workloads by utilizing a predefined cache configuration file and instruction traces generated during GPU kernel execution. It also imports cache model statistics and additional profiling information to enhance its accuracy and fidelity in modeling RC behavior. The software design of *RFC-sim* is both scalable and flexible. This simulator supports workloads running on NVIDIA GPU architectures, including Volta (SM70), Turing (SM75), and Ampere (SM80). This work utilizes Turing architecture.

As illustrated in Figure 6, RFC-sim digests the SASS instruction traces, models the behavior of the RC and RF in GPUs, and reports statistical metrics like cache performance and energy consumption to the scoreboard. SASS traces as the input to the simulator are generated by NVBit, an instrumentation tool interacting with the GPU driver used to track and dump the warp instructions inside GPU kernels that are executed by the GPU hardware. SASS traces are parsed by a trace parser and then delivered to a FSM (Finite-State Machine) [27] modeling the RC array, whose state transition logic is determined by both the input traces and the model configuration including the SRAM bank organization, caching policies, etc. The simulator continuously updates statistics to the scoreboard during the simulation process. As depicted in Figure 7a, the instruction traces exhibit instruction information including program counters, SIMD masks, the instruction and a bit flag before each SASS instruction indicates whether there is a destination register. RFC-sim sequentially parses the SASS traces, instruction by instruction, and translates them into an internal data structure (based on C++), shown in Figure 7b, that RFC-sim can comprehend. The parsed instructions consequently determine the operation flow of the RFC-sim.

A. RFC-Sim Execution Model

RFC-sim models the register cache array as a FSM, where the states encode the register indices being cached along with their associated control information. The workload, shown in Figure 7a, is defined as a sequence of warp instructions executed by the SM cores, and is the input to the FSM. The FSM transits to the next state on the execution of a warp-level SASS instruction with the control logic based on:

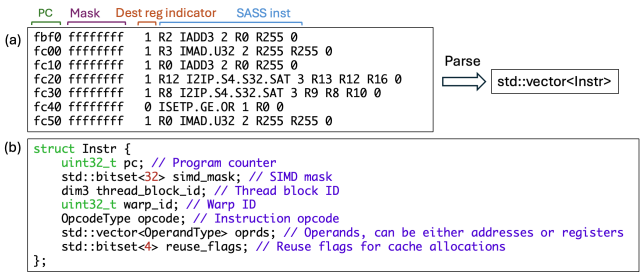


Fig. 7. (a) SASS traces; (b) the parsed data structure in RFC-sim (pseudo C++ codes).

- **SIMD mask of the input warp instruction:** the SIMD mask of a warp instruction indicates the divergence of the threads inside the warp. Since each thread has its private GPRs, the SIMD mask determines the register bank access behavior of register reads/writes in the operand collection/write-back stages of a warp instruction. The mask values are outside the scope of this paper.
- **Register operands within the input warp instruction:** The state transition logic follows the same structure as the SM processing block pipeline. The FSM initially looks at the source register operands during the operand collection stage [28], moves to a temporary state based on the source operands, and then examines the destination register accessed during the write-back stage. This determines the FSM state after executing the warp instruction.
- **Caching policy of RC:** This includes associativity, allocation, mapping, replacement, and eviction policies, which have been thoroughly discussed in Section IV. Software-based allocation policies, such as the compiler-aided policy controlled by the reuse flag, employed in Volta and Turing architectures, dictate the control logic along with additional information in the input instruction. Since reuse flags are not shown in NVBit-generated traces, we obtain them by statically disassembling encoded SASS instructions via *cuobjdump* [29], a command-line utility provided by NVIDIA, and then attach the flags to the parsed instructions.

B. Scoreboard Statistical Report

Throughout the simulation process, a scoreboard collects events such as cache hits/misses, RC/RF bank port accesses, and other data, generating a statistical report of the simulation. The RC model and its state transition logic (as implied by the green dashed arrow) are illustrated in Figure 8. The yellow box on the top-left presents an 8-bank RC model, where each bank has 3 read ports and 1 write port, and handles requests from 4 threads. The model depicts a 4-way associative cache with 8 entries for each thread. On the execution of a SASS instruction, each thread independently conducts caching logic. In this example, an incoming SASS instruction activates the first 4 threads based on its SIMD mask to make the R/W ports of bank 0 respond. Assuming compiler-aided allocation and interleaving cache set mapping for destination register operands, *R1* reserves a cache entry in cache set 0 and *R5* reserves a cache entry in cache set 1. If *R8* and *R12* are the data to be replaced according to the replacement logic

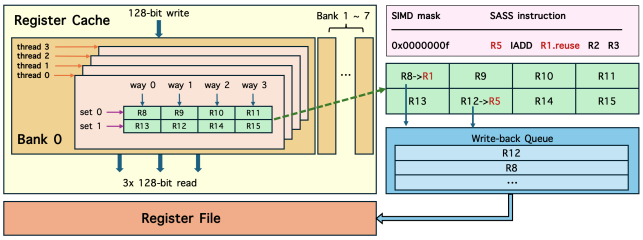


Fig. 8. RC model incorporated into RFC-sim.

