Lightweight Register File Caching in Collector Units for GPUs

Mojtaba Abaie Shoushtary, Jose Maria Arnau, Jordi Tubella Murgadas, Antonio Gonzalez
Polytechnic University of Catalonia
Barcelona, Spain

ABSTRACT

Modern GPUs benefit from a sizable Register File (RF) to provide fine-grained thread switching. As the RF is huge and accessed frequently, it consumes a considerable share of the dynamic energy of the GPU. Designing a large, high-throughput RF with low energy consumption and area for GPUs is challenging. In this paper, an energy-efficient hierarchical RF design for GPUs, called Malekeh, is introduced. Malekeh keeps registers in energy-efficient small caches and maximizes cache efficacy by using lightweight policies and supporting adaptive algorithms. The policies' effectiveness is improved by leveraging register reuse distance information provided by the compiler as a hint. Malekeh reduces the RF reads by 48.5% and dynamic energy by 29.1%. It also improves performance by 9.6% with a negligible overhead of 0.04% in area.

CCS CONCEPTS

• **Computer systems organization → Parallel architectures**.

KEYWORDS

GPU, Cache, Register File, Compiler Guided

1 INTRODUCTION

Modern Graphics Processing Units (GPUs) rely on swift thread switching to hide stalls. Conventionally, GPUs have been using large Register Files (RFs) to implement fast thread switching.

Since GPUs' RF is huge and accessed frequently, it consumes considerable dynamic energy. For example, RF consumes 24% of the dynamic power of an NVIDIA Volta GV100[6]. It makes optimizing the RF energy efficiency appealing.

Previous works have proposed caching mechanisms to reduce the energy consumption of the RF in GPUs[3–5, 11]. An RF cache is smaller than the RF banks and is closer to the consumer pipelines. Therefore, provided a high hit ratio, the cache potentially saves energy and improves performance.

In this work, an RF cache mechanism called Malekeh is introduced. Malekeh is a low-cost, hierarchical RF architecture caching registers within existing operand buffering structures called Operand Collector Units (OCUs). Conventionally, OCUs keep read operands before dispatching the instruction, though, not function as a cache. Malekeh turns the OCUs into a cache with minor changes to capture the temporal reuses of the registers with negligible overhead and avoid accessing the power-hungry RF banks.

Malekeh does not increase the number of OCUs to keep the overhead low. Increasing the number of OCUs would lead to high overheads due to: a) extra OCUs and b) a bigger OCU–bank interconnection network. Having a low OCU count keeps the overhead small but makes capturing locality while sustaining performance challenging since OCUs are time-shared by different warps, each with its own private register set.

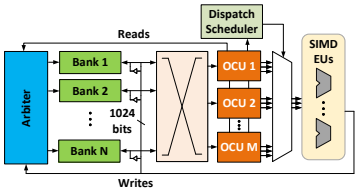

Figure 1: RF microarchitecture in GPUs

Which particular warp uses each OCU at a given point in time depends on the instruction scheduling and OCU allocation policies. Besides, Malakeh manages each OCU as a cache; therefore, effective allocation, replacement, and write policies are crucial. All these policies and their synergy determine the hit ratio, energy-saving, and performance. Designing effective policies for such a system that provide a high hit ratio and performance while being lightweight is indispensable and challenging. Malekeh addresses this problem with compiler-guided and lightweight policies tailored to maximize energy-saving and performance.

Malekeh uses register reuse distance as a hint to implement effective management policies for the different components involved. The reuse distance is computed by the compiler and passed to the hardware. Our analysis shows that reuse distance significantly improves the effectiveness of the policies, although passing its exact value to the hardware would be very costly. Therefore, a practical approximation for reuse distance is adopted which performs almost as well as the precise value.

In addition, Malekeh benefits from a dynamic algorithm that delays replacing certain OCUs that can potentially increase hit ratio and performance. The algorithm reacts to runtime behavior coming out of code characteristics. The goal of this dynamic algorithm is to unleash the full available potential.

2 BACKGROUND

Conventional GPUs' RF contains multiple single or double-ported large banks instead of monolithic multi-ported ones with some OCUs to buffer read registers before dispatching the instruction to the EUs, Figure 1.

