Bank Stealing for a Compact and Efficient Register File Architecture in GPGPU

Naifeng Jing, Shunning Jiang, Shuang Chen, Jingjie Zhang, Li Jiang, Chao Li, Member, IEEE, and Xiaoyao Liang

Abstract—Modern general-purpose graphic processing units (GPGPUs) have emerged as pervasive alternatives for parallel high-performance computing. The extreme multithreading in modern GPGPUs demands a large register file (RF), which is typically organized into multiple banks to support the massive parallelism. Although a heavily banked structure benefits RF throughput, its associated area and energy costs with diminishing performance gains greatly limit the future RF scaling. In this paper, we propose an improved RF design with bank stealing techniques, which enable a high RF throughput with compact area. By deeply investigating the GPGPU microarchitecture, we find that the state-of-the-art RF designs’ is far from optimal due to the deficiency in bank utilization, which is the intrinsic limitation to a high RF throughput and a compact RF area. We investigate the causes for bank conflicts and identify that most conflicts can be eliminated by leveraging the fact that the highly banked RF oftentimes experiences underutilization. This is especially true in GPGPUs, where multiple ready warps are available at the scheduling stage with their operands to be wisely coordinated. In this paper, we propose two lightweight bank stealing techniques that can opportunistically fill the idle banks and register entries for better operand service. Using the proposed architecture, the average GPGPU performance can be improved under a smaller energy budget with significant area saving, which makes it promising for sustainable RF scaling.

Index Terms—Area efficiency, bank conflict, bank stealing, general purpose graphic processing unit (GPGPU), register file (RF), register liveness.

I. INTRODUCTION

MODERN general-purpose graphic processing units (GPGPUs) have emerged as pervasive alternatives for high-performance computing. GPGPU exploits the inherent massive parallelism by utilizing a large number of concurrent threads executed in single instruction multiple threads (SIMTs) manner. To hide the long memory access latency during the thread execution, the SIMT execution is featured by a fast thread switching mechanism, termed the zero-cost context switching, supported by a large-sized register file (RF), where the active contexts of the concurrent threads can be held in each streaming multiprocessor (SM).

For instance, the classic Nvidia Fermi architecture includes 15 SMs, where each SM is equipped with 32768 32-bit registers to support 1536 concurrent threads, resulting a nearly 2-MB on-chip RF storage in total in a single GPU [1]. The capacity keeps increasing in each product generation, e.g., doubled in Kepler architecture [2], in seek of even higher thread-level parallelism (TLP).

To manage such a large RF, a multibanked structure is widely used [3]–[5]. Multibanked RF sustains the multiported RF throughput by serving concurrent operand accesses as long as there are no bank conflicts. If conflict occurs, the requests have to be serialized, and the RF throughput is reduced leading to performance loss. As a result, in face of the RF capacity scaling in the future GPGPUs, adding more banks is a natural way to accommodate the increasing number of registers without worsening the bank conflict problem. Otherwise, the performance gained from higher TLP could be compromised because more threads from schedulers will be competing for an insufficient number of RF banks that aggravates the conflicts and reduces the throughput.

However, banks are not free. Adding banks requires extra column circuitry, such as sensor amplifiers, prechargers, bitline drivers, and a bundle of wires [6]. Meanwhile, the crossbar between banks and operand collectors grows significantly with the increasing number of banks. All these issues affect the RF area, timing, and power [7]. As modern GPGPUs tend to become area-limited and they have already been power-limited as pointed in [5], it is important to incorporate a suitable number of banks while seeking for more economic ways to mitigate the conflicts for the ever-increasing RF capacity.

In this paper, we explicitly address the issue for an efficient RF design to sustain the capacity scaling in the future GPGPUs. We first identify the causes for the bank conflicts. Then, in the presence of the many bubbles observed in bank utilization and RF entries, we propose to apply fine-grained architectural control by stealing the idle banks to serve concurrent operand accesses, which effectively recovers the throughput loss due to bank conflicts. By leveraging the multiwarp and multibank characteristics in GPGPUs, we
A. High-Throughput GPGPU Architecture

A modern GPGPU consists of a scalable array of multi-threaded SMs, which execute thousands of concurrent threads to enable the massive TLP for the extraordinary computational throughput. Inside each SM, threads are often issued in a warp. At any given clock cycle, a ready warp is selected and issued to the data processing lanes by the scheduler. For example, in the classic Nvidia Fermi GPGPU architecture, each SM typically has 32 data processing lanes, two schedulers, and a large number of execution units.