TABLE III
RF AND RC ENERGY MODEL PARAMETERS

Cache and Associativity	Bitwidth	Read (pJ)	Write (pJ)
RF	32	16.3764	15.2452
2-way RC	128	23.4685	24.2801
4-way RC	128	35.3369	36.7010
8-way (fully) RC	128	43.2275	44.0041

(assumption specific to this example), they are written back to the RF. During instruction execution, the scoreboard records cache misses for *R1, R2, R3*, and *R5* in RC, along with the associated RC/RF read/write accesses, cache allocations, and data evictions. For instance, cache allocation for *R1* results in a write access to RC bank 0, and the eviction of *R8* results in a write access to the RF bank write port. It is noteworthy that RFC-sim operates based on each thread having its private register cache, thus unconsidered GPU dynamic characteristics like warp scheduling do not affect reported events.

C. Energy Model

RFC-sim can be integrated with an energy model to deliver dynamic energy estimation results derived from its extracted behavior statistics, a feature utilized in this study. Simulated using CACTI 7.0 [30], a power analysis tool for estimating power consumption in integrated circuits, the dynamic energy model of the RF and RC used in the experiment is shown in Table III. The memory is modeled using the 22 nm technology provided by CACTI. Since our model considers data movement (locality) optimization, the reported benefits remain valid across other technologies. RC power models include 2/4/8-way fully associative cache designs.

In the best cases, i.e., all masks are set to on, the energy consumption of 128-bit accesses to the fully associative cache could be considered equivalent to 4 32-bit accesses, resulting in the per 32-bit read energy of 10.8069pJ and the per 32-bit write energy of 11.001pJ. With the benefit originating from the larger locality captured by the fully associative cache against the set associative cache, it is possible to have the fully associative cache outperforming the set associative cache regarding energy efficiency. The energy models show that the trade-off lies between the locality that the register cache can capture and the energy cost of accessing the cache entries.

VI. EVALUATION

In this section, we present a comprehensive overview of the utilized RC model, followed by a thorough assessment across diverse parameters. Our evaluation encompasses two types of

TABLE IV
RODINIA 3.1 BENCHMARKS

Benchmark	Abbr.	Domain
Breadth-First Search	bfs	Graph Algorithms
B+ tree	b+tree	Search
Gaussian Elimination	gaussian	Linear Algebra
Hotspot	hotspot	Physics Simulation
K-means	kmeans	Data Mining
Nearest Neighbor	nn	Pattern Recognition
LU Decomposition	lud	Linear Algebra
Path Finder	pathfinder	Medical Imaging

TABLE V
SQUARED GEMM BENCHMARKS

ID	Squared GEMM Size	Data Format
HGEMM-256	256	half-precision
HGEMM-512	512	half-precision
HGEMM-1024	1024	half-precision
HGEMM-2048	2048	half-precision
IGEMM-256	256	INT8
IGEMM-512	512	INT8
IGEMM-1024	1024	INT8
IGEMM-2048	2048	INT8

benchmarks executed on CUDA cores and TC. Additionally, we delve into different RF traffic analyses, elucidating their respective impacts on overall energy consumption.

A. Benchmarks

The workloads evaluated in the experiment fall into 2 categories: 1) general-purpose workloads (executed by CUDA cores), and 2) MMA-intensive workloads (executed by TCs). For the general-purpose workloads, 8 benchmarks are selected from Rodinia 3.1 benchmark suite [26], as shown in Table IV. For MMA-intensive workloads, the squared GEMM with 2 different data formats, i.e., half-precision (FP16) and INT8, are considered. All GEMM kernels are implemented via CUTLASS 3.2 [2] and CUDA 11.5 compiled into SM75 [10] assembly, with the underlying SASS opcode of *HMMA.1688.FP32* and *IMMA.8816.S8.S8.SAT* for Half-precision GEMM (HGEMM) and INT8 GEMM (IGEMM), respectively. A list of the evaluated GEMM kernels is presented in Table V.

B. Register File Model

Based on our speculations of the Turing (sm75) architecture, the RF in the experiment is modeled as a 2-bank register file with 64 KB capacity per processing block (256 KB and 8 banks per SM), where each bank is equipped with dual ports with 32-bit bitwidth. As for RC, we evaluate an 8-bank organization where each port is 128-bit wide, the data capacity is fixed to 8 KB (8 warps, 32 threads per warp, 8 entries per thread, and 4 bytes of data per entry). This setup is flexible regarding the adaptation of associativity and cache set mapping. It enables us to implement a 2-way associative RC for ISA, allowing four source operand positions, as well as 4/8-way fully associative designs. Larger RC designs (such as 16 entries per thread) are not presented in this work due to their assessed energy inefficiencies.

TABLE VI
CACHE MODELS

Associativity	Allocation	Dest. mapping
8-way (fully)	write-allocate	N/A
8-way (fully)	compiler-aided	N/A
4-way	write-allocate	linear
4-way	write-allocate	interleaving
2-way	write-allocate	linear
2-way	write-allocate	interleaving
2-way	compiler-aided	linear
2-way	compiler-aided	interleaving

As shown in Table VI, in our examinations, we consider RCs with 2-way, 4-way, and 8-way (fully-associative) associativity. As discussed earlier, the adopted compiler-aided allocation policy mirrors the write-allocate policy concerning destination registers in RCs. Meanwhile, the compiler-aided allocation policy also leverages compiler-attached reuse flags to exploit the temporal locality of source registers in its strategy. To discern the advantages of employing compiler-aided allocation, we evaluate both write-allocate and compiler-aided allocation approaches. The mapping of cache sets for destination operands can follow either a linear or interleaving approach. Both methods are analyzed for general-purpose workloads, while only the interleaving approach is evaluated for MMA-intensive workloads in this study. Our results are compared to the baseline RF without RC utilization.