OCU stores some metadata, such as Warp ID, in addition to a valid bit, a ready bit, and a data field per source operand of the issued instruction. The data field stores the fetched register value for any source operand whose valid bit is set. The ready bit indicates that the register value has been received. Typically, a warp has 32 threads, each accessing 32-bit operands, so the data width needs 1024 bits.

When the instruction issue scheduler issues a new instruction, the OCU allocator determines the target OCU and reserves it until all the source operands are ready. At this moment, the instruction

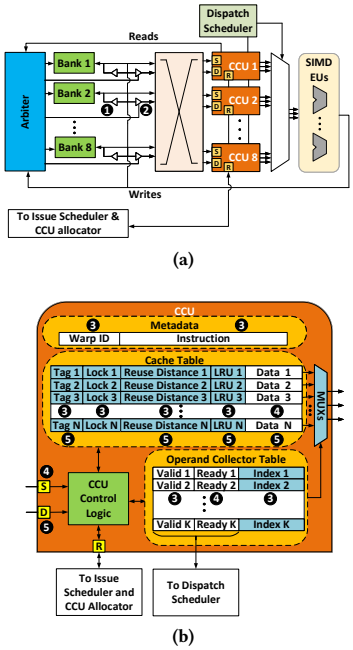

Figure 2: Malekeh microarchitecture: (a) RF microarchitecture; (b) Caching Collector Units (CCU)

in the OCU is eligible to dispatch. The OCU is released once its instruction is dispatched.

The number of source operand slots in each OCU depends on the ISA. We use Turing ISA that includes HMMA instructions requiring 7 source operand slots at maximum[9].

3 MICROARCHITECTURE

Malekeh requires minor microarchitecture modifications and ISA extensions to pass information to the hardware, as explained in this section.

3.1 Register File

There are four main differences compared to the baseline architecture shown in Figure 2a. 1) replacing OCUs with a unit with caching capabilities, named Caching Collector Units(CCUs), 2) providing a write pathway through the crossbar to a dedicated port named D, 3) adding tri-state buffer ❷ beside buffer ❶ controlled by the arbiter to filter superfluous writes for more energy saving, and 4) providing Warp ID, and the minimum reuse distance of the stored live values to the issue scheduler/CCU allocator through port R.

3.2 Caching Collector Unit (CCU)

To exploit the locality in the RF accesses, OCUs are extended with cache capabilities while keeping their functionality by adding minimal extra fields and a control unit, Figure 2b.

Each CCU has a port S to receive the read source operand values, a port D to receive the value written to the instruction's destination, and a port R to exchange information between the CCU and the Issue scheduler/CCU allocator.

In addition, it contains Metadata, as the baseline, a Cache Table (CT), an Operand Collector Table (OCT), some Multiplexers (MUXs) to deliver source operands to the EU's input latches, and a CCU Control Logic.

The CT is the component providing the caching capability. It reuses already existing data fields and adds a tag array to identify the cached registers; a lock bit to avoid replacing the slots needed when dispatching the instruction; a reuse distance field computed by the compiler and meant to guide cache policies, and issue scheduler/CCU allocator; and Least Recently Used (LRU) priority information.

The overhead of extra fields is small because Malekeh reuses data fields already existing in the baseline, the largest storage in CCU with 1024 bits. The tag width is capped by Cuda to only 1 byte, the lock is 1 bit and the reuse distance is approximated by only 1 bit showing the reuse is far or near compared to a threshold we call RTHLD. The LRU is also only 3 bits as we empirically discovered that more than 8 slots provide diminishing returns while adding to the overhead.

The OCT makes CCUs still function as OCUs. Similar to the baseline, they have valid and ready bits, although augmented with index fields referring to the CT entries. The index field is only 3 bits, assuming the CT size of 8, and avoids storing redundant data.

The CCU Control Logic governs the CCU during CCU allocation, source operand delivery, and destination write arrival.