To keep the execution units busy even if some warps are stalled for long memory operations, a large number of warps can be switched with zero-cycle penalty, which requires that every thread in SM is allocated with some dedicated physical registers in the RF. As a result, the RF size is generally very large. More importantly, the RF capacity keeps doubling for every product generation in pursuit of even larger parallelism.

B. Register File

The requirement for simultaneous multiple read/write operations poses challenges to the RF design. For example, in the Nvidia PTX standard [10], each instruction can read up to three registers and write one register. To guarantee the throughput, the simplest solution is to use a multiported SRAM, which requires two more transistors per bit for an additional port. Obviously, constructing such a heavily ported (six-read and two-write ports for dual-issue) large RF (hundreds of kilobytes) is typically not feasible or economical.

To solve this problem, banked RF structure [3]–[5] has been proposed to avoid the unaffordable area and power overhead in the multiported designs. While there are various possible RF organizations, one of the most commonly used is to combine multiple single-ported banks into a monolithic RF, which reduces the power and design complexity. Each bank has its own decoder, sense amplifiers, and so on to operate independently. Registers associated with each warp are mapped to the banks according to the register/warp identifiers and the maximum registers a thread can use [3], such that different registers in a warp and the same registers in different warps can be evenly distributed across the multiple physical banks. In this way, the mapping offers a fair balancing and higher possibility for the multiple operands in an instruction to be fetched across multiple banks in a single cycle with no conflicts. Otherwise, they have to be serialized degrading the RF throughput.

A typical banked RF structure is shown in Fig. 1. For a scheduled instruction to be dispatched into execution, it first reserves a free operand collector. The collectors can receive multiple operands via crossbar concurrently. If running out of collectors, the instruction waits until a collector becomes available. The arbiter receives RF requests from the collectors, and then tries to distribute them to different banks but only allows a single access to the same bank at a time. Once bank conflicts occur, the requests have to be serialized. For example, write operations usually take higher priority over reads on
accessing the same bank [7], [11], [12], anticipating an early clearance of the dependent instructions on the scoreboard. After all the operands for an instruction are collected, it is dispatched to a dedicated execution unit, e.g., integer, special function, or ld/st units, and the hosting collector is released. Note that the bank conflicts directly affect the operand fetching latency, and then saturate the limited number of collectors dedicated to the execution units, eventually causing issue stalls at the beginning of the execution pipeline. With fewer conflicts, collectors can turn around faster and better serve the execution units.

III. MOTIVATION

In pursuit of larger TLP, the RF capacity keeps doubling for every product generation. Conventional designs simply apply coarse-grained structural expansion by adding more banks to accommodate more registers. However, this solution is not scalable due to the significant area and power overheads with marginal performance gains.

A. Design Space Exploration

Conflicts have been acknowledged as the major reason for the reduced RF throughput. Adding banks can improve the throughput, because the possibility of conflicts is reduced by distributing registers into more independent banks. In this section, we conduct a design space exploration to experimentally study how banking impacts the performance, area, delay, and power of a pilot RF design described in Section V.

1) Performance: We vary the number of banks with the total RF capacity fixed to 128 KB per SM that is typical in Fermi-like GPGPUs [1]. The performance results are averaged across all the benchmarks and shown in Fig. 2 by normalizing to the 4-bank RF. Fig. 2 shows that adding more banks benefits the performance significantly when a small number of banks are used. With increasing banks, the benefit diminishes as the 16-bank RF delivers almost the same performance as the 32-bank case due to the much reduced conflicts. The exploration reveals that 16 banks can be optimal for the 128-KB RF.

