Journal of Systems Architecture 149 (2024) 103099

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Conflict-aware compiler for hierarchical register file on GPUs

Eunbi Jeong [a], Eun Seong Park [a], Gunjae Koo [c], Yunho Oh [b], Myung Kuk Yoon [a]

[a] *Department of Computer Science and Engineering, Ewha Womans University, Seoul, 03760, Republic of Korea*
[b] *School of Electrical Engineering, Korea University, Seoul, 02841, Republic of Korea*
[c] *Department of Computer Science and Engineering, Korea University, Seoul, 02841, Republic of Korea*

ARTICLE INFO

Keywords:
Graphics processing units
Hierarchical register files
Compiler optimization
Non-volatile memory

ABSTRACT

Modern graphics processing units (GPUs) leverage a high degree of thread-level parallelism, necessitating large-sized register files for storing numerous thread contexts. To reduce the energy consumption in traditional static random access memory (SRAM)-based register files, recent research has explored non-volatile memory (NVM) for implementing register files. The hierarchical register file (HI-RF) combines SRAM-based register caches with NVM-based register files. In HI-RF, the register cache acts as a write buffer, indexed using both register IDs and warp IDs. HI-RF uses a direct-mapped register cache with two indexing schemes: a *concatenating* scheme and a *thread context-aware* scheme. Compiler-assigned register IDs significantly impact cache conflicts, particularly among registers sharing the same LSBs. To address this, we introduce a conflict-aware compiler (CAC) for GPUs equipped with HI-RF. CAC optimizes register assignments based on approximated register write counts. Our evaluation demonstrates that CAC improves performance by 11.1% and 5.9% with the concatenating and thread context-aware index schemes, respectively when compared to a conventional compiler. Simultaneously, it reduces the energy consumption by approximately 73.0 percentage points compared to SRAM for both indexing schemes.

1. Introduction

In modern computing systems, graphics processing units (GPUs) have evolved into indispensable components due to their remarkable capacity to handle high-throughput applications, including video processing, scientific simulations, and deep learning tasks [1–3]. GPUs demonstrate exceptional proficiency by exploiting thread-level parallelism (TLP), facilitating the concurrent execution of numerous threads. To manage the extensive contexts associated with this high degree of TLP, GPUs require significantly large-sized register files. Several prior studies revealed that these register files are the most energy-consuming components within GPUs [4,5].

To overcome this limitation, the replacement of static random access memory (SRAM)-based register files with energy-efficient non-volatile memory (NVM)-based register files has garnered significant attention in prior studies [6–9]. However, due to critical issues such as high energy consumption for writes, high write latency, and limited write counts of NVM, it is impractical to use NVM-based register files as the sole register files on GPUs. As a result, a recent study proposes adopting a hierarchical register file (HI-RF) structure that combines an SRAM-based register cache and an NVM-based register file [10]. Considering that GPU applications often exhibit substantial temporal locality, incorporating the fast register cache effectively alleviates the challenges posed by the NVM [11].

Within HI-RF, the register cache serves as a write buffer, storing computed data from execution units. The data stored in the cache is only written back to the NVM if there is a cache conflict, which is commonly observed in the register cache due to its typical direct-mapped cache design. Based on the prior work, there are two schemes of cache indexing for the direct-mapped register cache, a *concatenating* scheme and a *thread context-aware* scheme [10,12]. Both indexing schemes utilize warp IDs and register IDs to calculate register cache indices. The warp ID is determined by the number of scheduled threads, while the register ID is assigned by compilers. Therefore, the cache conflicts within the register cache can be influenced by the compiler's register assignment.

Regardless of the indexing scheme used for the HI-RF, specific bits from the register ID are used for computing the register cache index without any modifications. These bits are crucial in determining cache conflicts in the HI-RF that employs a direct-mapped policy for the register cache. In this work, we introduce the concept of a critical set, which consists of registers sharing the same portion of bits from the register ID for computing the register cache index. If registers

E-mail addresses: eb0313@ewha.ac.kr (E. Jeong), eunseong0607@ewhain.net (E.S. Park), gunjaekoo@korea.ac.kr (G. Koo), yunho_oh@korea.ac.kr (Y. Oh), myungkuk.yoon@ewha.ac.kr (M.K. Yoon).

https://doi.org/10.1016/j.sysarc.2024.103099
Received 3 November 2023; Received in revised form 30 January 2024; Accepted 28 February 2024
Available online 29 February 2024

within the same critical set have a similar number of write operations performed on them, it will increase cache conflicts. On the other hand, if the write operations are distributed well across different critical sets, widening the write count gap between registers in each critical set, the number of cache conflicts will effectively decrease.

Based on this assumption, in our initial evaluation, we assess performance under two scenarios: an ideally best case and an ideally worst case. In the ideally best case, we minimize cache conflicts by maximizing the write count gap of registers in the same critical set, using register write access information obtained from the post-execution analysis. Conversely, in the ideally worst case, we intentionally maximize cache conflicts by grouping registers with similar write counts into the same critical set based on register write count information. Our evaluation results demonstrate that the ideally best case improves performance by 13.8% and 5.7%, while the ideally worst case shows 19.7% and 11.9% of performance degradation compared to the baseline configuration, with the concatenating scheme and the thread context-aware scheme, respectively. Our results reveal the crucial role of register assignment in compilers for HI-RF.

Both ideal cases are implemented using register write counts from post-execution information, which compilers cannot obtain during the compilation. Since write operations are performed on destination operand registers in instructions, it is possible to approximate write counts by tracking how often registers are used as destination operands. Therefore, we leverage the destination operand count of registers as a substitute for the actual write count information. This paper introduces a conflict-aware compiler (CAC) that performs the register assignment with register destination count analysis to reduce cache conflicts in the HI-RF. By utilizing this approximated write count information, CAC can identify optimal register set combinations to assign register IDs. The proposed compiler optimization scheme can be easily integrated with conventional compilers without requiring extensive modifications.

The proposed CAC is evaluated using a cycle-driven simulator, GPGPU-Sim 4.0 [13] and a circuit-level simulator, NVSim [14], with 12 applications. To implement CAC's register assignment, we first modify the simulator to have similar register assignments as SASS [15] and apply the proposed register assignment to the modified version for a detailed analysis of our compiler technique. Benchmark applications are categorized into two groups: a *compiler-sensitive* group and a *compiler-insensitive* group, based on their register usage, which determines the sensitivity to their compiler's register assignment. In our evaluation, CAC shows a performance improvement of 11.1% and 5.9% with the concatenating scheme and the thread context-aware scheme, respectively. Additionally, we analyze occurrences of cache conflicts in the register cache. The proposed compiler optimization scheme effectively reduces cache conflicts by 6.2% and 2.5% with the concatenating scheme and the thread context-aware scheme, respectively. Additionally, CAC shows 73.1 percentage points and 72.9 percentage points lower energy consumption than SRAM with the concatenating scheme and the thread context-aware scheme, respectively.

The remainder of this paper is organized as follows. Section 2 presents the baseline GPU architecture with the HI-RF structure and two register cache indexing schemes. In Section 3, the limitation of the current compiler for HI-RF is presented. In Section 4, we propose CAC for HI-RF, which addresses the problem of conventional compiler. Section 5 shows the performance evaluation of the proposed CAC. In Section 6, we present several related works, and finally, we conclude our paper in Section 7.

2. Background

In this section, we present the baseline GPUs featuring a HI-RF, which includes both an SRAM-based register cache and an NVM-based register file [10]. Additionally, we introduce two previously proposed cache indexing schemes [10,12].

2.1. Graphics processing units

Fig. 1 illustrates the baseline GPU architecture with HI-RF [10]. When a kernel is executed, thread blocks (TBs) are launched on GPUs. A TB scheduler assigns those TBs to streaming multiprocessors (SMs), distributing them based on the available resources. The scheduled TBs are divided into groups called warps or wavefronts, each comprising a fixed number of threads (*e.g.,* 32 threads in NVIDIA GPUs and 64 threads in AMD GPUs). SMs serve as basic computational units within GPU, which are responsible for executing instructions in parallel. The SM comprises four sub-cores (or processing blocks), each of which contains its own resources (*e.g.,* a warp scheduler, a register file, and streaming processors (SPs)). In each sub-core, tens of warps can be executed concurrently, and the execution of these scheduled warps is managed by the warp scheduler, issuing warps to SPs. Finally, all the threads within the same warp are executed simultaneously in lockstep, which is known as the single instruction multiple threads (SIMT) execution model.

To support the concurrent execution of such a large number of threads (usually thousands of threads are concurrently executed on each SM), SMs are equipped with large-sized register files. These register files are essential in enabling quick access to the required data for threads, facilitating efficient parallel processing. As GPU generations have evolved, the size of the register files has grown to meet the increasing demand of modern parallel workloads [3,16]. However, prior research indicates that the register files significantly contribute to the GPU energy consumption, accounting for approximately 20% of the total energy usage [5,17]. Consequently, several research studies have suggested a range of energy-efficient register file designs [6,8,18,19].

In addition to the register files, each SM has an L1 cache and shared memory (also known as scratchpad memory), which are shared by sub-cores. Programmers possess complete control over the shared memory space, while the L1 cache operates transparently without their direct intervention. The size of both the L1 cache and shared memory can be adjusted by programmers according to their specific needs and preferences. Furthermore, GPUs have an L2 cache and global memory that are shared by all threads scheduled on the SMs. These memory components have progressively expanded in size as a response to the growing demands of machine learning applications and their extensive data prerequisites [3,20].