The parts marked by ❸ are involved in CCU allocation which has the following steps: 1) flushing CCU if the issued instruction's warp id mismatches that of the metadata; 2) updating metadata; 3) looking up all the source operands; 4) replacing and allocating one slot to a source operand when it misses in the cache; 5) locking all the source operands; 6) setting their reuse distance by encoded information in the issued instruction; 7) updating LRU values; 8) setting valid and index fields in OCT for all the source operands; 9) setting the ready bits for operands hit in the cache; and 10) requesting missed source operands from RF banks.

The components denoted by ❹ are involved in source operand delivery. Once a source operand value is received over port S, it fills the corresponding slot in CT, and its affiliated ready bit in OCT will also be set.

Block marked with ❺ are the ones that contribute to handling a write request received at port D. When a write request arrives, the CT is looked up first. In case of a miss, one entry is replaced and allocated to the written value. Unlike source operands that receive allocated slots after issue, slot allocation for writes is postponed to after writeback when their value is produced. This helps Malekeh to manage its small storage efficiently and to keep the overhead low. The detailed policies used in the cache are explained in the next section.

4 CACHE POLICIES

This section elaborates on cache policies that exploit the register reuse distance information provided by the compiler. Our analysis shows that the reuse distance improves cache policies' effectiveness significantly, so a practical binary approximation is proposed to provide this information with minimal overhead. The binary approximation considers reuses above a predefined threshold, RTHLD,

as far, and those below this threshold as near. The following sections explain the cache replacement and write policies in particular.

4.1 Replacement Policy

The cache replacement policy uses the lock bit, reuse distance, and LRU fields to select the replaced entry.

First, it excludes all the registers having the lock bit set. These are the registers used as a source operand in the current instruction occupying the CCU. Replacing them would be detrimental because all the source operands are needed when the instruction is dispatched to the execution units.

After excluding the locked registers, the replacement policy selects randomly among those registers with a far reuse distance, if any exists. If all registers are marked as near, the replacement candidate will be according to the LRU policy.

4.2 Write Policy

The destination register is always updated in the RF banks, but it is updated in the CCU based on its reuse distance. Writing to the RF banks for all the write requests allows replacing any CCU and flushing its cache at any time.

Multiple write requests can arrive at a CCU at the same time when different instructions of the same warp have different latencies, and they finalize execution at the same time. Our analysis showed that the benefits in performance and hit ratio of extra ports are minimal, so Malekeh includes only one write port in the CCU for destination operands. In case of multiple write requests, the write policy selects the first write slot containing a near register and discards the rest.

It is also possible that a write corresponds to a warp not having any CCU allocated. It occurs when the write belongs to an instruction of a warp whose CCU has been replaced by another warp by the time the write happens. In that case, the write request would only be directed to the RF banks.

5 ISSUE SCHEDULING POLICY

Malekeh's issue scheduling policy determines the warps selected to issue an instruction at each cycle. Then, the CCU allocation policy determines their target CCUs. The issue will be stalled if no CCU can be allocated to the selected warps.

The issue scheduler is aware of the warp ids of the instructions occupying each CCU and prioritizes the warps having data in any CCUs over the others. This way, the chances of reuse are increased because when a CCU is allocated to a new different warp, its CT is flushed. Then, the future reuses to these flushed values must be fetched from the RF banks.

Unlike the two-level issue schedulers proposed in the literature, Malekeh does not limit the number of candidate warps. It only partitions the warps into two categories, the warps having data in the CCUs and the rest, and prioritizes one category over the other. Within each category, the older warps have higher priority since it favors the same warp unless having no ready instruction. This way, the contention with younger warps is reduced which leads to less cache trashing and potentially better performance for sensitive applications.

6 CCU ALLOCATION POLICY

The CCU allocation policy determines the target CCU for each warp selected by the issue scheduler. If there is currently a CCU allocated to the same warp, it chooses this CCU; otherwise, a CCU will be allocated to this warp, among those CCUs whose instructions have already been dispatched, and its CT will be flushed.

Malekeh reduces this replacement penalty by first replacing the CCU containing only far values. Then, if there is at least one near value in the CCU, Malekeh tries to wait for some cycles. During waiting time, the instructions of the same warp may finish, and awake instructions that were not ready. In the next cycle, these awaken instructions can reuse the data in the CCU whose CT would have been flushed if we had not waited.