We also conduct the exploration with 2× TLP on a 256-KB RF and 4× TLP on a 512-KB RF per SM to study the features in newer GPGPU architectures, such as Kepler. Detailed results are given in Section VI-C. Briefly, it is observed that the optimal design points shift toward more banks with larger RF for higher throughput. When more warp instructions are competing for RF accesses, the number of banks needs to be increased to reduce the chance of conflicts. Otherwise, the performance gains from larger TLP can be compromised.

2) RF Area: We use 128-KB RF as a case study, Fig. 3(a) shows the breakdown of the banking area, such as the array cells, peripheral circuitry, and crossbar. It clearly shows that more banks cause significant area overhead. The area of array cells stays constant across different schemes due to the fixed RF capacity. In contrast, the area of the peripheral circuitry, such as decoders, prechargers, equalizers, bitline multiplexers, sense amplifiers, and output drivers, expands notably with increased banks. This is because the GPGPU registers are of 128 bytes,1 much wider than that in CPU. This significantly increases the number of bitlines and widens the data buses. Each individual bank duplicates the peripheral circuitry for independent array control, resulting in even larger area expansion compared with the conventional CPUs [13].

Meanwhile, the dimension of the crossbar between RF banks and collectors grows significantly with the number of banks. As a result, nearly 50% additional area is observed from a 16-bank RF to a 32-bank RF, although the two designs deliver quite similar performance, as shown in Fig. 2.

3) RF Timing: Because we fix the RF capacity, a larger bank count naturally results in shorter access time inside the memory array, as shown in Fig. 3(b). However, array access time is just a portion of the total RF delay and the advantage is compromised by the wire transfer time via a larger crossbar. With an increasing number of banks, the crossbar delay tends to dominate. However, we find that even the worst case scheme studied in this paper can still meet the sub-1-GHz GPGPU frequency, so we do not consider their timing differences.

4) RF Power: Fig. 3(c) shows the dynamic power of the banks. The cell array power slightly decreases with more banks, because the bitlines are shorter under fixed RF capacity. In contrast, the dynamic power spent on data transfers via the crossbar increases considerably due to the larger RF area. Meanwhile, the leakage power increases significantly, as shown in Fig. 3(d), because the banks duplicate the peripheral circuitry.

5) Summary: The design space exploration reveals the fact that fewer banks benefit the RF physical design, such as area, timing, and power. However, simply reducing the number of banks is not acceptable from the performance point of view. For example, there is around 5% performance loss from a 16-bank design to an eight-bank design, as shown in Fig. 2, while this paper shows that there is still room for performance improvement even above the conventional 16-bank RF, if the conflicts can be effectively eliminated as we will demonstrate by our proposed 8-bank RF design.

B. Causes for Bank Conflicts

Without conflicts, the operands should pass through the RF with one cycle latency offering maximum throughput. In reality, the operands have to go through many steps until they can be dispatched to the execution units. An operand starts its journey by first going through the arbiter for granted

---

1There are 32 threads per warp, each with a register of 32 bits. These threads access the RF simultaneously for the 32 registers, resulting in a combination of 128 bytes for each warped register access.
access to a bank, then passing through the crossbar to temporarily reside in the collector and finally being dispatched. In the case of a conflict, only one operand can proceed, causing latency penalty to other competing operands. We identify two major types of conflicts that lead to the latency penalty.

1) **Interread Conflict:** This occurs between different warp instructions. For example, reading register $r_a$ may conflict with a read to the same bank from another warp instruction.

2) **Interwrite Conflict:** For example, reading $r_a$ may conflict with a write to the same bank from another warp instruction. When write is prioritized, $r_a$ has to be deferred.

We conduct architectural simulation to quantify the conflict impact in terms of the penalty on average RF access latency. Higher penalty means longer operand waiting that results in reduced RF throughput and more issue stalls. From Fig. 4, we see that the interread and interwrite conflicts are quite common, resulting in an extra 0.1 to 0.6 cycle latency that directly causes performance loss. Another type of conflicts (i.e., the intrainstruction conflicts) comes from the competing operands in the same instruction, which rarely happens and can be effectively eliminated by compiler renaming.