2.2. NVM-based register file

Traditional GPUs employ SRAM-based register files, known for their high energy consumption due to the high leakage power of SRAM. Thus, several prior studies have explored the integration of NVM-based register files on GPUs, taking advantage of their significantly low static energy consumption with low leakage power, which has been observed to be much lower than that of the SRAM-based register files [6,7]. In addition to such low leakage power consumption, NVM provides high density and good scalability [22–24]. The detailed parameters used to evaluate SRAM and NVM-based register files are provided in Table 3 in Section 5.1. Among the various NVM technologies such as phase change memory (PCM), resistive memory (ReRAM), and spin transfer torque magnetic random access memory (STT-MRAM), STT-MRAM has emerged as the most promising candidate that can replace SRAM, due to its relatively lower read/write latency, and higher write endurance than other types. Thus, several previous studies suggest STT-MRAM-based register files on GPUs to address the issues associated with SRAM-based register files [21,25,26].

Fig. 2 shows the STT-MRAM-based register file architecture and the detailed cell design of STT-MRAM [21], which is employed for our evaluation. Within each register bank, the cell array consists of 4096 rows of 64 bits, with each row comprising STT-MRAM cells. The STT-MRAM cell utilizes a magnetic tunneling junction (MTJ) pillar to store a bit in a resistive state, with an accessibility controlled by an access

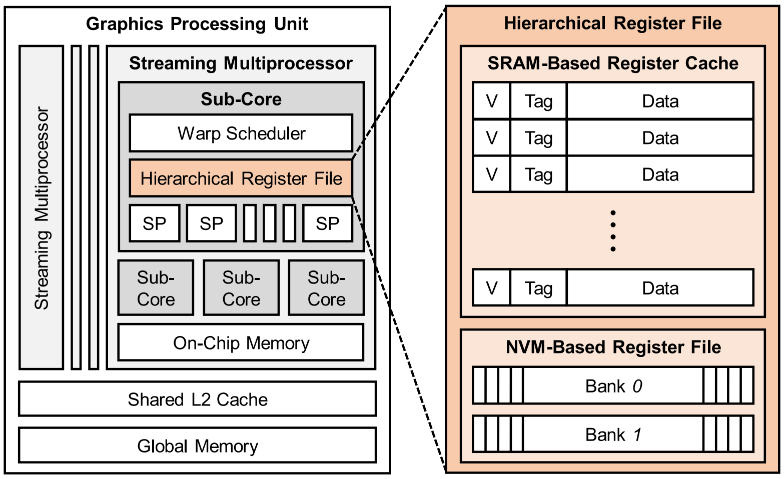


Fig. 1. Overview of baseline GPU architecture with hierarchical register file.

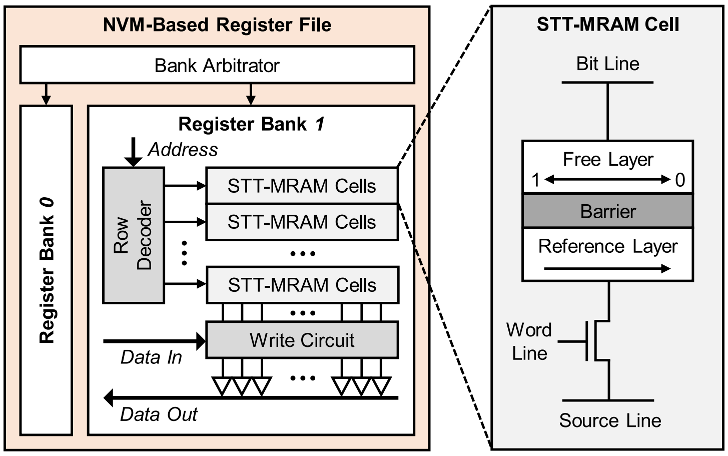


Fig. 2. STT-MRAM-based register file architecture and cell design of STT-MRAM [21].

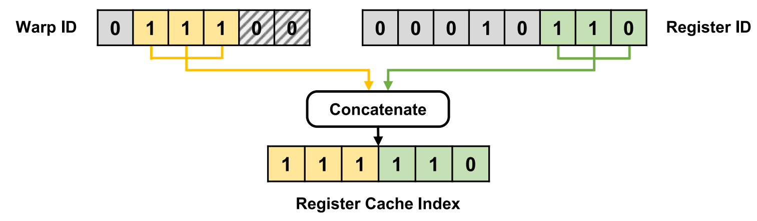
transistor. STT-MRAM uses spin-polarized current which is passed via the MTJ to manipulate the magnetic alignment of its two layers — the free layer and the reference layer. When these layers are parallel, the MTJ exhibits low resistance (*i.e.,* the logic value of 0). Conversely, if the layers are anti-parallel, the MTJ exhibits high resistance (*i.e.,* the logic value of 1). To further optimize the cell area and minimize the static power consumption of STT-MRAM-based register file, prior research proposed the register file design with multi-level cell (MLC) STT-MRAM technology that can double the density of STT-MRAM [21,26]. Additionally, another study proposed STT-MRAM-based register files to address the read disturbance issue inherent in STT-MRAM due to the scaling technology [25]. In conclusion, considering its advantages and its compatibility with CMOS, STT-MRAM could be an attractive solution for the integration as the register files on GPUs [22,24,27,28].

2.3. Hierarchical register file

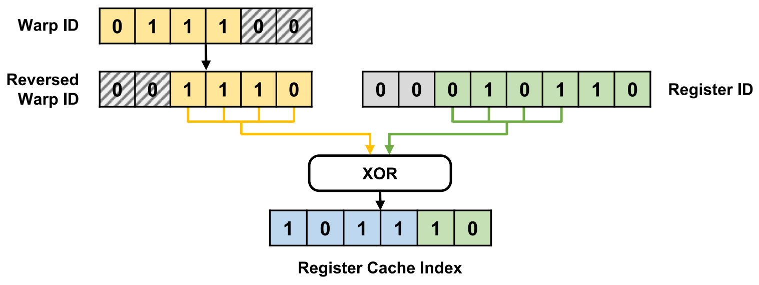
NVM has proven effective in reducing energy consumption when compared to SRAM. However, the adoption of NVM-based register file on GPUs still introduces challenges, particularly concerning its long write latency, leading to a performance degradation. Even with STT-MRAM, known for its minimal write latency among NVMs, NVM-based register file suffers from a write latency four times longer than SRAM-based register file [7,28,29]. To address this issue, Li et al. proposed a hybrid register file that incorporates an SRAM write buffer for the NVM-based register file [7]. In this structure, data is first written to the fast SRAM and then stored in NVM while executing the next instructions. While this approach can mitigate the long latency problem associated with NVM, it does not address the write endurance issue, which is also recognized as a critical problem with NVM. In a hybrid design of this nature, all data written to the SRAM write buffer must eventually be written back to NVM.

To tackle these issues, researchers have shown considerable interest in HI-RF [10]. This approach leverages the benefits of both SRAM-based register cache and NVM-based register file to mitigate the limitations associated with using solely NVM-based register file. The apricot color of Fig. 1 illustrates the detailed structure of the HI-RF. For read operations of the HI-RF, the corresponding values can be loaded from both the SRAM-based cache and the NVM-based registers. Meanwhile, all the computed results are first written only to the SRAM-based cache. In contrast to the hybrid register file, data is written back to the NVM only if there is no available cache line remaining for storing the value (*i.e.,* when cached data must be evicted). However, the choice of the direct-mapped cache of the HI-RF brings attention to the issue of cache conflicts. Because with the direct-mapped policy, data associated with a specific cache index can exclusively reside in a single-mapped entry within the cache. Nevertheless, by leveraging the data locality commonly observed in GPU applications [11], write-back operations are minimized with the HI-RF, thereby mitigating the write endurance problem of hybrid register files.

The HI-RF structure demonstrates a comparable area overhead when contrasted with the conventional SRAM-based register file, attributed to the higher density of NVM. Table 1 illustrates the relative



(a) Concatenating scheme.



(b) Thread context-aware scheme.

Fig. 3. Two register cache indexing schemes.

Table 1
Normalized area overhead of three different register file designs: SRAM-based, NVM-based, and hierarchical.

Register file	SRAM-based	NVM-based	Hierarchical
Area Overhead	1.0×	0.499×	0.945×

area overhead of both NVM-based (STT-MRAM-based) register file and HI-RF, encompassing an NVM-based register file and an SRAM-based register cache, in comparison to an SRAM-based register file. Area measurements are performed using a circuit-level simulator, NVSim [14], with a 22-nm process node. Due to the higher density of NVM compared to SRAM, an NVM-based register file can be constructed at 49.9% of the size of an SRAM-based register file. Conversely, for the HI-RF, the area complexity is similar in size (94.5%) to the conventional SRAM-based register file.

2.4. Register cache indexing schemes

The choice of a cache indexing scheme is a crucial factor in determining the overall performance of GPUs since the HI-RF employs a direct-mapped register cache design. In this section, we explain two previously proposed register cache indexing schemes, which are depicted in Fig. 3. The first scheme is referred to as a *concatenating* scheme, where the register ID and warp ID are simply concatenated to form the cache index [10]. The second scheme is known as a *thread context-aware* scheme, which computes the register cache index by taking into account a correlation between a number of scheduled warps and a number of registers utilized by threads [12].