Long waiting may be detrimental to performance. Therefore, a threshold, called STHLD, is introduced to limit the number of waiting cycles. A local counter per core will be compared to STHLD when CT has at least one near value. Only when the counter is higher than STHLD will the CCU be replaced. Otherwise, the CCU allocator is stalled and the counter is incremented.

It is possible to delay issuing instructions to gain hit ratio while not harming performance because the ready instructions may not be on the execution's critical path. However, identifying the execution's critical path by the compiler or during runtime is unfeasible since it not only depends on data dependencies but also on hardware resources. Malekeh uses a simple heuristic based on counting the stall cycles and limiting them to a threshold (STHLD).

The higher the STHLD, the more the CCU allocator will wait and the more bubbles it will generate, which on the one hand, improves hit ratio, but damages performance.

The optimum value for STHLD maximizes both performance and the hit ratio. This optimum value depends on the characteristics of the code being executed. Malekeh uses an adaptive algorithm to find the optimum value at run time, section 7.

7 ADAPTIVE ALGORITHM TO SET STHLD

Malekeh uses an adaptive scheme to dynamically set the STHLD value which controls the CCU allocation policy's efficacy. STHLD affects both performance and hit ratio. Finding the optimum STHLD depends on the characteristics of the running application over time. Fortunately, since GPU applications are regular, and cores are homogeneous, the application characteristics remain similar for long intervals until the application phase changes.

Motivated by that, Malekeh partitions the execution time into equal intervals and set the STHLD at the end of each interval. It uses the IPC measured in the previous and current intervals to modify the STHLD for the next interval. After several intervals, the STHLD converges to its optimum point.

Figure 3a depicts the expected hit ratio as a function of STHLD. As the STHLD increases, Malekeh delays replacing a CCU more. During that period, some values might be reused that would have been flushed otherwise. Therefore, the hit ratio monotonically increases until reaching a saturation point. The saturation point is the maximum reuse existing in the application.

Figure 3b shows the expected IPC as a function of STHLD. As the STHLD increases, more accesses will be serviced directly by the cache. Therefore, the IPC increases slightly. However, after some

Figure 3: Expected IPC and hit ratio when changing STHLD: (a) Hit ratio, (b) IPC

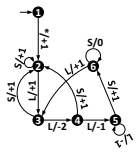

Figure 4: Adaptive algorithm to dynamically set STHLD

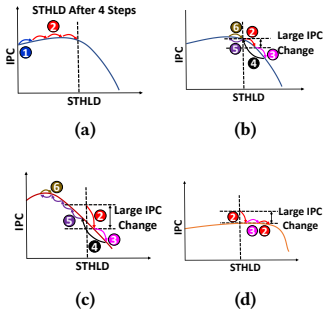

Figure 5: Example of setting STHLD: (a) Initial curve and first four steps, (b) Next steps when staying in the same curve, (c) Next steps when transitioning to a curve with a narrower flat region, (c) Next steps when transitioning to a curve with a wider flat region.

point, the IPC begins to drop because of so many stalls generated. We refer to this point as the knee point. Before the knee point is called the flat region, and after that is called the steep region.

Based on these expectations, we propose an adaptive algorithm to dynamically set the value of STHLD, a finite state machine depicted in Figure 4. The state machine transitions are based on the relative differences between the IPC of the current and the previous interval. Small and large relative differences are denoted respectively by S and L. If a transition happens regardless of the difference, it is shown by an asterisk. The transitions trigger an increase or decrease of STHLD by a delta shown on the transition edges.

The adaptive algorithm starts from state 1 and remains in state 6 until a large IPC change is detected.

The deltas applied by this state machine for a sample code are shown in Figure 5. In this figure, the corresponding state is shown by a number, and the deltas are represented by the arrow's length. The arrows showing steps taken in each state are colored the same as the circled number showing the corresponding state. Figure 5 shows that the adaptive algorithm is designed to walk on the curve and, based on the IPC fluctuation, modify the STHLD until it converges to the knee point. The knee point is the optimum STHLD.