### IV. PROPOSED TECHNIQUES

#### A. Overview

In this paper, to reduce the conflicting accesses for faster operand fetching, we propose the bank stealing techniques to opportunistically steal unused banks for better operand coordination. For interread conflict, it steals a free bank at the current cycle for an operand that is supposed to be fetched in the next cycle avoiding the potential upcoming conflict. For interwrite conflict, it steals an idle bank with a free register entry to temporarily store the write-back value to avoid the conflict with a read. The bank stealing technique effectively leverages the underutilized banks to maximally serve the pending RF accesses.

In fact, the stealing techniques adequately exploit the multiwarp, multibank feature in GPGPUs. Unlike CPU, GPGPU pipeline is likely to have multiple ready warps waiting at the issue stage (14 out of 48 warps as observed in our simulation). This makes it possible for a wise operand coordination especially with the underutilized banking resources.
In addition, as the banked RF structures are very common and essential for SIMT, the proposed schemes can be broadly applicable to a wide range of GPGPU architectures.

B. Bank Stealing for Interread Conflicts

1) Basic Idea: In the conventional RF design, whenever a conflict occurs, one of the register reads has to be deferred causing pipeline stall. However, if one of the conflicting reads can be moved to the previous cycle and steal the bank when vacant, the conflict and pipeline stall can be eliminated.

As shown in Fig. 6, at time $T_1$, warps $W_a$ and $W_b$ are issued at the pipeline issue stage. Here, we assume that both instructions have only one operand to read for simplicity. Their operands $r_a$ and $r_b$ are granted to read the RF by passing the conflict checking at the arbitration stage at $T+1$. In addition, at $T+1$, warps $W_c$ and $W_d$ are issued, and their operands $r_c$ and $r_d$ are found to conflict in the arbitration stage at $T+2$. Conventionally, either $r_c$ or $r_d$ has to be delayed resulting in serialized reads at $T+3$ and $T+4$. Differently, in our bank stealing scheme, the read of $r_d$ can be moved one cycle earlier if it is not conflicting with $r_a$ and $r_b$ by passing conflict checking at $T+1$. In that case, $W_c$ and $W_d$ can finish operand reading at $T+2$ and $T+3$ without pipeline stall, saving one cycle penalty. However, if $r_d$ conflicts with $r_a$ or $r_b$ at $T+1$, the stealing is aborted and the pipeline behaves as before. In fact, the bank stealing increases the opportunity for read success by stealing the unused RF bandwidth in the previous cycle.

2) Warp Scheduling for Read Stealing: To enable the read stealing, we need to identify the warps ($W_c$ and $W_d$) to be issued in the next cycle ($T+1$) at the arbitration stage of the current warps ($W_a$ and $W_b$), and check for the bank conflicts of operands ($r_a$, $r_b$, and $r_d$) between the current and next warp instructions. The question is how to identify the candidate warps to be issued one cycle ahead for bank stealing with the current warps, so that we can put their operands into the bank conflict checking in the arbitration stage of the current warp instructions.

In fact, this is feasible due to the warp organization featured in GPGPUs. It is very common that multiple warps are ready and waiting for issue in the instruction pool. The selection of candidate warp can be done with the aid of the default warp scheduler. As shown in Fig. 7, a scheduler typically uses a hierarchical selection tree after the ready checking to select and wake up a warp for issue, as indicated in [14]. The ready checking considers only the warps that are eligible for execution without hazards, and the hierarchical tree selects the warp with the highest priority at the bottom level. Note that at one stage above the bottom level of the tree, there are two warps. In our scheme, we always treat the other warp that is not selected for issue at the current cycle as the candidate warp for stealing. The operands of the current and candidate warps will be latched into the arbitration stage to check for conflicts. Upon conflicts, the operands can be read ahead if two conditions are met: 1) there must be a free collector to hold the stolen operands and 2) the stealing should be able to resolve the conflicts in the next cycle and not cause conflict in the current cycle. Note that the scheduler should be overridden in the next cycle to prioritize the candidate warp for issue if their operands have been read ahead.