Note that in the baseline architecture, each SM can concurrently execute up to 64 warps, requiring six bits to represent their respective warp ID within each SM. These 64 warps are then assigned to one of the four sub-cores within the SM based on their warp IDs. To evenly distribute the 64 warps among the four sub-cores, the two least significant bits (LSBs) of the warp ID are employed for choosing the sub-core. Consequently, these two bits are unused when calculating the register cache index within each sub-core. In other words, four bits from the most significant bits (MSBs) of the warp ID are used to compute the register cache index. Furthermore, it is important to note that threads in GPUs can have up to 256 registers, necessitating an 8-bit register ID to represent all available IDs. For the computation of the register cache index, six bits of the register ID are utilized, starting from the LSB.

Concatenating scheme: Fig. 3(a) illustrates an example of the concatenating scheme. As shown in the figure, the first scheme involves concatenating partial bits of the warp ID and partial bits of the register ID to compute the register cache index. Initially, the cache index computation entails extracting the lower three bits starting from the third LSB of the warp ID and the lower three bits of the register ID from its LSB. These two selectively chosen bits are then combined through the concatenation to generate the index for the register cache. For example, considering a warp ID of 011100 and a register ID of 00010110, three bits from the warp ID (111) and the lower three bits from the register ID (110) are concatenated to form the cache index of 111110.

Thread context-aware scheme: Differing from the approach of concatenating bits from the warp ID and the register ID, the thread context-aware scheme considers the correlation between the number of scheduled warps and the number of registers employed by each thread [12]. Fig. 3(b) shows an example of the thread context-aware scheme. In GPUs, the number of scheduled warps decreases as the number of registers used by threads increases, and vice versa. Consequently, when computing the register cache index, this scheme initiates by reversing the bits of the warp ID and subsequently applying XOR operations using the six bits of the register ID and the four bits of the warp ID. It is worth noting that only six bits of the register ID are used, as the cache index bit length is six. Additionally, the two LSBs of the warp ID are excluded from the index computation, as they are used to determine the sub-core for the execution. As an example, considering a warp ID of 011100 and a register ID of 00010110, the first four bits from the warp ID (0111) are reversed, resulting in 1110. This is then followed by XOR operations with the lower six bits of the register ID. Given that the number of sampled bits from the warp ID is fewer than that of the register ID, XOR operations are conducted on the four bits from the MSB of each ID. After completing these steps, the computed cache index with the thread context-aware scheme becomes 101110.

3. Motivation

The SRAM-based register cache within the HI-RF is relatively smaller than the NVM-based register file. Therefore, effectively utilizing the register cache is a critical factor for minimizing register write-back operations that arise from conflicts within the register cache. These operations can lead to the performance degradation due to an extended duration of NVM write operations. Additionally, reducing cache conflicts holds significance as the NVM has limitations on the total number of write operations, known as the write endurance problem [10].

In Section 2.4, we present two indexing schemes for the register cache in the HI-RF. In both indexing schemes, the warp ID and the register ID serve as base bits for calculating the index. The methods for assigning these two distinct IDs differ significantly. The warp IDs are established based on the number of threads generated by applications, with a determination typically made by software developers. On the other hand, the register IDs are determined by compilers. However, current compilers do not consider the structure of the HI-RF and its associated cache indexing schemes, thus cache conflicts can fluctuate depending on how the compiler assigns register IDs.

As illustrated in Fig. 3, whether the concatenating or thread context-aware scheme is employed, several bits extracted from the register ID are used without any modifications as a part of the register cache index. In the concatenating scheme, as shown in Fig. 3(a), these bits correspond to the three LSBs of the register ID. Similarly, in the thread context-aware scheme, shown in Fig. 3(b), the two LSBs of the register ID are utilized without modifications. Thus, cache conflicts can occur when multiple distinct warps share the same two or three LSBs in their register IDs, since HI-RF relies on a direct-mapped cache design. Consequently, these sampled bits, used without alterations, can influence register cache conflicts. In this paper, we group registers that share the same two or three LSBs of their IDs and name the set of registers as a *critical set*. For example, when an application utilizes 27 registers with the concatenating scheme for the register cache indexing, registers with IDs of 1 ($00000\underline{001}$), 9 ($00001\underline{001}$), 17 ($00011\underline{001}$), and 25 ($00111\underline{001}$) are combined into one critical set, as they share the same three LSBs of 001. Similarly, other registers also belong to critical sets determined by specific LSBs of their register IDs.

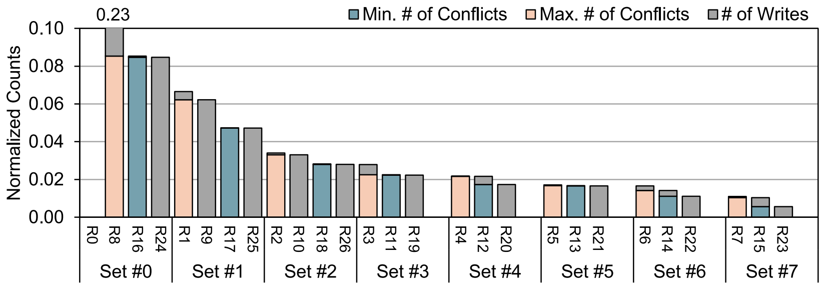
The cache conflicts can arise when warps use registers within such critical sets. The frequency of cache conflicts strongly correlates with the write count of individual registers because the HI-RF initiates write-back operations exclusively when the data is written to a cache line containing valid data. Thus, if registers with high write counts are grouped into the same critical set, it can increase conflicts within the register cache. Alternatively, if the registers with high write counts are distributed among various critical sets and registers within the same critical set exhibit substantial differences in write counts, this can potentially reduce conflicts among the registers within those critical sets.

The assignment of register IDs is performed by compilers. If the compilers frequently assign registers that are used as destination registers to the same critical set, it can result in intensive cache conflicts within the same cache entry, leading to performance degradation. Given this assumption, we create two register assignment policies: an *ideally best* policy (referred to as a `Best` policy) and an *ideally worst* policy (referred to as a `Worst` policy). In both policies, we initiate the process by counting the occurrences of writes for all register IDs using a `Baseline` policy. For example, Fig. 4 shows the normalized register write count results of the NN application under the concatenating scheme with these two different register assignment policies. The write count for each register is normalized to the total register write counts.

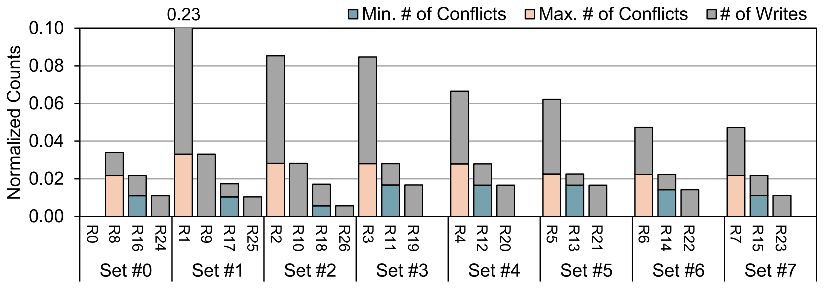
The figure shows the maximum and minimum conflicts possible within each critical set, considering the worst scenario where two registers repeatedly attempt to write alternately to the same cache line. As the concatenating scheme is utilized, registers sharing the same three LSBs are grouped into one critical set, resulting in eight sets. The `Worst` policy is designed to maximize register cache conflicts by intending to assemble registers with similar write counts into the same critical set. This involves altering register IDs to achieve a similar number of write accesses between registers within each critical set. As shown in the figure, the registers within each critical set exhibit write counts with small gaps between themselves. Due to this assignment, registers with high write counts are also mapped to the same critical set, which exacerbates cache conflicts in some cache entries. In contrast, the `Best` policy is implemented to minimize register cache conflicts. To reduce the conflicts, registers that are written frequently should be distributed well across different critical sets. Thus, adjustments are made to the register IDs to maximize the gap of write accesses between registers within the same critical set. As illustrated in Fig. 4(b), the assignment of register IDs with the `Best` policy enlarges the gap between the write accesses of registers mapped to a single cache entry. For instance, R8 of Set #0 in the `Worst` policy corresponds to R1 of Set #1 in the `Best` policy, as they have the same write count. Under the `Worst` policy, conflicts within R8 and R16 can potentially result in a maximum of 8.5% of the total write counts being attributed to write-back operations. In contrast, in the `Best` policy, the maximum number of write-back operations within Set #1, involving R1 and R9, is approximately 3.3% of the total write counts when they attempt to write values into the same register cache line repeatedly. Consequently, the `Best` policy effectively reduces conflicts in the register cache by distributing registers with high write count across critical sets.

To verify a performance gap between these policies, we measure the performance under three distinct register assignment policies: `Baseline`, `Best`, and `Worst`. In Fig. 5, the performance results are normalized to that of `Baseline`, which employs a default compiler register assignment policy. Detailed simulation methodology is provided in Section 5. As shown in the figure, we divide benchmark applications into two groups: a *compiler-sensitive* group and a *compiler-insensitive* group. Applications within the compiler-sensitive group, positioned on the left side of the graph, exhibit a notable performance gap between the `Best` and `Worst` policies. Conversely, applications within the compiler-insensitive group, positioned on the right side of the graph, demonstrate nearly identical performance under both the `Best` and `Worst` policies, with a maximum gap of 2.0 percentage points. For the concatenating scheme, on average, the `Best` policy enhances performance by 13.8% compared to `Baseline`, whereas the `Worst` policy leads to the performance decrease of 19.7% over `Baseline`. Overall, the performance difference between the `Best` and `Worst` policies is 33.5 percentage points. Notably, the MM application shows 69.5 percentage points gap between the `Best` and `Worst` policies. Similarly, for the thread context-aware scheme, the `Best` policy increases the performance by 5.7%, while the `Worst` policy decreases the performance by 11.9% compared to `Baseline`, resulting in an average performance gap of 17.6 percentage points.