In this example, the algorithm started on the curve shown in Figure 5a and after four intervals with a small change in IPC, a large change is detected. The large change could be due to moving to the steep region of the same curve, Fig 5b, or a change in the application phase that changes the curve to the curves shown in Figs 5c or 5d for the following next intervals.

The curve changes as the phase of the application changes because the characteristics of the application differ. The new curve may have a narrower (Figure 5c) or wider (Figure 5d) flat region. In case of moving to the curve with a narrower flat region, the current STHLD will be in a steep region and the adaptive algorithm will reduce it to reach the STHLD corresponding to the knee point of the IPC curve. On the other hand, in the case of moving to the curve with a long flat region, the current STHLD will be in the flat region, so the adaptive algorithm will increase the STHLD to approach the point leading to the knee point of the IPC curve.

In case of a large change, the dynamic algorithm takes a speculative move by increasing STHLD to gain more hit ratio and moves to state 3. If the new phase corresponds to the curve shown in Fig 5d, the speculative move was correct and we benefit from the increased IPC and hit ratio due to a higher STHLD in that interval. On the other hand, if the phase has the same curve, Fig 5b, or the curve shown in Fig 5c, we lose performance since the speculative move was in the steep region where performance decreases when STHLD is increased, but only for one interval. After realizing that this speculative step is detrimental, the scheme decreases STHLD and after some additional steps, it converges to the optimal point.

We empirically found that an interval size of 10000 cycles provides a good trade-off between performance and hit ratio.

8 METHODOLOGY

We used Accel-sim[7] modeling a scaled-down configuration based on the Geforce RTX 2060[8] GPU with the parameters shown in Table 1. We scaled down the number of SMs, the size of L2, and the number of memory channels by one-third.

Table 1: Baseline GPU configuration used in this work

#SMs	10
#Threads/Warps per SM	1024 / 32
RF Size	256 KB
Issue Scheduler	GTO[10]
L2 Size	1 MB
L1/shared memory	64 KB

The benchmarks are selected from Rodinia[2] and Deepbench[1]. The former is representative of general-purpose computing applications, whereas the latter consists of modern deep learning workloads.

We used accel-sim in trace mode and annotated the traces with precise reuse distances and their binary approximation to evaluate the binary approximation effectiveness.

To evaluate the dynamic energy of the RF, we extended the power model provided in Accelwattch[6]. The model includes the arbiter, crossbar, RF banks, and CCUs.

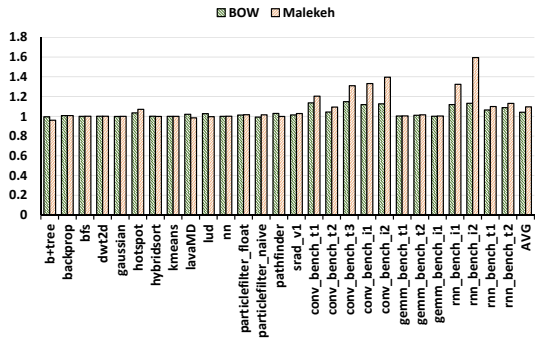

Figure 6: IPC normalized to the baseline

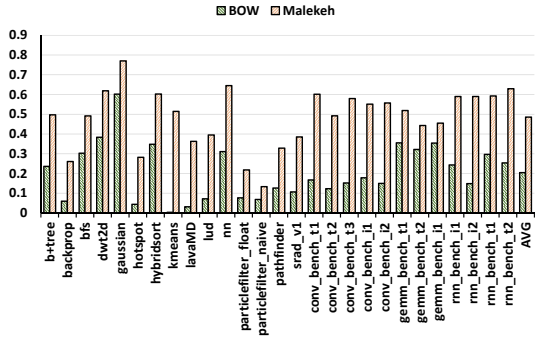

Figure 7: RF cache hit ratio

We compare our work against BOW[3], a state-of-the-art proposal for RF caching in GPUs. BOW replaces OCUs with buffers that keep the source operand values of instructions in a sliding window. It proposes forwarding register values between the instructions in the sliding window instead of fetching them from RF banks to save energy.