3) Design Cost and Scheduler Impact: As discussed earlier, the candidate warp is selected by utilizing the default scheduling and then latched to the next arbitration stage. To enable the read stealing, the default scheduler should be overridden in the next cycle when a stealing has been performed. As shown in Fig. 7, this logic can be simple that only involves three multiplexors. For multiplexor $m_1$, it selects the current warp from the selection tree if the read stealing has failed. Otherwise, it overrides the scheduler by selecting the candidate warp latched in the last cycle. On the other hand, to update the candidate warp accordingly, $m_2$ still selects the other warp from one level above the bottom as the candidate warp if the read stealing has failed. Otherwise, it selects the current warp as the next candidate warp for stealing, because the stolen warp in the last cycle is being issued now. $m_3$ gives the selecting signal depending on final scheduling decision $left\_sel$.

Regarding the timing impact, both $m_1$ and $m_2$ depend on the $left\_sel$ signal from the selection tree, and the read stealing succeeds signal feedback from the arbitration stage. To minimize the impact, the feedback signal from arbitration stage should be available as early as possible. Fortunately, it is out of the critical path in the arbitration stage and can be asserted immediately once we find that there are no requests of higher priority. Therefore, $m_3$ does not add extra delay, because it aligns with the multiplexor in the last level of the selection tree. $m_2$ aligns with $m_1$ to give the current warp and candidate warp. Then, the two paths can have the same gate-level depth by adding a 2-to-1 multiplexor to the baseline, whose delay is estimated as $1.2 \times$ of an FO4 inverter delay.

Fig. 6. Operand read with bank stealing for interread conflict removal.

Fig. 7. Warp selection for read stealing aided by the default scheduler.
With stealing enabled, the default instruction flow might be changed from the original scheduler, but the impact turns out to be small on the performance as observed in our experiments. That is because the candidate warps are chosen from the second level to the bottom level of the selection tree, which are likely to be scheduled in the next cycle by the default scheduler anyway, albeit not always of the next highest priority. It is also possible to select other ready warps as the candidate warp for stealing, but the performance can subject to more changes. We select the candidate warp as proposed in Fig. 7, which can be compatible to the conventional scheduler and a similar performance can be expected. For example, when the read stealing is built upon the greedy-then-oldest (GTO) scheduler, the original scheduling sequence of \( Wa, Wb, Wb, \ldots \), may be changed to \( Wb, Wa, Wb, \ldots \), depending on the actual conflicts happened. In this way, the scheduler with stealing is not for conflicts alone. It can maximally inherit prior wisdom on scheduler designs, such as GTO, round-robin, and other schedulers [14], [15], that are essentially built upon them, but adding the bank conflicts into account.

The essence of bank stealing lies in the fact that the RF banks experience underutilization, as shown in Fig. 5. By operand coordination across multiwarps, it leverages the idle banks at current cycle and opportunistically fills them with predicted future reads. Note that read stealing does not reduce the total number of reads. It just makes a better and more balanced utilization of the available RF bandwidth.

### C. Bank Stealing for Interwrite Conflicts

1) Basic Idea: In this section, we propose another type of bank stealing to eliminate the interwrite conflicts. As a default scheme for the write conflicts, the read accesses on the same bank have to yield due to the higher priority of register writes [7], [11], [12], anticipating an earlier release of the dependent instructions from the scoreboard. Note that the compiler is not able to avoid this conflict beforehand, because the write occasion is unknown for load operations.

Due to the multiwarp nature in GPGPUs, oftentimes, the written value is not immediately used, because the dependent instructions are held in the issue pool. In that case, deferring the write will allow the read to proceed, successfully reducing the pipeline stalls and overlapping the latency among warps. The write stealing can be shown in Fig. 8. A register read of \( r_0 \) arrives at the same cycle as a write to \( r_8 \) in the arbitration stage at time \( T \) causing a bank conflict.