C. Evaluation Parameters

This section describes our evaluation metrics including:

- **RC Read-hit rate or RF traffic reduction**. The read traffic surpasses the write traffic owing to the sheer number of source operands compared to the destination operands. Therefore, the read-hit rate stands out as the most essential metric determining the performance and energy efficiency of RC. The source operand register data has to be read first from RC, hence higher RC read-hit rate serves as a direct indicator of decreased traffic on RF's read ports. The reduced traffic from RF read ports is numerically identical to the RC's read-hit rate;
- **RC write-hit rate and RF write traffic reduction:** RC write hits, on the other hand, is not numerically identical to the reduction of RF writes. Given the RC's utilization of the write-back eviction policy, RF write traffic accounts for the instances when RC entries are evicted and written back to the RF. Consequently, a decrease in RF write traffic occurs only when written data in RC undergoes multiple writes before eviction and subsequent writing back. Without data reuse, a RC employing a write-back eviction policy behaves similarly to a write-through approach, and the write-hit rate does not contribute to reducing the total number of RF writes. This reduction signals that the dirty data is actively updated within the RC instead of resorting to frequent access of the RF;
- **Energy reduction:** This metric represents the energy savings attained through the implementation of RC, as opposed to utilizing RF alone. To deliver optimal energy efficiency, the RC should capture the locality of workloads, cache the data, and reduce traffic to RF.

D. General-Purpose Workloads

This section presents the examination results of the general-purpose workloads executed on CUDA cores, including analyses of RF traffic and power consumption. Each figure illustrates the obtained results over one of our evaluation metrics, assessed using various benchmarks and RC configurations mentioned in Table VI.

1) RC read-hit rate or RF traffic reduction: The Figure 9 illustrates the RC read hit rate, i.e., RF traffic reduction, of the eight benchmarks selected from Rodinia 3.1. These benchmarks are assessed across various RC designs, including an 8-way (fully) associative RC based on write-allocate policy and other 2/4/8-way associative RCs with linear or interleaving destination mapping using compiler-aided allocation. Notably, fully associative caches do not require destination mapping policies as there are no sets to choose from. Several observations can be made from the results:

- **Associativity:** It is noted that the locality increases with associativity, which is reasonable given the larger sets' ability to capture locality. This trend is apparent across all benchmarks, with 8-way (fully) associative RC demonstrating the highest success in capturing data locality, consequently boosting the RC read-hit rate.
- **Compiler-aided vs write-allocate:** Another observation from the figure is that the compiler-aided allocation does not improve the cache performance on the evaluated benchmarks compared to the 8-way write-allocate RC design. In the best case, compiler-attached reuse flags only deliver 1.3% of RC read-hit rate improvement in *b+tree* while utilizing an 8-way (fully) associative cache. Observed from the codes, the compiler barely tends to attach reuse flags for compiler-aided allocation policy to take advantage of in the 8 Rodinia benchmarks. This observation consistently emerges across other evaluation metrics for general-purpose workloads as well.
- **Interleaving vs linear mapping:** The interleaving mapping outperforms the linear mapping in a 2-way associative cache in most of the cases except *bfs*. Since *bfs* is usually considered an irregular workload, interleaving mapping enhances the capture of temporal locality by placing neighboring registers in different cache sets. This configuration enables more registers to reside in the RC during consecutive instructions, increasing the likelihood of them being accessed again. However, the situation is different in a 4-way associative cache, where the performance of linear and interleaving mapping largely depends on the specific workloads. For instance, in the case of *lud* and *pathfinder*, interleaving mapping outperforms linear mapping when using a 4-way associative cache while it is the opposite for a 2-way associative cache.

2) RC write-hit rate and the RF write traffic reduction: The RC write-hit rate and the RF write traffic reduction are presented in Figure 10a and b, respectively, with RC configurations similar to those discussed in the previous section. Upon our examination, we can draw several observations:

- **Associativity:** As opposed to RC read-hit rate, higher associativity does not always result in a higher RC write-hit rate. This phenomenon arises from the constraint that CUDA core instructions typically have one destination register. Hence, increasing RC set associativity does not contribute as much to reducing RC write-hit rate and consequently, RF traffic reduction.
- **Interleaving vs linear mapping:** Interleaving mapping largely outperforms linear mapping regarding exploiting the temporal locality among the destination registers. Furthermore, in the majority of benchmarks, the interleaving mapping in the 2-way set associative RC exhibits superior performance compared to the 4-way RC, underscoring the significance of destination register mapping in reducing RF write traffic compared to associativity. Furthermore, the operands involved in an instruction are often located close to each other, enabling them to be assigned to different cache sets. This facilitates more efficient utilization of the cache space in the interleaving mapping.