The applications in the compiler-insensitive group show little performance variations across different register assignment policies. In essence, the performance of these applications is hardly affected by how register IDs are assigned, regardless of the chosen cache indexing schemes. Based on our analysis, the minimal performance gap observed in some applications is due to the limited utilization of registers; in other words, when only a small number of registers are utilized, the register cache index is more likely to be mapped to a single unique entry, mitigating the occurrences of the cache conflicts. Consequently, the performance does not fluctuate depending on different register assignment policies. The applications in the compiler-sensitive group show a high utilization of registers compared to the compiler-insensitive applications. As these applications utilize more registers, they become more sensitive to different register assignments, resulting in performance variations. Therefore, we establish a threshold of 15 registers to concretely define compiler-sensitive (15 registers or more) and compiler-insensitive (less than 15 registers) applications.

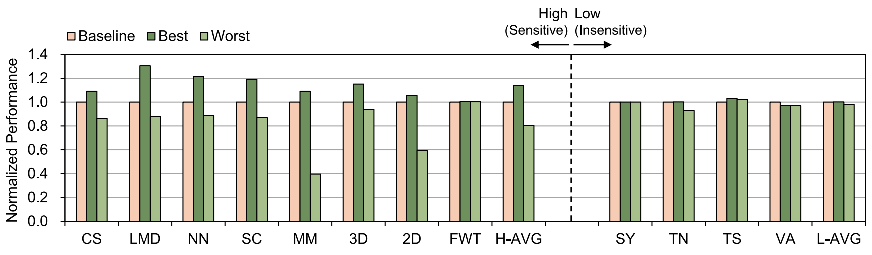


(a) Worst policy.

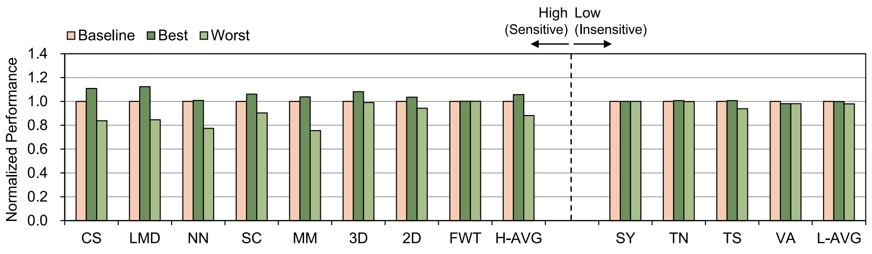


(b) Best policy.

Fig. 4. Register write counts for the NN application with two different register assignment policies: `Worst` and `Best`.



(a) Performance results of a concatenating scheme.



(b) Performance results of a thread context-aware scheme.

Fig. 5. Normalized performance of two cache indexing schemes using three different register assignment policies: `Baseline`, `Best`, and `Worst`.

Based on our observations, the performance of GPUs with HI-RF is greatly influenced by the register assignment policy performed by compilers. Particularly, the applications in the compiler-sensitive group can undergo a significant performance degradation due to an inappropriate register assignment, as we observe with the `Worst` policy. To minimize register cache conflicts, compilers should assign registers by considering the distribution of write operations on registers across critical sets, similar to the strategy employed in the `Best` policy. In this paper, we propose a compiler optimization scheme aimed at efficiently utilizing the HI-RF on GPUs, with a specific focus on reducing cache conflicts in the register cache while considering critical sets of registers.

4. Conflict-aware compiler

In Section 3, we introduce two register assignment policies: `Best` and `Worst`. Both of these policies are designed based on post-execution information, primarily relying on register write counts, as register write operations introduce conflicts in the register cache. Therefore, it is practically unfeasible for compilers to implement the `Best` policy since the actual number of register writes cannot be determined until an application completes its operations.

To address this challenge, we propose an alternative approach that approximates the register write counts by measuring how frequently registers are used as destination operands within instructions. Compilers can easily obtain this destination operand count (destination count) information by parsing the instructions. While this approach does not provide the exact write counts due to some instructions being repeatedly executed, it offers a practical means to approximate the number of write operations performed on registers during an execution. Leveraging this information, we introduce a Conflict-Aware Compiler (CAC) for HI-RF, which optimizes register assignment to alleviate register cache conflicts. CAC utilizes destination counts as approximated values for the write counts on registers and performs register assignment similar to the Best policy. This involves sorting the register IDs using destination counts as key values. Then, the number of write operations performed on each register within the same critical set exhibits differences. Consequently, similar to the Best policy, registers within the same critical set should exhibit significant gaps in their write counts.

Algorithm 1 Register Assignment Optimization

```
Input: a list of register IDs and a list of destination counts for each register
Output: a new list of register structs that contain mappings of original register ID and new register ID
1:  procedure assignRegId($regIdList$, $dstCountList$)
2:      Combine the $regIdList$ and $dstCountList$ into a list of register structs
3:      $regList \leftarrow$ zip($regIdList$, $dstCountList$)
4:      Sort register IDs based on their destination counts
5:      $sortedRegs \leftarrow SortByDstCount(regList)$
6:      Create the new register ID list based on the sorted register IDs
7:      for each $reg$ in $sortedRegs$ do
8:          $newRegList$.append($sortedRegs$[i])
9:      end for
10:     return $newRegList$
11: end procedure
12:
13: function $SortByDstCount(regIdList, dstCountList)$
14:     Sort a list of register structs by the destination counts in descending order
15:     $sortedRegs \leftarrow regIdList$.sort(key=lambda $reg$:$reg.dstCount$, reverse = True)
16:     return $sortedRegs$
17: end function
```

Algorithm 1 provides a detailed description of the register assignment process employed by CAC. To initiate the process, CAC takes two input lists: a list of register IDs used by an application (referred to as a *regIdList*) and a list of destination counts for each register (referred to as a *dstCountList*). It is important to note that the *dstCountList* contains approximated write counts sampled from register destination counts within instructions. These destination counts serve as the keys for adjusting the register IDs. The primary goal of this procedure is to distribute write operations evenly across different critical sets while increasing the gap of writes between registers that belong to the same critical set. To achieve this, CAC sorts the register IDs based on their destination counts. The procedure starts by creating a list of register structures, with each structure containing a mapping between the original register ID and its associated destination count. Subsequently, CAC utilizes the *SortByDstCount* function to sort these register structures in descending order according to their destination counts. The result of the sorting is then stored in the *sortedRegs* list. Once the *sortedRegs* list is obtained, CAC generates a new register ID list by incorporating the sorted values. This new list reflects the adjustments made to the original register IDs based on their destination counts. Finally, the information of the mapping between the new and original register IDs, is returned as *newRegList*. Consequently, these new register IDs can be employed to assign new register IDs using the original register IDs as references. This process aims to reduce register cache conflicts by effectively distributing write operations across different critical sets, ultimately improving the performance of GPUs with HI-RF.

Table 2
Parameters of baseline GPU.

Parameters	Value
# of SMs	80 SMs
Warp size	32 threads
Maximum # of threads per SM	2,048 threads
# of warp schedulers per SM	4 schedulers
Warp scheduling policy	GTO
L1 cache size per SM	32 KB
L2 cache size per SM	4.5 MB
# of SRAM register cache entries	256 entries
SRAM register cache entry size	1,024 bits
# of NVM register banks	8 banks
NVM register bank width	64 bits
Maximum # of NVM registers	65,536 registers

Table 3
Parameters of SRAM and NVM [10,12,14,29].

Parameters	SRAM	NVM
Read latency (cycle)	1	1
Write latency (cycle)	1	4
Read energy (pJ/bit)	0.203	0.239
Write energy (pJ/bit)	0.191	0.300
Leakage power (mW)	248.7	16.2

Our proposed compiler optimization scheme can be integrated into compilers without requiring significant modifications. CAC can be implemented as an extension of compilers such as NVCC, which is NVIDIA's commercial compiler for compiling CUDA applications [30]. When an application is targeted to run on GPUs equipped with HI-RF (*e.g.,* when an option for using HI-RF is provided), CAC could be activated. The compiler then parses the SASS instructions of the application, gathers the necessary information for the proposed register assignment, and assigns the register IDs as explained previously. It is important to note that the adjustments by compilers are made to SASS, not to the PTX, as the SASS is the final assembly executed on GPUs. Moreover, this optimization does not interfere with or impact the original compiler optimization process; instead, it functions as an additional step that complements the existing compilers.

In summary, CAC optimizes register assignment to effectively reduce cache conflicts in HI-RF, functioning in a manner similar to the Best policy by utilizing destination count information. This reduction in cache conflicts and subsequent reduction in write-back operations improve the performance of GPUs equipped with HI-RF. Additionally, this optimization can help alleviate the write endurance problem associated with the NVM-based register file of HI-RF.