Conventionally, the arbiter will grant the write access to \( r_8 \) that blocks the read of \( r_0 \). Instead, in our scheme, \( r_8 \) can be deferred. We also clear the dependent instructions of the scoreboard and resort to a dependence checking to enforce the dependence. Actually, since there are many pending warps, the dependent instructions are likely to be trapped in the issue pool, which offers even more opportunities for the write back of \( r_8 \). Unlike CPUs, this turns out to be a common case in our simulation, as we find that there is 60% of chance that the first dependent read arrives more than three cycles later. This implies the potential of write stealing, and our experiment shows that the majority of stolen writes can find a write-back slot within three cycles.

Note that there can be other alternatives applicable for interwrite conflict removal. For instance, a multiport or pseudo dual-port RF helps at the cost of significant area overhead but with infrequent usage. Another solution is a fully associative write buffer as widely used in CPUs [16], or dedicated buffers distributed on each bank. However, it turns out to be costly because in GPGPUs, each warp register contains at least 32 threads and each with 32 bits. This means that we need to add 1 kb to buffer just one register entry. Wiring the results to the buffers is also costly. For example, the crossbar has to be widened to connect each individual bank, and multiple write ports are required for concurrent write backs. These modifications further increase the area and power, given the super-wide datapath in GPGPUs. Since the RF itself is experiencing underutilization most of the time, it is meaningful to take it for the buffering to maximize the resource reuses, which is one of the primary goals of this paper.

2) Temporary Storage for Write Stealing: The prerequisite of the write stealing is to find a place to temporarily hold the write-back value. We propose to use the off-the-shelf inactive register entries in the RF with the lightweight compiler support. This is reasonable in GPGPU architecture given the two observations that: 1) the entries and banks are oftentimes underutilized for GPGPU applications and 2) the banks are already connected to the result bus that offers direct datapath for the operand to be written into the stolen bank.

To find the inactive register entries, we associate a small status table holding the free register ID for each bank, i.e., \( R_{\text{free}} \) with a valid bit \( f \), as shown in Fig. 9. It is analyzed with compile-time liveness information that can be embedded.
into instruction encoding, and then assigned to hardware by passing along the pipeline as similarly done in [17] and [18]. For instance, \( R_{\text{free}} \) is initialized as an inactive register, e.g., \( r_a \), into the first instruction encoding. When \( r_a \) is used by the following instruction, its encoding will embed another register, e.g., \( r_b \) if inactive, which also carries an update of \( R_{\text{free}} \) to \( r_b \) at runtime. Note that any warp running on the SM is able to update \( R_{\text{free}} \). Therefore, it is expected to assign \( R_{\text{free}} \) as the least used register if the program is limited by registers or a register that is never used if the program is not limited by registers. We have found that a single entry for each bank is sufficient for the write stealing, and therefore, a single \( R_{\text{free}} \) is enough to identify the current inactive register.

Once an inactive register \( R_{\text{free}} \) is found upon an interwrite conflict, a link between the temporary \( R_{\text{free}} \) and the actual destination register \( R_{\text{dest}} \) is established and recorded into the table with a valid bit \( m \) and the destination register/bank ID, as shown in Fig. 9. The information will be used in the arbiter to determine when it can be written back to the destination register and removed from the temporary register. Typically, the arbiter will issue a write back when: 1) the result bus is free and 2) the bank holding the temporary register is free. The write-back value is first read out from the temporary register and put onto the result bus. In the next cycle, the value will be written back to the destination register. Nevertheless, it is still possible that the write back to the destination register can conflict with a read to that bank. If that happens, we will not steal again but complete the write by stuffing the read.

Write stealing offers a second chance to write back the value, hoping to avoid the bank conflict in the first round. In case the temporary register does not manage to write back before the incoming dependent accesses from other instructions (e.g., read-after-write (RAW) or write-after-write (WAW)), a forced write-back is required resulting in a pipeline stall to ensure the data integrity. This is achieved when the dependent instruction arrives at the arbitration stage, the arbiter can prioritize the write-back over the dependent instruction to avoid any hazard. However, because registers are not always immediately used in the multiwarp GPGPUs, most of the time, the temporary register can manage the write-back without pipeline stalls. Finally, because the free registers are dead registers, very rarely they become alive in the program execution, and the temporarily stored value needs an immediate write-back, which otherwise might cause pipeline stall.