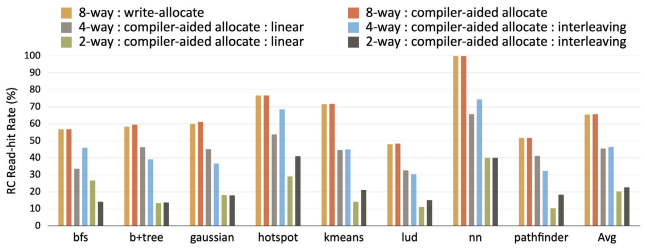
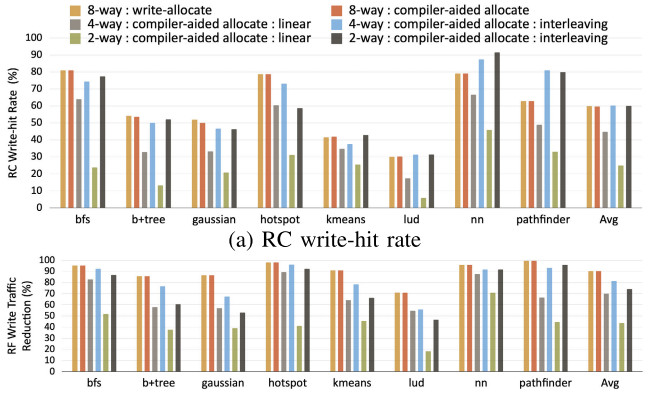


Fig. 9. RC read-hit rate or RF traffic reduction in general-purpose workloads.



(a) RC write-hit rate

(b) RF write traffic reduction

Fig. 10. (a) RC write-hit rate; and (b) RF write traffic reduction in general-purpose workloads.

3) Energy consumption: The characterization of energy reduction is shown in Figure 11 when RC is utilized. As can be seen in the figure with the addition of RC, the dynamic energy consumed on RF alone can be reduced by up to 40% on the evaluated Rodinia 3.1 benchmarks. While the energy savings of RC schemes heavily depend on the workloads, the following observations can be made:

- **Associativity:** Higher set associativity, as detailed in Table III, does not always result in more energy savings due to the overhead of complex searching algorithms in caches with higher set associativity.
- **Interleaving vs linear mapping:** The interleaving mapping has proven to be more beneficial than the linear mapping in 2/4-way associative RCs, and this pattern has a correlation with RF write traffic reduction.

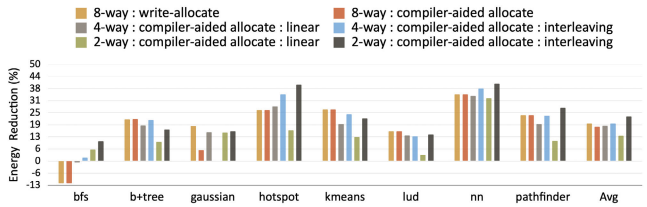


Fig. 11. RC and RF total register energy reduction in general-purpose workloads.

Generally, a well-designed RC is capable of saving significant energy consumption and mitigating the traffic in/out of RF by capturing the temporal locality among the registers.

E. MMA-Intensive Workloads

This section presents the examination results of the MMA-intensive workloads executed on TCs including analyses of RF traffic and power consumption. The register accessing pattern in the MMA-intensive workloads is intrinsically distinct from the general-purpose workloads. The extensive operand traffic generated by the Half-precision GEMM (HGEMM) operations implemented via the *HMMA.1688.FP32* operation presents challenges for RC to achieve optimal performance and energy efficiency when compared to the *IMMA.8816.S8.S8.SAT* operation. This variation in performance likely stems from layout differences between the two operations, which is beyond the scope of this study. These two operations are evaluated across different RC designs, encompassing 2/4/8-way (fully) associative RC designs equipped with either write-allocate or compiler-aided allocation policies. The evaluation spans GEMM sizes ranging from 256 to 2048. The characterization results on HGEMM and IGEMM are described in this section.

1) RC read hit rate or RF traffic reduction: Figure 12 illustrates the RC read-hit rate, which leads to reduced read traffic in the RF, evaluated using HGEMM and IGEMM operations. Several observations can be drawn based on our obtained results:

- **Associativity:** As demonstrated in both Figure 12a and b, depicting the RC read-hit rate for both the HGEMM and IGEMM operations, respectively, it is evident that increasing associativity consistently leads to a higher RC read-hit rate. This observation is reasonable given the sheer number of operands in MMA instructions.
- **Compiler-aided vs write-allocate:** Compiler-aided allocation exhibits a significant improvement in the RC read-hit rate (i.e., RF read traffic reduction) of approximately 6% in HGEMM and 20% in IGEMM on average, compared to the write-allocate policy. This underscores the potential of the compiler-aided allocation policy in capturing locality when used with MMA operations due to the large number and high locality of source operand registers, which are ignored in the write-allocate policy.
- **HGEMM vs IGEMM:** RC demonstrates greater success in capturing locality within IGEMM workloads, achieving a substantial RC read-hit rate i.e. reduction of RF traffic of up to 26%. On the other hand, HGEMM can achieve a 12% RC read-hit rate with the adoption of compiler-aided design, which does not match the results obtained from the IGEMM operation.

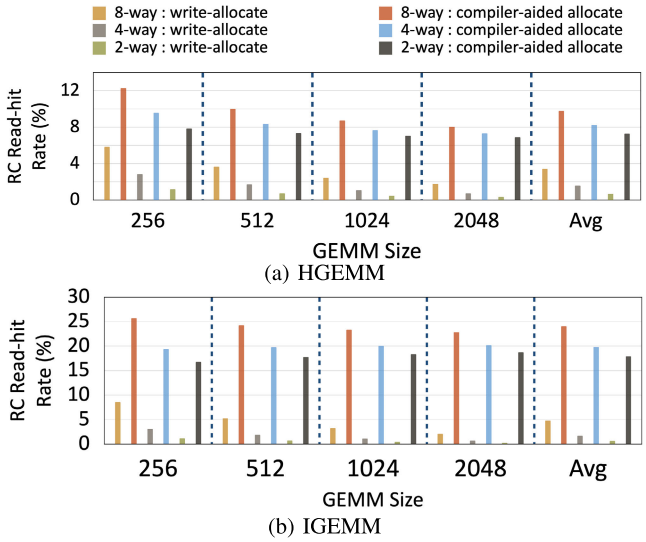


Fig. 12. RC read-hit rate or RF traffic reduction for (a) HGEMM; and (b) IGEMM operations in MMA-intensive workloads.