9 EVALUATION

In this section, we analyze the benefits and overheads of Malekeh and compare it with BOW[3] which has a sliding window of three instructions.

9.1 Performance

On the one hand, Malekeh may improve performance since the registers found in the CCUs do not need to be retrieved again from the RF, which is faster and also reduces the conflicts in the ports of the RF banks. On the other hand, some policies used in Malekeh have the potential to damage performance. In particular, when the issue is stalled even if there are ready instructions, for the sake of better reuse in the RF cache, performance may be penalized.

Figure 6 shows that Malekeh not only does sustain the IPC of the baseline in all the benchmarks except b+tree, but it also improves their IPC in many cases. It increases the IPC of the baseline by 9.6% on average. The maximum improvement is achieved for rnn_bench_i2 which is 59.5%. On the other hand, for the only benchmark in which IPC is reduced, b+tree, the IPC drop is less than 4%.

Malekeh improves BOW's IPC by 5.4% on average and up to 46.3% for rnn_bench_i2. On the other hand, BOW achieves a maximum IPC improvement of 14.9% for conv_bench_t3, but still much lower than that of Malakeh (36%).

Malekeh improves BOW's IPC because it provides a higher hit ratio. Generally, performance and hit ratio are correlated only when there are no bottlenecks in other pipeline stages. In those cases, Malekeh outperforms BOW as its hit ratio improvement is significant.

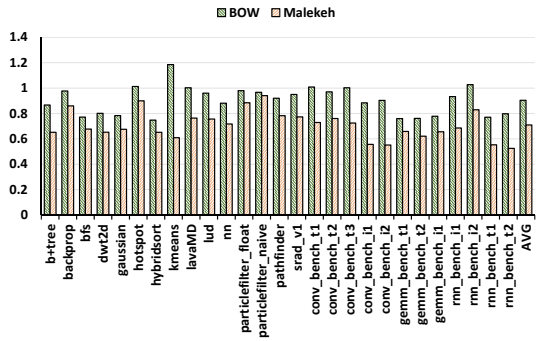

Figure 8: RF dynamic energy normalized to the baseline

9.2 Read Hit Ratio

Figure 7 depicts the achieved read hit ratio of the RF cache. On average, Malakeh gets 48.5% of the register source operands from the cache. The maximum hit ratio is for gaussian, which is about 77%, and the minimum is for particlefilter_naive, which is about 13.3%. Malekeh has a higher hit ratio than BOW by 28.1% on average and up to 51.1% for Kmeans.

Malekeh surpasses BOW due to a) better RF cache and Issue scheduling/CCU allocating coordination, b) more effective cache management policies using reuse distance as a hint, c) an efficient dynamic algorithm that maximizes both IPC and hit ratio, and d) having less frequent flushes than BOW, which needs to flush the cache every time a branch jumps to a new sliding window.

9.3 RF Dynamic Energy

Figure 8 shows the RF dynamic energy consumption normalized to the baseline. Malekeh reduces a 9.6% reduction. The highest energy saving of Malekeh is for rnn_bench_t2 with a 47.5% reduction and the lowest is for particlefilter_naive with a 5.9% reduction.

Better energy saving of Malekeh versus BOW comes out of a) a higher number of requests serviced directly by the RF cache, b) a more efficient write policy avoiding unnecessary writes with no reuse, which only consume energy without any compensations coming out of future reuses, c) keeping the cache small and managing the space efficiently by eliminating redundant values through indirect indexing d) benefiting from lightweight policies tailored to maximize hit ratio and IPC with a small overhead d) simple cache controlling mechanism using some extra fields instead of a complex

forwarding mechanism in BOW, whose complexity grows with cache size and it consumes energy even in the case of hits to move data between instructions of the sliding window.

9.4 Overheads

Malekeh is designed to keep the hardware overhead small. It neither does increase the number of OCUs nor the amount of data that they store nor the crossbar size. On the other hand, it requires some extra storage in each OCU to keep some extra control fields, which amount to 128 B per SM, or in other words, a 0.04% increase of the area of an RF with 256 KB.