5. Methodology and evaluation

In this section, we present our simulation methodology and conduct a performance analysis of the proposed CAC. Additionally, to validate the effectiveness of CAC, we assess the number of conflicts within the register cache and the energy consumption of registers. Lastly, we conduct various sensitivity studies.

5.1. Methodology

For the performance evaluation of GPUs employing HI-RF, we utilize a cycle-driven simulator, GPGPU-Sim 4.0 [13]. It is important to note that PTX assembly is not directly executed on GPUs; instead, it needs to be translated into SASS representation, determining register ID assignments [15]. The GPGPU-Sim simulator, responsible for translating PTX assembly into PTXPlus for GPU architecture evaluation, has

Table 4
Benchmark applications.

Application name	Abb.	Application name	Abb.
Convolution Separable [36]	CS	2D Convolution [34]	2D
LavaMD [33]	LMD	Fast Walsh Transform [36]	FWT
Neural Network [35]	NN	Symmetric Rank-2k [34]	SY
Scan [32]	SC	Naive Matrix Trans. [36]	TN
Matrix Multiply [36]	MM	Trans. Simple Copy [36]	TS
3D Convolution [34]	3D	Vector Add [36]	VA

been modified in the PTXPlus translation for our simulation. These modifications align with SASS translation and incorporate insights from SASS translation analysis, particularly focusing on register ID assignment. Our analysis indicates that the modified PTXPlus translation demonstrates similar register usages after undergoing SASS translation. On top of that, we propose a CAC implementation that adjusts the register assignment to effectively utilize HI-RF.

Our baseline GPU architecture is similar to the NVIDIA Volta architecture [31], and the HI-RF structure is implemented in the same manner as in the prior work [7,10,29]. The detailed GPU parameters are listed in Table 2. Additionally, to measure the energy consumption of SRAM-based register file, NVM-based register file, and HI-RF, we use a circuit-level simulator, NVSim [14]. The parameters for SRAM and NVM are listed in Table 3. We select 12 benchmark applications from SHOC [32], Rodinia [33], PolyBench [34], ISPASS2009 [35], and NVIDIA SDK [36], as shown in Table 4.

For our comparative analysis of performance, we use seven different configurations: `SRAM`, `NVM`, `HI`, `HI_CAC`, `HI_Best`, `HI_Worst`, and `HI_Full`. `SRAM` and `NVM` represent the register file implementations with SRAM and NVM, respectively. Other configurations with names including `HI` are based on the HI-RF architecture but differ in register assignment policies. `HI` is the default register assignment policy and `HI_CAC` is the proposed register assignment policy. `HI_Best` and `HI_Worst` policies are identical to those explained in Section 3, which are implemented with the post-execution information. Lastly, `HI_Full` means that HI-RF has a fully-associative register cache. This approach addresses the limitations of HI-RF by implementing a register cache as a fully-associative cache structure instead of using the direct-mapped cache. However, due to the complexity of the hardware, the adoption of a fully-associative register cache poses significant challenges [12]. Therefore, the results of `HI_Full` can be interpreted as outliers in our analysis.

As mentioned in Section 3, we categorize benchmark applications into two groups: a *compiler-sensitive* group and a *compiler-insensitive* group, employing a threshold of 15 registers. In our evaluation, we primarily deal with compiler-sensitive applications since the register assignment has a more significant impact on their performance. Compiler-insensitive applications, characterized by a small number of registers, are minimally influenced by register assignment policies. Throughout evaluation figures, the average value of each group is located in the rightmost bars of the respective group.

5.2. Performance analysis

Fig. 6 shows the normalized performance of the benchmark applications under seven different configurations (`SRAM`, `NVM`, `HI`, `HI_CAC`, `HI_Best`, `HI_Worst`, and `HI_Full`). The performance is measured in instructions per cycle (IPC), and each performance value is normalized to that of `SRAM`.

Our proposed compiler, denoted as `HI_CAC`, shows performance improvements for the compiler-sensitive applications with both the concatenating and thread context-aware schemes. On average, with the concatenating scheme, `HI_CAC` improves the performance of compiler-sensitive applications by 11.1% compared to `HI`. In contrast, the compiler-insensitive applications exhibit minimal performance differences across various assignment policies. Note that the overall performance improvement achieved through `HI_CAC` is slightly lower than that of `HI_Best`; the improvement with `HI_CAC` is 1.5 percentage points lower than that with `Best`. This gap arises because the `HI_Best` policy benefits from post-execution information obtained from each benchmark application, as discussed in Section 3. Additionally, the `HI_Full` policy demonstrates the most notable performance improvements across all applications. Unlike the direct-mapped cache design, in the fully-associative cache design, register write operations can be performed in any entry within the register cache using the least recently used replacement policy. Thus, with any register assignment techniques, `HI_Full` exhibits the most performance improvements. However, as presented in prior work, implementing such a cache is challenging due to its inherent complexity and associated hardware requirements [37]. Overall, our proposed scheme with HI-RF shows performance results similar to `SRAM` and 44.4 percentage points better than `NVM`.

Fig. 6(b) presents the performance results for the thread context-aware scheme, which exhibit similar patterns to those of the concatenating scheme. `HI_CAC` improves the performance of compiler-sensitive applications by 5.9% over `HI`. The average performance improvement with the `HI_Best` policy is 5.4% over `HI`, and the `HI_Full` policy exhibits the most substantial performance improvement of 9.7% over `HI`. Overall, `HI_CAC` exhibits 1.9% lower performance than `SRAM` and 44.5 percentage points higher performance over `NVM`.

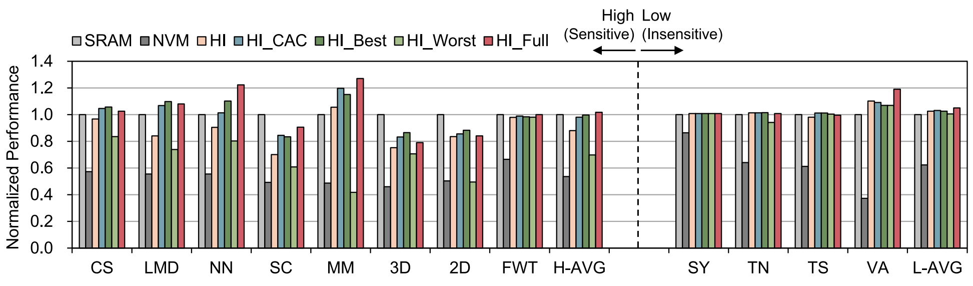
Note that while both schemes show performance improvements with `HI_CAC` for the compiler-sensitive applications, the performance improvement in the thread context-aware scheme is smaller than that in the concatenating scheme. This is because the concatenating scheme simply concatenates bits from the register ID and warp ID, making its performance more sensitive to the compiler's register assignment. Conversely, the thread context-aware scheme computes the cache index considering the correlation between the number of scheduled threads and the number of used registers. Thus, the impact of register assignment is reduced under the thread context-aware scheme compared to the concatenating scheme. To summarize, our proposed scheme demonstrates effective performance improvement under any register cache indexing scheme, achieving a performance level similar to SRAM.

Moreover, specific applications such as MM, NN, and VA exhibit enhanced performance when HI-RF is employed compared to SRAM. The key factor contributing to this improved performance is the register cache design, which decreases register access cycles by mitigating register bank conflicts that can substantially degrade overall performance. Similar results are observed in several prior works [38,39].

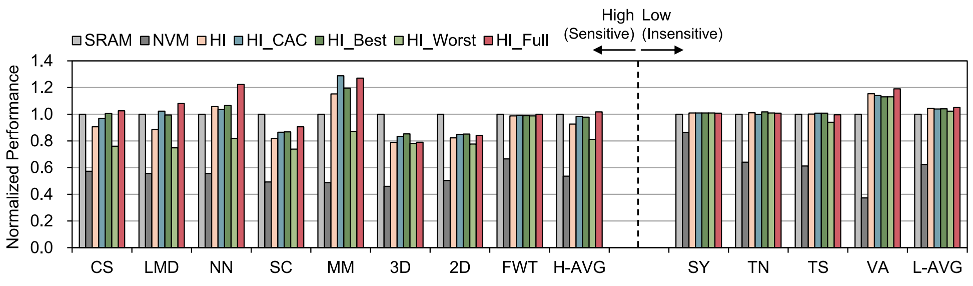
5.3. Cache conflict analysis

To evaluate the correlation between the cache conflicts and performance, we measure the occurrences of register cache conflicts in each application. The number of cache conflicts is determined by analyzing the total write-back operations to the NVM register file of HI-RF. Given that the compiler-insensitive applications are unaffected by different register assignment policies, our cache conflict analysis focuses on the benchmark applications that are sensitive to the compiler's register assignment.

As we explain in Section 3, register cache conflicts degrade the performance of GPUs equipped with HI-RF. Fig. 7 shows the register cache conflict results for the compiler-sensitive applications using two cache indexing schemes under five distinct register assignment policies. For the concatenating scheme, the `HI_Worst` policy exhibits significantly large cache conflicts, which is overall 36.5% higher than the `HI` policy. Considering the long write latency issue of the NVM-based register file within the HI-RF structure, these frequent conflicts could potentially prolong an execution duration and degrade the performance,

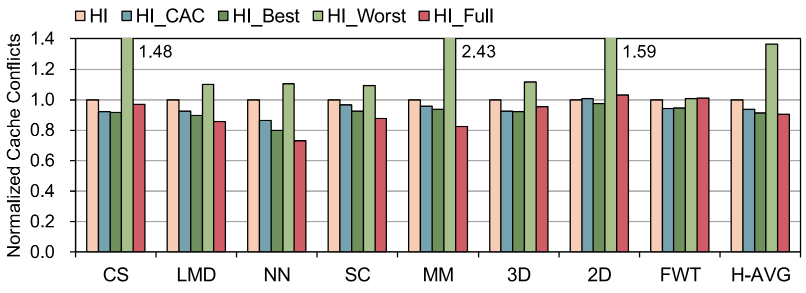


(a) Performance results of concatenating scheme.