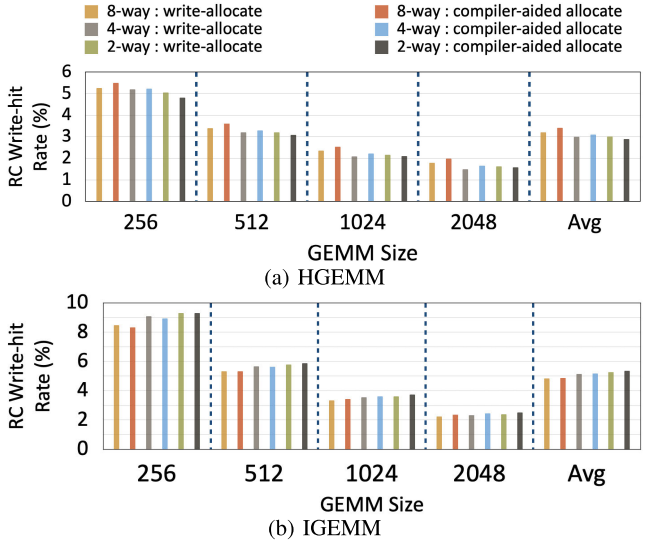


Fig. 13. RC write-hit rate for (a) HGEMM; and (b) IGEMM operations in MMA-intensive workloads.

2) RC write-hit rate and the RF write traffic reduction: Figures 13 shows the RC write-hit rate and Figure 14 shows the reduced RF write traffic, for both HGEMM and IGEMM, respectively. The following observations can be made based on the results.

- **Associativity:** As depicted in Figure 13, increasing RC set-associativity leads to a slightly better RC write-hit rate in HGEMM, while IGEMM illustrates the opposite effect. This could be attributed to variations in the register layouts of the HGEMM and IGEMM instructions. Nevertheless, the discrepancies in the RC write-hit rate are consistently minimal across different configurations. However, higher associativity, as shown in Figure 14, can significantly reduce RF traffic. Our findings demonstrate that an 8-way (fully) associative RC configuration consistently outperforms other configurations in this aspect.

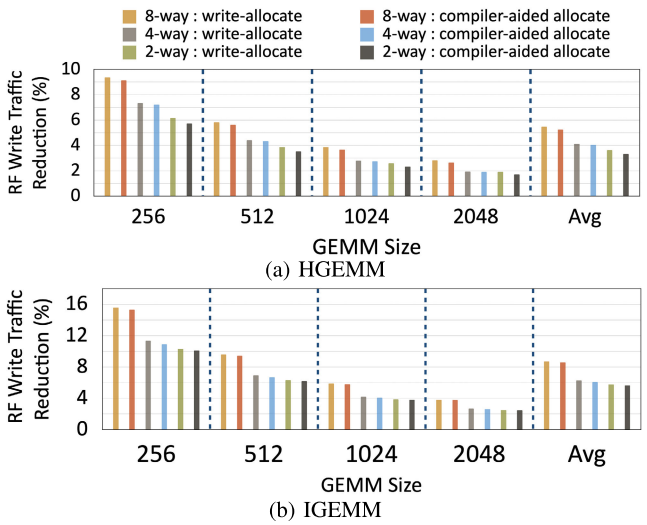


Fig. 14. RF write traffic reduction for (a) HGEMM; and (b) IGEMM operations in MMA-intensive workloads.

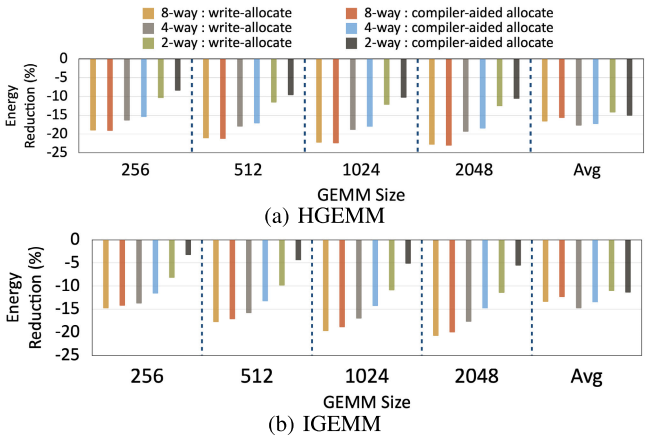


Fig. 15. RF energy reduction for (a) HGEMM; and (b) IGEMM operations in MMA-intensive workloads.

- **Compiler-aided vs write-allocate:** The compiler-aided design is generally unable to outperform the write-allocate policy in mitigating RF write traffic in the majority of cases. This is due to the fact that RF write traffic comprises the eviction of dirty RC entries, which are unaffected by the compiler-attached reuse flags used in the compiler-aided allocation policy.
- **HGEMM vs IGEMM:** In comparison to HGEMM, the performance of RC is slightly improved in IGEMM, exhibiting a slightly higher RC write-hit rate and reducing RF write traffic by up to 6% more than HGEMM, and by up to 16% in total.