Compared with the traditional scheme, write stealing adds extra read and write operations, which is negative on energy. However, the added energy can be small due to the limited write stealing operations as observed in our experiments. Given the write energy only accounts for a portion of the dynamic RF energy (around 30%–40%), and the leakage tends to dominate with shrinking technology nodes (up to 70%) as exposed in [19], the overall RF energy can be reduced.

The required datapath changes are minimal. The result bus already exists and serves as the regular channel for writing back the results to the RF banks. The status table contains 19 bits per bank (two valid bits, 7-bit free register ID, and 10-bit destination register/bank IDs) and altogether 152 bits for eight banks. Compared with the traditional buffering schemes incurring at least thousands of bits overhead, the hardware cost can be negligible. In addition, given the highly paralleled structure in GPGPUs, our proposal applies a distributed buffering in each bank by using inactive register entries. This can eliminate the wiring and multiport problems that embarrass the centralized write buffer solution.

### D. Arbitration for Bank Stealing

Finally, the arbiter is modified to enable the bank stealing. For each bank in any clock cycle, there can be three more types of incoming accesses except for the normal writes and reads, i.e., stealing reads, stealing writes, and forced writes. A stealing read means that the read tries to steal a bank for interread conflict removal. A stealing write means that the write tries to use a free bank with a free register to temporarily hold the write-back value. A forced write means that the write-back value will be forced to write back to the destination register.

1) **Arbitration Changes:** Fig. 10(a) shows the logic to arbitrate the read/write request to a bank. At first, all operand requests are decoded into bank requests as in ①. Then, the requests are passed through a set of cascaded selection in ②, where the bank requesting signal will be generated for each type of accesses, such as normal read or stealing write, as shown in Fig. 10(b). Finally, the priority logics in ③ will grant the ultimate access according to the priority order. Note that all these steps are required in the conventional design.

To support bank stealing, we add logic to check the data dependence in ②. In general, the forced write takes the highest priority, because it either comes from the previously stealing write that cannot be deferred for a second time or this is the first write back but it cannot borrow a vacant place for a stealing write. In either case, all other conflicting accesses have to yield. If there are more forced writes to the same bank, they will be executed in the consecutive cycles.

When a normal write conflicts with other types of accesses, it becomes a stealing write if a free register in a free bank can serve as temporary storage. This stealing checking is performed in ③, where the normal writes are arbitrated with bank vacant status, as shown in Fig. 10(c). We use a round-robin policy to arbitrate the banks qualified for stealing.

In summary, the priorities for all these accesses in ③ are ordered as: forced write > normal read > stealing read > normal write > stealing write. For each cycle, only one access with the highest priority is granted to the bank while all other accesses have to stall.

2) **Design Cost and Analysis:** As shown in Fig. 10, the arbitration changes mainly lie on the cascaded selection, the write stealing checking, and the prioritization. The added logic is symmetric, modular, and scalable, and therefore, we believe that the design complexity is manageable. We design the proposed circuitry and find it to be less than 1.2% of the RF area. Because the logic mainly involves 1-bit signals, the area overhead can be negligible when compared with the extra buffers of 1024 bit wide, which requires similar arbitration logic changes as well.
In terms of timing, the shaded line in Fig. 10 shows that the normal read is the original critical path in the arbiter. This is because there can be much more normal reads to be arbitrated, and the cascaded selection on them is generally long enough to overlap the delay of all other types of accesses that run in parallel. For example, the decoding of stolen reads in ① can be done in parallel with the decoding of the normal reads. The checking and selection of stolen writes in ② and ③ can be done in parallel with the selection of the normal reads. The only logic added is in the prioritization logic ④ on two paths. For the read enable signal path \( ren \), a two-input OR gate converges the normal read and the introduced stealing read if no forced write blocks them. For the write enable signal path \( wen \), a four-input AND gate converges the normal write and the stealing write if no read request blocks them. Because the normal read is assumed as the critical signal, the AND gate only functions until the normal read settles. Note that the normal read also forms the original critical path, which requires a two-input AND gate as well to assert the read request with no blocking normal write. Therefore, the two paths align approximately in timing by adding one gate-level depth, e.g., a two-input AND and a two-input OR gate, whose delay can be estimated as 0.73× of an FO4 inverter delay. We simulate and prove that tighter delay constraints can absorb its delay to the sub-1-GHz nominal frequency.