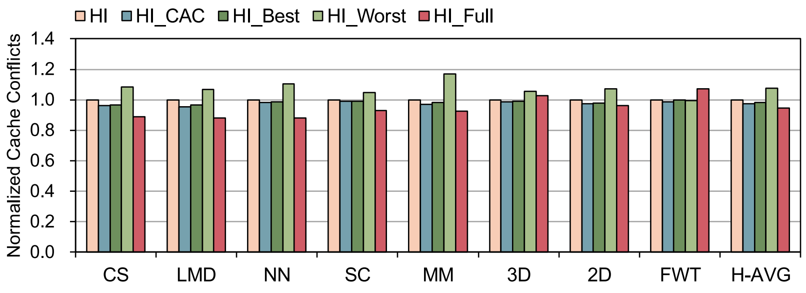


(b) Performance results of thread context-aware scheme.

Fig. 6. Normalized performance of two cache indexing schemes using seven different configurations: SRAM, NVM, HI, HI_CAC, HI_Best, HI_Worst, and HI_Full.



(a) Register cache conflict results of concatenating scheme.

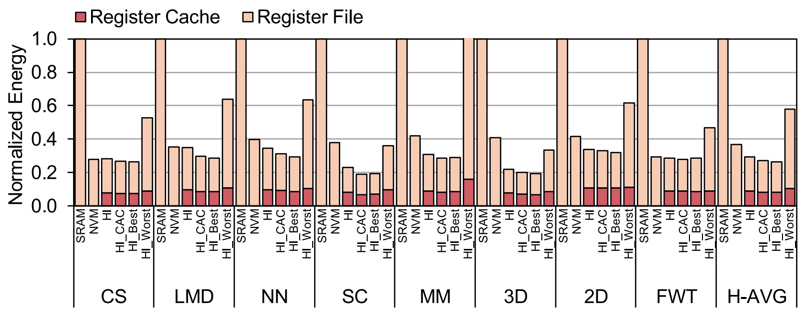


(b) Register cache conflict results of thread context-aware scheme.

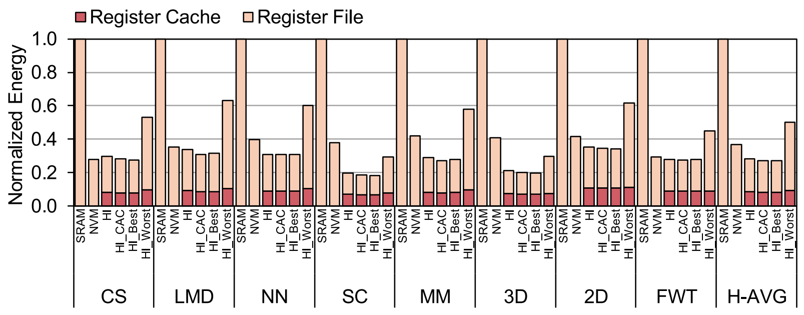
Fig. 7. Normalized register cache conflicts of two cache indexing schemes using five different register assignment policies: HI, HI_CAC, HI_Best, HI_Worst, and HI_Full.

as observed in Fig. 6. Conversely, the proposed compiler, HI_CAC, shows an average reduction of 6.2% in cache conflicts compared to HI. Similarly, the HI_Best policy reduces cache conflicts by 8.6%, which outperforms HI_CAC by 2.4 percentage points. This gap arises because the HI_Best policy leverages a precise knowledge of the exact register write counts for the ideal scenario instead of using the approximated counts. Finally, HI_Full shows the smallest cache conflict results, with an average reduction of 9.4% compared to HI.

For the thread context-aware scheme, the HI_CAC and the HI_Best policies decrease cache conflicts by 2.5% and 1.8% over HI. Similar to the concatenating scheme, HI_FULL exhibits an average of 5.5% fewer cache conflicts than HI. Note that the HI_Worst policy only increases the cache conflicts by 7.4% compared to HI, contrasting with the 36.5% increase observed in the concatenating scheme. This is because under the thread context-aware scheme, considering the relation between the number of scheduled threads and the number of



(a) Register energy consumption results of concatenating scheme.



(b) Register energy consumption results of thread context-aware scheme.

Fig. 8. Normalized register energy consumption of two cache indexing schemes using six different configurations: SRAM, NVM, HI, HI_CAC, HI_Best, and HI_Worst.

registers, the cache index computation involves XOR operations instead of simply concatenating partially chosen bits. Consequently, the impact of the register assignment on cache conflicts under the thread context-aware scheme decreases compared to the concatenating scheme. Based on our evaluation results, the policies that demonstrate performance improvements also show a reduction in the number of cache conflicts. In summary, the proposed CAC effectively decreases cache conflicts with its register assignment optimization, followed by the observed performance improvements.

5.4. Energy consumption

Fig. 8 shows the normalized register energy consumption of six configurations (SRAM, NVM, HI, HI_CAC, HI_Best, and HI_Worst) under the two register indexing schemes. The data is normalized to the energy value of SRAM, and each includes the values for both the register cache and register file. Since SRAM and NVM configurations do not utilize the register cache, they only exhibit the energy consumption of the register file.

Overall, NVM and HI configurations show significantly lower energy consumption than SRAM, with 63.3% and 70.6% of decrease, respectively. This low energy consumption is attributed to NVM's substantially lower leakage power consumption in comparison to SRAM. By effectively reducing cache conflicts in the register cache, CAC can further reduce the energy consumption of HI-RF. Our proposed compiler reduces overall energy consumption by 73.1 percentage points and 2.5 percentage points over SRAM and HI with the concatenating scheme, respectively. Similarly, HI_CAC reduces energy consumption by 72.9 percentage points and 1.2 percentage points over SRAM and HI with the thread context-aware scheme. Note that the average energy consumption of HI_Worst is 28.3 percentage points and 21.6 percentage points higher than NVM, with the concatenating scheme and the thread context-aware scheme, respectively. Such high energy consumption is due to the HI_Worst policy designed to maximize register cache conflicts, leading to redundant write-back operations in the register file.

5.5. Sensitivity study

In the concatenating scheme, the performance of HI-RF is influenced by the number of bits used from a register ID since it directly affects a number of critical sets. We evaluate performance by varying configurations of the concatenating scheme: W4R2, W3R3, and W2R4. W# represents the number of bits used for a warp ID, and R# represents the number of bits used for a register ID in cache indexing. Fig. 9 shows performance results of the concatenating scheme with three different configurations under four different register assignment policies: HI, HI_CAC, HI_Best, and HI_Worst.

Our results show that HI_CAC improves performance over HI by 11.8%, 11.9%, and 6.8% for W4R2, W3R3, and W2R4, respectively. As the number of bits selected from the register ID increases, the performance improvement decreases. This lower performance improvement is attributed to the reduction in the number of critical sets. When more bits from the register ID are utilized, the number of registers sharing the same LSBs will increase, leading to a large size of the critical set. Consequently, with a smaller number of large critical sets, the impact of enlarging the write operation gaps within each critical set becomes less effective. In contrast, with a larger number of small critical sets, the impact of the proposed optimization will increase. Overall, W4R2 exhibits the most significant performance improvement with HI_CAC, as it uses only two bits for grouping critical sets, resulting in more numerous, smaller critical sets.

In our baseline GPU architecture, each register cache is equipped with 256 entries. The performance of GPUs is significantly influenced by the number of cache entries in each register cache, as a larger cache size can reduce cache conflicts. Therefore, we evaluate performance by varying the register cache size to assess the effectiveness of our proposed compiler optimization scheme. Fig. 10 shows performance results for the concatenating scheme, represented as CS_Base (concatenating scheme with HI), and CS_CAC (concatenating scheme with HI_CAC). Similarly, TEA_Base and TEA_CAC represent the thread context-aware scheme with HI and the thread context-aware scheme with

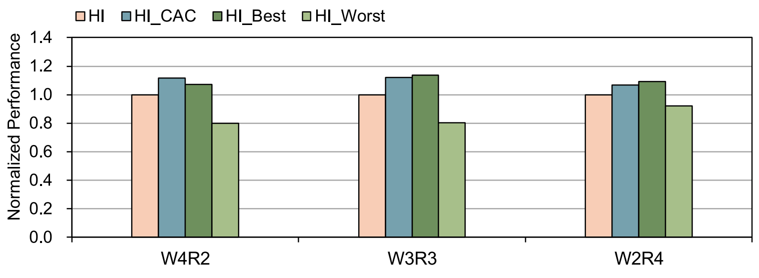


Fig. 9. Normalized performance of concatenating scheme varying the number of register ID bits used for computing a cache index.