3) Energy consumption: Figure 15a and b shows the total RF energy reduction for both HGEMM and IGEMM operations, respectively. Despite the previously reported reduced RF read and write traffic, none of the RC schemes managed to achieve energy savings and could instead incur an energy overhead of up to 23%. However, employing the optimal RC configuration has the potential to minimize energy consumption overhead. The following observations arise from the obtained results:

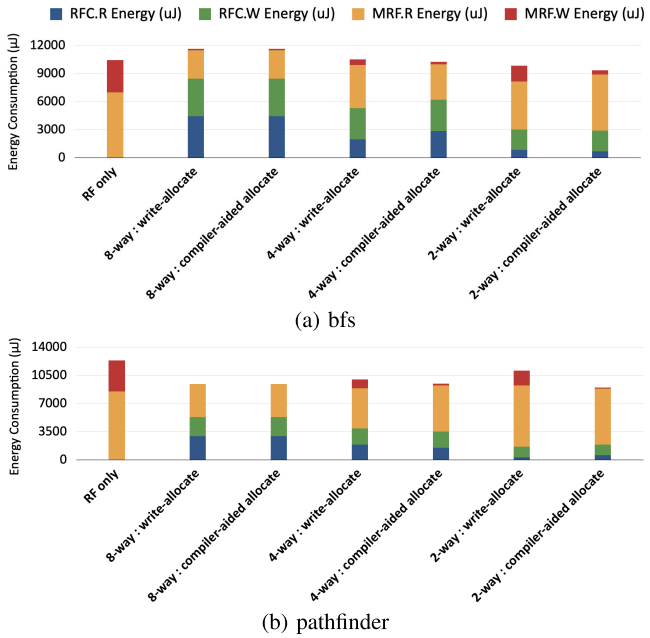
- **Associativity:** As evidenced in both HGEMM and IGEMM results, reducing the associativity can consistently reduces RC energy consumption overhead. As detailed in Table III, RCs with lower associativity consume less energy per read and write, leading to overall energy savings in MMA-intensive workloads.
- **Compiler-aided vs write-allocate:** Compiler-aided allocation can notably cut energy consumption overhead, especially in 2-way set associative RCs, compared to the write-allocate policy. This is due to the substantial decrease in RF read traffic combined with the low energy consumption of 2-way set associative RCs.
- **HGEMM vs IGEMM:** RC utilization leads to an average energy consumption overhead of 16% and 13% for HGEMM and IGEMM operations, respectively, with minimum values of 8% and 3%. These findings indicate that IGEMM operations tend to be slightly more energy-efficient than HGEMM operations.

F. Energy Breakdown

In this section, we investigate the register energy breakdown before and after integrating RC into the SM processing blocks of the GPU. Two benchmarks, *bfs* and *pathfinder*, are chosen to represent general-purpose workloads, and both HGEMM and IGEMM operations are analyzed as representatives of MMA-intensive workloads in this context. We aim to discuss the energy improvements of RC in general-purpose workloads and explore the causes of energy consumption overhead when MMA-intensive workloads are utilized.

1) General-purpose Workloads: Figure 16a and b display the RF and RC energy consumption breakdown of the *bfs* and *pathfinder* benchmarks, respectively. Here, the *RC.R* and *RC.W*, similar to RF, denote the energy consumed by RC read and write operations, respectively. The first bar from the left in the figure represents the energy consumption of RF without RC integration. Subsequent bars depict RC configurations, including 2/4/8-way (fully) associative RC designs equipped with either write-allocate or compiler-aided allocation policies. All RC configurations here follow the interleaving destination mapping policy due to its superior performance. As evident in the results, most RC configurations exhibit substantial energy consumption savings. Noteworthy is the 2-way RC with compiler-aided allocation, achieving up to 15% and 28% savings for bfs and pathfinder benchmarks, respectively. RC is proven energy-efficient when its overhead is outweighed by energy saved from fewer RF reads and writes. Our experiments show RC excels in capturing write locality, nearly eliminating RF writes in some configurations and significantly reducing them in others for general-purpose workloads.

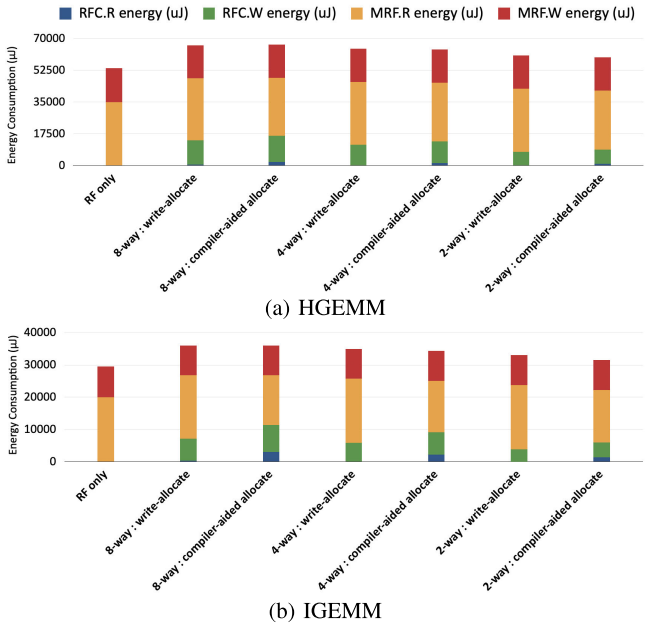
2) MMA-intensive Workloads: Figure 17a and b present the RF and RC energy consumption breakdown of MMA-intensive workloads, specifically HGEMM and IGEMM operations, respectively. As discussed earlier, in MMA workloads, RC lacks energy efficiency and struggles to capture data locality. The 8-way RC incurs the highest energy overhead, adding 23% and 21% to total energy



(a) bfs

(b) pathfinder

Fig. 16. RF and RC total energy breakdown in general-purpose workloads including (a) bfs and (b) pathfinder.