The adaptive algorithm is also designed to have minimal overhead. It just stores the IPC of the last interval per chip. Therefore only one extra register is needed to store the value provided by the performance counter that measures IPC. The logic for the adaptive algorithm is a finite state machine that runs at the end of each interval. For the assumed interval size of 10000 cycles, the energy overhead of it is negligible. The overhead of adding one bit to each source and destination register is also small.

10 RELATED WORK

Gebhart et al.[4] propose an RF caching mechanism for GPUs having clustered execution pipelines. An RF cache is added to each cluster and filled only with the outputs of every operation. Malekeh's cache accommodates both source and destination registers in CCUs.

The authors extend their RF cache to be managed by the compiler[5]. A register allocation policy assigns registers to different levels of the hierarchy to optimize energy efficiency. On the other hand, Malekeh uses a more efficient hardware-controlled cache guided by the compiler. Malekeh reacts to run time events while having some future information about reuse provided by the compiler.

Asghari Esfeden et al.[3] proposed a forwarding operand mechanism implemented in the OCUs. A detailed comparison between Malakeh and this work is presented in section 9.

Sadrosadati et al.[11] propose a cache added between the RF banks and the OCUs. They prefetch the register values into the cache and move them to the OCUs when needed. Malekeh implements the cache inside the OCUs, so it avoids the overhead of moving data from the cache to the OCUs.

11 CONCLUSIONS

In this paper, we have introduced Malekeh, a novel caching scheme for the RF of GPUs that leverages the already existing storage in operand collector units. To make this caching effective, Malekeh leverages a cache-sensitive instruction scheduler and compiler information about the reuse of registers. We have shown that Malekeh provides 9.6% IPC improvement, 48.5% RF cache hit ratio, and 29.1% dynamic RF energy saving on average by introducing less than 0.04% area overhead.

ACKNOWLEDGMENTS

This work has been supported by the CoCoUnit ERC Advanced Grant of the EU's Horizon 2020 program (grant No 833057), the Spanish State Research Agency (MCIN/AEI) under grant PID2020-113172RB-I00, and the ICREA Academia program.

REFERENCES

[1] baidu. 2020. Deepbench: Benchmarking deep learning operations on different hardware. https://github.com/baidu-research/DeepBench.

[2] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*.

[3] Hodjat Asghari Esfeden, Amirali Abdolrashidi, Shafiur Rahman, Daniel Wong, and Nael Abu-Ghazaleh. 2020. BOW: Breathing Operand Windows to Exploit Bypassing in GPUs. In *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[4] Mark Gebhart, Daniel R. Johnson, David Tarjan, Stephen W. Keckler, William J. Dally, Erik Lindholm, and Kevin Skadron. 2011. Energy-efficient mechanisms for managing thread context in throughput processors. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*.

[5] Mark Gebhart, Stephen W. Keckler, and William J. Dally. 2011. A Compile-Time Managed Multi-Level Register File Hierarchy. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[6] Vijay Kandiah, Scott Peverelle, Mahmoud Khairy, Junrui Pan, Amogh Manjunath, Timothy G. Rogers, Tor M. Aamodt, and Nikos Hardavellas. 2021. AccelWattch: A Power Modeling Framework for Modern GPUs. In *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[7] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. 2020. Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling. In *Proceedings of the 47th Annual International Symposium on Computer Architecture (ISCA)*.

[8] NVIDIA. 2018. NVIDIA TURING GPU ARCHITECTURE. Retrieved Nov. 23, 2022 from https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

[9] Md Aamir Raihan, Negar Goli, and Tor M. Aamodt. 2019. Modeling Deep Learning Accelerator Enabled GPUs. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*.

[10] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. 2012. Cache-Conscious Wavefront Scheduling. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[11] Mohammad Sadrosadati, Amirhossein Mirhosseini, Seyed Borna Ehsani, Hamid Sarbazi-Azad, Mario Drumond, Babak Falsafi, Rachata Ausavarungnirun, and Onur Mutlu. 2018. LTRF: Enabling High-Capacity Register Files for GPUs via Hardware/Software Cooperative Register Prefetching. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.