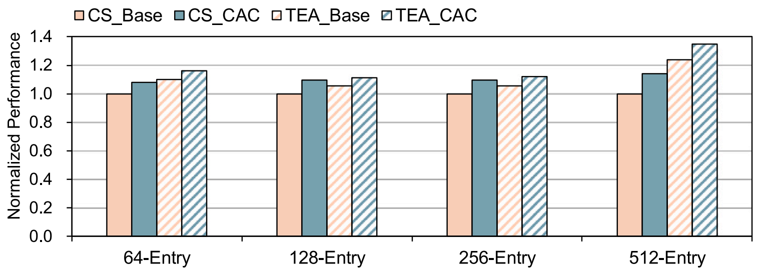


Fig. 10. Normalized performance of four different configurations varying the number of cache entries.

HI_CAC, respectively. The performance of each assignment scheme is normalized to that of CS_Base scheme, with the baseline cache entry size configuration (*i.e.,* 256 entries). As shown in the figure, the performance is best with the largest cache size, which is attributed to the reduction of cache conflicts. With different sizes of register cache (64, 128, 256, and 512 entries), CS_CAC shows performance improvements of 7.8%, 9.5%, 9.6%, and 14.3% over CS_Base, respectively. Also, TEA_CAC enhances performance by 5.7%, 5.4%, 5.9%, and 9.1% with different sizes of register cache (64, 128, 256, and 512 entries), respectively. In summary, the HI_CAC policy effectively improves performance compared to the HI for register caches of any different size. This demonstrates that our proposed compiler optimization can effectively reduce cache conflicts and improve the performance of HI-RF across various register cache sizes.

6. Related work

This paper introduces a compiler optimization scheme for HI-RF to reduce cache conflicts. To the best of our knowledge, this is the first work that uncovers the impact of register assignment on the HI-RF architecture and proposes the optimized register assignment scheme of a compiler for HI-RF. In this section, we discuss prior work related to NVM-based register files and GPU register file design with compiler optimization.

6.1. NVM-based register files

Register files on GPUs have traditionally been designed using SRAM. However, SRAM-based register files have some inherent limitations, including low density and high leakage power consumption [6,40,41]. In response to these challenges, researchers explored various novel memory technologies to design more energy-efficient register files for GPUs. In this section, we introduce some approaches that utilize NVM technologies for architecting GPU register files. Goswami et al. introduced STT-MRAM (spin-torque-transfer RAM) based register files [40]. Since STT-MRAM performs write operations with less power than SRAM, it can reduce the power consumption of the original SRAM-based register files. Mittal et al. proposed a novel design for GPU register files that utilizes SOT-RAM (Spin-Orbit Torque RAM) [41]. SOT-RAM based register files improve energy efficiency compared to SRAM-based register files. A hybrid register file design has been proposed to further reduce the energy consumption of these NVM-based register files and mitigate the long write latency of them [7]. In this approach, STT-MRAM serves as the main register file, with an NVM-based write buffer. This integration of fast memory as a write buffer significantly reduces the write latency associated with previously suggested NVM-based register files, thus enhancing their performance and efficiency. Jeon et al. introduced the HI-RF design, which combines an SRAM-based register cache with an STT-MRAM register file [10]. This architecture aims to reduce the number of write operations to the STT-MRAM registers, resulting in significant improvements in energy efficiency and mitigating issues related to the write endurance of STT-MRAM. CAC's primary goal is to reduce cache conflicts within the register cache of HI-RF, ultimately resulting in reduced energy consumption.

6.2. Compiler-driven register file optimization

Several prior work leverages compiler optimization techniques to design energy-efficient GPU register files. Esfeden et al. proposed breathing operand windows (BOW) architecture, which exploits a high locality of register accesses [39]. This paper revealed that the same registers are repeatedly accessed within a short instruction window. Based on this observation, BOW forwards data that can be reused in subsequent instructions and bypasses unnecessary register writes with the liveness analysis of the compiler. Jeon et al. proposed GPU register file virtualization [18]. This paper revealed that the number of live registers is much less than the total number of registers for kernel execution. With register lifetime analysis of the compiler, the proposed method manages registers by releasing dead registers, which consequently minimizes unnecessary demand for physical registers. Oh et al. proposed the compiler-assisted HI-RF (CASH-RF) architecture. CASH-RF uses the compiler analysis to detect unnecessary write operations [42]. With this information, CASH-RF eliminates unnecessary write-back operations to the NVM register files, which reduces the energy consumption of HI-RF. Sadrosadati et al. proposed the latency-tolerant register file (LTRF) architecture for HI-RF [43]. LTRF leverages compiler analysis to prefetch registers from the register file to the register cache. With this approach, the register file capacity can increase while reducing the energy consumption of HI-RF. Our proposed CAC suggests a novel register assignment scheme of the compiler for HI-RF architecture. As such, these introduced approaches are orthogonal to our technique, and they can be combined with CAC to further optimize register file energy consumption.

7. Conclusion

This paper presents a novel compiler optimization scheme for HI-RF aimed at reducing cache conflicts within the register cache. Based on our observation, the performance of GPUs equipped with HI-RF varies depending on how the register IDs are assigned by compilers. We introduce the concept of critical sets, denoting sets of registers that share identical LSBs in their register IDs. Our proposed compiler, CAC, takes these critical sets into consideration when assigning register IDs to alleviate cache conflicts resulting from the grouping of write operations on specific register sets. To achieve this, CAC employs destination counts as an approximation of the number of write operations on registers. Our experimental results demonstrate that the proposed compiler shows performance improvement of 11.1% and 5.9% while reducing cache conflicts by 6.2% and 2.5% over the baseline, with the concatenating scheme and the thread context-aware scheme, respectively. Also, CAC shows 73.1 percentage points and 72.9 percentage points lower energy consumption than SRAM with the concatenating scheme and the thread context-aware scheme, respectively.

CRediT authorship contribution statement

Eunbi Jeong: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Resources, Methodology, Conceptualization. **Eun Seong Park:** Software, Data curation. **Gunjae Koo:** Writing – review & editing, Writing – original draft, Funding acquisition, Conceptualization. **Yunho Oh:** Writing – review & editing, Writing – original draft, Supervision, Methodology, Funding acquisition, Conceptualization. **Myung Kuk Yoon:** Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Resources, Methodology, Funding acquisition, Conceptualization.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Myung Kuk Yoon reports financial support was provided by Institute for Information & Communications Technology Planning & Evaluation. The corresponding author collaborates with professors from Yonsei University and Korea University in South Korea, as well as the University of Illinois Urbana-Champaign in the USA. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The authors do not have permission to share data.

Acknowledgments

This work was supported in part by the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (MSIT) of the Government of South Korea (NRF-2021R1C1C1012172), in part by the Ministry of Education of the Republic of Korea and the National Research Foundation of Korea under Grant NRF-2022R1C1C1011021, and in part by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. RS-2022-00155966, Artificial Intelligence Convergence Innovation Human Resources Development (Ewha Womans University)). Yunho Oh and Myung Kuk Yoon are the corresponding authors.