(a) HGEMM

(b) IGEMM

Fig. 17. RF and RC total energy breakdown in MMA-intensive workloads including (a) HGEMM and (b) IGEMM.

consumption in HGEMM and IGEMM operations, respectively. RC's difficulty in capturing read locality compared to general-purpose workloads is evident from the energy breakdown. Furthermore, the captured RC writes do not lead to a significant reduction in RF writes, resulting in substantial energy consumption overhead. This can be attributed to the large number of destination registers in MMA instructions relative to the small capacity of the RC. For each MMA instruction, multiple destination registers must be written to, triggering additional energy-consuming RC allocations. Besides allocating RC entries for result matrices, compiler-aided allocation results in more RC writes.

VII. Discussion

While RC showed promise in optimizing traffic and improving the energy efficiency of the RF when utilized with general-purpose workloads, RC fails to achieve this when MMA-intensive workloads are utilized. Our findings suggest that this limitation arises from the substantial number of source and destination registers involved in MMA instructions, coupled with the constrained size of RC. While allocating more space to RC could potentially address this issue, such a solution is impractical due to its high cost in terms of hardware resources and area, as well as its lack of scalability and energy efficiency.

Another challenge posed by MMA-intensive workloads is the inefficient use of RC to store destination registers. We propose compiler-based solutions for smarter destination register allocation policies, rather than relying on the simplistic write-allocate policy. Additionally, optimizing MMA instruction execution by reordering them to leverage register temporal locality could be beneficial.

An alternative approach to handling inefficiencies in MMA-intensive workloads is bypassing the RC mechanism for TC-executed instructions. This eliminates RC energy consumption overhead, allowing repurposing of allocated RC space for extending the main RF or other SM memory units like data cache or shared memory. This solution could yield significant improvements in performance and energy efficiency for MMA-intensive tasks.

VIII. Conclusion

Register Cache (RC) has demonstrated success in enhancing energy efficiency within CUDA cores and has become a standard feature in recent GPU architectures; however, its impact on Tensor Cores (TCs) remains relatively unexplored in existing literature. In this study, we conducted a comprehensive investigation into various configurations of register cache, encompassing associativity, allocation, and mapping. Subsequently, we explored its integration into the GPU Streaming Multiprocessors (SM) and assessed its performance using both general-purpose workloads executed on CUDA cores and Matrix Multiply and Accumulate (MMA)-intensive workloads executed on TCs. To accurately measure power measurements, we developed an analytical simulator named RFC-sim. Our findings shed light on the inefficiencies associated with employing RC for TC workloads. The power consumption can be increased by up to 23% when a fully-associate RC cache is utilized. We ultimately proposed suggestions aimed at enhancing the efficiency of RC in TC workloads.

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, no. 6, pp. 84–90, May 2017, doi: 10.1145/3065386.

[2] S. Markidis, S. W. D. Chien, E. Laure, I. B. Peng, and J. S. Vetter, "NVIDIA tensor core programmability, performance precision," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, May 2018, pp. 522–531.

[3] M. Dayarathna, Y. Wen, and R. Fan, "Data center energy consumption modeling: A survey," *IEEE Commun. Surveys Tuts.*, vol. 18, no. 1, pp. 732–794, 1st Quart., 2016.

[4] J. Leng et al., "GPUWattch: Enabling energy optimizations in GPGPUs," in *Proc. 40th Annu. Int. Symp. Comput. Architecture* New York, NY, USA: Association for Computing Machinery, Jun. 2013, pp. 487–498, doi: 10.1145/2485922.2485964.

[5] J.-L. Cruz, A. Gonzalez, M. Valero, and N. P. Topham, "Multiple-banked register file architectures," in *Proc. 27th Int. Symp. Comput. Architecture*, Feb. 2000, pp. 316–325.

[6] M. Gebhart et al., "Energy-efficient mechanisms for managing thread context in throughput processors," in *Proc. 38th Annu. Int. Symp. Comput. Architecture (ISCA)*. New York, NY, USA: Association for Computing Machinery, Jun. 2011, pp. 235–246, doi: 10.1145/2000064.2000093.

[7] Z. Jia, M. Maggioni, J. Smith, and D. P. Scarpazza, "Dissecting the NVIDIA Turing T4 GPU via microbenchmarking," 2019, *arXiv:1903.07486*.

[8] NVIDIA. (2009). *Fermi GPU Architecture Whitepaper*. Accessed: Mar. 4, 2024. [Online]. Available: https://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf

[9] NVIDIA. (2017). *NVIDIA Tesla V100 GPU Architecture*. Accessed: Mar. 4, 2024. [Online]. Available: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[10] NVIDIA. (2018). *Turing GPU Architecture Whitepaper*. Accessed: Mar. 4, 2024. [Online]. Available: https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

[11] M. A. Raihan, N. Goli, and T. Aamodt, "Modeling deep learning accelerator enabled GPUs," 2018, *arXiv:1811.08309*.

[12] A. Akram and L. Sawalha, "A survey of computer architecture simulation techniques and tools," *IEEE Access*, vol. 7, pp. 78120–78145, 2019.

[13] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, Oct. 2009, pp. 163–174.

[14] J. Lew et al., "Analyzing machine learning workloads using a detailed GPU simulator," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Mar. 2019, pp. 151–152.

[15] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-Sim: An extensible simulation framework for validated GPU modeling," in *Proc. ACM/IEEE 47th Annu. Int. Symp. Comput. Architecture (ISCA)*, May 2020, pp. 473–486.

[16] O. Villa, M. Stephenson, D. Nellans, and S. W. Keckler, "NVBit: A dynamic binary instrumentation framework for NVIDIA GPUs," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2019, pp. 372–383, doi: 10.1145/3352460.3358307.

[17] V. Kandiah et al., "Accelwattch: A power modeling framework for modern GPUs," in *Proc. 54th Annu. IEEE/ACM Int. Symp. Microarchitecture*. New York, NY, USA: Association for Computing Machinery, Oct. 2021, pp. 738–753, doi: 10.1145/3466752.3480063.

[18] M. Gebhart, S. W. Keckler, and W. J. Dally, "A compile-time managed multi-level register file hierarchy," in *Proc. 44th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Dec. 2011, pp. 465–476.

[19] H. Jeon, G. S. Ravi, N. S. Kim, and M. Annavaram, "GPU register file virtualization," in *Proc. 48th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Dec. 2015, pp. 420–432.

[20] F. Khorasani, H. A. Esfeden, A. Farmahini-Farahani, N. Jayasena, and V. Sarkar, "RegMutex: Inter-warp GPU register time-sharing," in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Architecture (ISCA)*, Jun. 2018, pp. 816–828.

[21] M. Abdel-Majeed, A. Shafaei, H. Jeon, M. Pedram, and M. Annavaram, "Pilot register file: Energy efficient partitioned register file for GPUs," in *Proc. IEEE Int. Symp. High Perform. Comput. Architecture (HPCA)*, Feb. 2017, pp. 589–600.

[22] J. Bailey, J. Kloosterman, and S. Mahlke, "Scratch that (but cache this): A hybrid register cache/scratchpad for GPUs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 11, pp. 2779–2789, Nov. 2018.

[23] M. Sadrosadati et al., "Highly concurrent latency-tolerant register files for GPUs," *ACM Trans. Comput. Syst.*, vol. 37, nos. 1–4, Jan. 2021, doi: 10.1145/3419973.

[24] X. Guan, H. Zhou, G. Bao, H. Li, L. Zhu, and J. Yao, "PresCount: Effective register allocation for bank conflict reduction," in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim. (CGO)*, Mar. 2024, pp. 170–181, doi: 10.1109/cgo57630.2024.10444841.

[25] M. A. Shoushtary, J. M. Arnau, J. T. Murgadas, and A. Gonzalez, "Lightweight register file caching in collector units for GPUs," in *Proc. 15th Workshop Gen. Purpose Process. Using GPU*. New York, NY, USA: Association for Computing Machinery, Feb. 2023, pp. 27–33, doi: 10.1145/3589236.3589245.

[26] S. Che et al., "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Oct. 2009, pp. 44–54.

[27] G. H. Mealy, "A method for synthesizing sequential circuits," *Bell Syst. Tech. J.*, vol. 34, no. 5, pp. 1045–1079, Sep. 1955.

[28] S. Liu, J. E. Lindholm, M. Y. Siu, B. W. Coon, and S. F. Oberman, "Operand collector architecture," U.S. Patent 7 834 881 B2, Nov. 2010. Accessed: Mar. 4, 2024.

[29] (Nov. 2023). *Cuda-Binary-Utilities Release 12.3*. Accessed: Mar. 4, 2024. [Online]. Available: https://docs.nvidia.com/cuda/pdf/CUDA_Binary_Utilities.pdf

[30] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0," in *Proc. 40th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Dec. 2007, pp. 3–14.

Vahid Geraeinejad received the bachelor's degree from the Sharif University of Technology, Iran, in 2016, and the master's degree from the University of Tehran in 2020. He is currently pursuing the Ph.D. degree with the KTH Royal Institute of Technology. He is currently a Researcher. His academic journey further extended through a full-year joint International Credit Mobility (ICM) Program, Mlardalen University, where he collaborated with the Division of Intelligent Future Technologies. With a profound interest in GPU optimization, his research focus lies in exploring innovative approaches to enhance the performance and efficiency of GPGPUs.

Qiran Qian received the B.S. degree from Harbin Institute of Technology, Shenzhen, in 2022, and the M.S. degree in computer science and engineering from the KTH Royal Institute of Technology in 2024. His master's thesis delving into the intricacies of optimizing the energy efficiency of register caching in GPGPUs. He is currently a Machine Learning Performance Engineer with WizardQuant. His research interests include computer architecture, high-performance computing, and machine learning systems.

Masoumeh (Azin) Ebrahimi (Senior Member, IEEE) received the Ph.D. degree (Hons.) from the University of Turku, Finland, in 2013, and the M.B.A. degree in 2015. She is currently an Associate Professor with the KTH Royal Institute of Technology, Sweden. She has led several national and international projects, such as MarieCurie-Vinnova, Academy of Finland, SSF, STINT, Vetenskapsrdet (VR), and WASP. She is the co-author of around 120 publications. Her research interests include on-chip/off-chip interconnection networks, AI/ML on hardware, neural network accelerators, and GPU architecture and optimization. She is a member of Digital Future and HiPEAC.