References

[1] L.B. Soares, M.M.A. da Rosa, C.M. Diniz, E.A.C. da Costa, S. Bampi, Design methodology to explore hybrid approximate adders for energy-efficient image and video processing accelerators, IEEE Trans. Circuits Syst. I. Regul. Pap. 66 (6) (2019) 2137–2150.
[2] J. Li, B. Xie, Q. Fang, B. Liu, Y. Liu, P.K. Liaw, High-throughput simulation combined machine learning search for optimum elemental composition in medium entropy alloy, J. Mater. Sci. Technol. 68 (2021) 70–75.
[3] S. Bavikadi, A. Dhavlle, A. Ganguly, A. Haridass, H. Hendy, C. Merkel, V.J. Reddi, P.R. Sutradhar, A. Joseph, S.M.P. Dinakarrao, A survey on machine learning accelerators and evolutionary hardware platforms, IEEE Des. Test 39 (3) (2022) 91–116.
[4] V. Kandiah, S. Peverelle, M. Khairy, J. Pan, A. Manjunath, T.G. Rogers, T.M. Aamodt, N. Hardavellas, AccelWattch: A power modeling framework for modern GPUs, in: MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture, 2021, pp. 738–753.
[5] S. Mittal, A survey of techniques for architecting and managing GPU register file, IEEE Trans. Parallel Distrib. Syst. 28 (1) (2016) 16–28.
[6] K. Korgaonkar, I. Bhati, H. Liu, J. Gaur, S. Manipatruni, S. Subramoney, T. Karnik, S. Swanson, I. Young, H. Wang, Density tradeoffs of non-volatile memory as a replacement for SRAM based last level cache, in: 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture, ISCA, IEEE, 2018, pp. 315–327.
[7] G. Li, X. Chen, G. Sun, H. Hoffmann, Y. Liu, Y. Wang, H. Yang, A STT-ram-based low-power hybrid register file for GPGPUs, in: Proceedings of the 52nd Annual Design Automation Conference, 2015, pp. 1–6.
[8] X. Liu, M. Mao, X. Bi, H. Li, Y. Chen, An efficient STT-RAM-based register file in GPU architectures, in: The 20th Asia and South Pacific Design Automation Conference, IEEE, 2015, pp. 490–495.
[9] S. Badri, M. Saini, N. Goel, An efficient NVM-based architecture for intermittent computing under energy constraints, IEEE Trans. Very Large Scale Integr. (VLSI) Syst. (2023).
[10] W. Jeon, J.H. Park, Y. Kim, G. Koo, W.W. Ro, Hi-end: Hierarchical, endurance-aware STT-MRAM-based register file for energy-efficient GPUs, IEEE Access 8 (2020) 127768–127780.
[11] M. Rhu, M. Sullivan, J. Leng, M. Erez, A locality-aware memory hierarchy for energy-efficient GPU architectures, in: Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture, 2013, pp. 86–98.
[12] I. Jeong, Y. Oh, W.W. Ro, M.K. Yoon, TEA-RC: Thread context-aware register cache for GPUs, IEEE Access 10 (2022) 82049–82062.
[13] M. Khairy, Z. Shen, T.M. Aamodt, T.G. Rogers, Accel-sim: An extensible simulation framework for validated GPU modeling, in: 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture, ISCA, IEEE, 2020, pp. 473–486.
[14] X. Dong, C. Xu, Y. Xie, N.P. Jouppi, Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 31 (7) (2012) 994–1007.
[15] NVIDIA, NVIDIA CUDA toolkit documentation, 2018, URL https://docs.nvidia.com/cuda/archive/9.0/parallel-thread-execution/index.html.
[16] NVIDIA, NVIDIA A100 tensor core GPU architecture, 2020, URL https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf.
[17] S. Lee, K. Kim, G. Koo, H. Jeon, W.W. Ro, M. Annavaram, Warped-compression: Enabling power efficient GPUs through register compression, ACM SIGARCH Comput. Archit. News 43 (3S) (2015) 502–514.
[18] H. Jeon, G.S. Ravi, N.S. Kim, M. Annavaram, GPU register file virtualization, in: Proceedings of the 48th International Symposium on Microarchitecture, 2015, pp. 420–432.
[19] S. Lee, K. Kim, G. Koo, H. Jeon, M. Annavaram, W.W. Ro, Improving energy efficiency of GPUs through data compression and compressed execution, IEEE Trans. Comput. 66 (5) (2016) 834–847.
[20] J. Choquette, Nvidia hopper h100 gpu: Scaling performance, IEEE Micro (2023).
[21] X. Liu, M. Mao, X. Bi, H. Li, Y. Chen, Exploring applications of STT-RAM in GPU architectures, IEEE Trans. Circuits Syst. I. Regul. Pap. 68 (1) (2020) 238–249.
[22] Y. Zhang, X. Wang, H. Li, Y. Chen, STT-RAM cell optimization considering MTJ and CMOS variations, IEEE Trans. Magn. 47 (10) (2011) 2962–2965.
[23] V.S.D. Eswar, K.D. Bhavani, D. Nandan, Study on energy reduction techniques in STT-RAM, J. Phys.: Conf. Ser. 1714 (1) (2021) 012041.
[24] C. Lin, S. Kang, Y. Wang, K. Lee, X. Zhu, W. Chen, X. Li, W. Hsu, Y. Kao, M. Liu, et al., 45Nm low power CMOS logic compatible embedded STT MRAM utilizing a reverse-connection 1T/1MTJ cell, in: 2009 IEEE International Electron Devices Meeting, IEDM, IEEE, 2009, pp. 1–4.
[25] H. Zhang, X. Chen, N. Xiao, L. Wang, F. Liu, W. Chen, Z. Chen, Shielding STT-ram based register files on GPUs against read disturbance, ACM J. Emerg. Technol. Comput. Syst. (JETC) 13 (2) (2016) 1–17.
[26] Y. Zhang, L. Zhang, Y. Chen, MLC STT-RAM design considering probabilistic and asymmetric MTJ switching, in: 2013 IEEE International Symposium on Circuits and Systems, ISCAS, IEEE, 2013, pp. 113–116.

[27] W. Kim, J. Jeong, Y. Kim, W. Lim, J. Kim, J. Park, H. Shin, Y. Park, K. Kim, S. Park, et al., Extended scalability of perpendicular STT-MRAM towards sub-20nm MTJ node, in: 2011 International Electron Devices Meeting, IEEE, 2011, pp. 24.1.1–24.1.4.

[28] A. Inci, M.M. Isgenc, D. Marculescu, DeepNVM++: Cross-layer modeling and optimization framework of nonvolatile memories for deep learning, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 41 (10) (2021) 3426–3437.

[29] H. Zhang, X. Chen, N. Xiao, F. Liu, Architecting energy-efficient STT-ram based register file on GPGPUs via delta compression, in: Proceedings of the 53rd Annual Design Automation Conference, 2016, pp. 1–6.

[30] NVIDA, CUDA compiler driver NVCC, 2023, URL https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/.

[31] NVIDA, NVIDIA tesla V100 GPU architecutre, 2017, URL https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[32] A. Danalis, G. Marin, C. McCurdy, J.S. Meredith, P.C. Roth, K. Spafford, V. Tipparaju, J.S. Vetter, The scalable heterogeneous computing (SHOC) benchmark suite, in: GPGPU, 2010.

[33] S. Che, M. Boyer, J. Meng, D. Tarjan, J.W. Sheaffer, S.-H. Lee, K. Skadron, Rodinia: A benchmark suite for heterogeneous computing, in: IISWC, 2009.

[34] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, J. Cavazos, Auto-tuning a high-level language targeted to GPU codes, in: InPar, 2012.

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

[36] NVIDA, NVIDIA CUDA toolkit, 2023, URL https://developer.nvidia.com/cuda-toolkit.

[37] S. Subha, An energy saving model for fully associative cache, in: 3rd International Conference on Trendz in Information Sciences & Computing, TISC2011, IEEE, 2011, pp. 141–143.

[38] Z. Jia, M. Maggioni, J. Smith, D.P. Scarpazza, Dissecting the NVidia turing T4 GPU via microbenchmarking, 2019, arXiv preprint arXiv:1903.07486.

[39] H.A. Esfeden, A. Abdolrashidi, S. Rahman, D. Wong, N. Abu-Ghazaleh, BOW: Breathing operand windows to exploit bypassing in GPUs, in: 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO, IEEE, 2020, pp. 996–1008.

[40] N. Goswami, B. Cao, T. Li, Power-performance co-optimization of throughput core architecture using resistive memory, in: 2013 IEEE 19th International Symposium on High Performance Computer Architecture, HPCA, IEEE, 2013, pp. 342–353.

[41] S. Mittal, R. Bishnoi, F. Oboril, H. Wang, M. Tahoori, A. Jog, J.S. Vetter, Architecting SOT-RAM based GPU register file, in: 2017 IEEE Computer Society Annual Symposium on VLSI, ISVLSI, IEEE, 2017, pp. 38–44.

[42] Y. Oh, I. Jeong, W.W. Ro, M.K. Yoon, CASH-RF: A compiler-assisted hierarchical register file in GPUs, IEEE Embedded Syst. Lett. 14 (4) (2022) 187–190.

[43] M. Sadrosadati, A. Mirhosseini, S.B. Ehsani, H. Sarbazi-Azad, M. Drumond, B. Falsafi, R. Ausavarungnirun, O. Mutlu, LTRF: Enabling high-capacity register files for GPUs via hardware/software cooperative register prefetching, ACM SIGPLAN Not. 53 (2) (2018) 489–502.

Eunbi Jeong is an undergraduate student in the Department of Computer Science and Engineering, Ewha Womans University, Seoul, Korea. She is currently an undergraduate intern in the Intelligence System and Parallel Computer Architecture Laboratory (IP-CAL) at the same university, where she plans to pursue the M.S. degree. Her research interests include GPU microarchitecture, and deep learning accelerator.

Eun Seong Park is currently working as an undergraduate researcher in the intelligence system and parallel computer architecture lab in the Department of Computer Science and Engineering at Ewha Womans University, Seoul, Korea. Her research interests include GPU architecture and artificial intelligence.

Gunjae Koo received the B.S. and M.S. degrees in electrical and computer engineering from Seoul National University, in 2001 and 2003, respectively, and the Ph.D. degree in electrical engineering from the University of Southern California, in 2018. He is currently an Associate Professor with the Department of Computer Science and Engineering, Korea University. Prior to joining Korea University, he was an Assistant Professor with Hongik University. He was a Senior Research Engineer with LG Electronics and a Research Intern with Intel. His research interests include computer system architecture and span parallel processor architecture, storage and memory systems, accelerators, and secure processor architecture.

Yunho Oh received the B.S., M.S., and Ph.D. degrees in the School of Electrical and Electronic Engineering at Yonsei University, Seoul, Korea, in 2009, 2011, and 2018, respectively. He had worked as a software engineer in Mobile Communications Business, Samsung Electronics. Currently, he is working as an assistant professor in the School of Electrical Engineering at Korea University. Prior to joining Korea University, I was an assistant professor in the Department of Electronic and Electrical Engineering at SungKyunKwan University (SKKU). From March 2019 to Feburary 2021, I worked as a postdoctoral researcher in Parallel Systems Architecture Lab (PARSA) at EPFL, Switzerland. His research interests include hardware and software architectures for energy-efficient datacenters, processor architectures (CPUs, GPUs, and neural network accelerators), in-storage processing, memory system, and high-performance computing.

Myung Kuk Yoon received his BS degree in Computer Engineering and Computational Mathematics from Washington State University (WSU), Pullman, Washington, USA, in 2011, and his Ph.D. degree in Electrical and Electronic Engineering from Yonsei University, Seoul, Korea, in 2018. He is currently working as an assistant professor with the Department of Computer Science and Engineering, Ewha Womans University. Prior to joining Ewha Womans University, he worked as a software developer at Samsung Inc. His research interests include GPU micro-architecture, machine learning accelerators, and parallel programming.