V. EVALUATION METHODOLOGY

As discussed earlier, the design overhead has been analyzed from a hardware perspective. In our experiment, we turn to an architecture perspective by evaluating the RF designs with different numbers of banks, regarding their performance, area, and energy efficiency using our proposed bank stealing.

A. Design of a Pilot RF

The behavior of a multibanked RF has been introduced in Section II-B, and we use the same register mapping and organization to the baseline RF design. To learn the physical parameters of such a multibanked RF, we synthesize a pilot RF with multiple single-ported banks by using SMIC commercial memory compiler tool [20] with industrial cell library. It gives standard SRAM designs with four metal layers for over-the-cell routing. We build the RF by varying the number of banks with the area, power, and delay number reported from the memory compiler.

We also evaluate the crossbar, which connects the banks and the operand collectors with 128-byte bus for each operand. We use the bit-sliced structure [21], [22] due to its higher density for such a wide crossbar. The 128-byte slices are organized into a 32×32 square array. This structure is regular and easy to scale to various numbers of inputs and outputs, which well fits the needs in our design space exploration. After implementing the layout, it turns out that the area is severely dominated by wiring even using the minimum pitching size. This is due to the large operand bitwidth, which requires the wires to be routed into the innermost slice at each direction. Therefore, we use two metal layers to split the horizontal and vertical wires to the four edges of the square to further reduce the wiring size. Finally, the area, power, and delay of the crossbar structure are calculated based on the HSPICE simulation of a single-bit slice due to its regularity.

B. Experiment Setting

For the performance measurement, we use the cycle level architectural simulator GPGPU-Sim v3.2 [23]. If not otherwise specified, each SM has 32 execution lanes and a 128-KB RF serving at most two warp instructions at a time. Other key configuration parameters for the simulator are shown in Table I, which generally conform to the Fermi architecture that is faithfully modeled by the simulator. We also apply our techniques to 256-KB RF and 512-KB RF designs featured in newer GPGPU architectures for a scalability study.
TABLE I
KEY PARAMETER SETTINGS IN THE GPGPU-Sim Simulator

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Cores (SMs)</td>
<td>15</td>
</tr>
<tr>
<td>Core Frequency (MHz)</td>
<td>700</td>
</tr>
<tr>
<td>Warp Size</td>
<td>32</td>
</tr>
<tr>
<td>Number of Threads / Core</td>
<td>1,536</td>
</tr>
<tr>
<td>Number of active CTAs / Core</td>
<td>8</td>
</tr>
<tr>
<td>Number of Registers / Core</td>
<td>32,768</td>
</tr>
<tr>
<td>Number of Collectors / Core</td>
<td>10</td>
</tr>
<tr>
<td>Number of Warp Schedulers / Core</td>
<td>2</td>
</tr>
<tr>
<td>Warp Scheduling Policy</td>
<td>GTO</td>
</tr>
</tbody>
</table>

Table II shows the benchmarks in our evaluation. We investigate 14 benchmarks from CUDA SDK [24] and Parboil [25] suite, which are widely used in GPGPU study. We use CUDA SDK 4.2 for compilation and the simulator is configured to use PTXPlus syntax for simulation. This is a hardware instruction set extended by GPGPU-Sim with register allocation information to enable more accurate simulation results, as reported in [26]. Due to the lack of open-source compiler alternative to commercial nvcc to generate codes with register liveness information, we intercept the compiled kernel before running the simulation. This interception binds our liveness analysis to each register. The simulator is also modified accordingly in pipeline to accept the liveness information to identify the RF holes at runtime.

In our delay and power evaluation, we use the primitives from the pilot RF design under different banking schemes. Detailed primitives are shown in Table III under 32-nm technology and they are obtained from the SIMC memory compiler tool and our custom layout. We collect RF statistics from GPGPU-Sim and use these primitives to compute the RF power, including the normal bank accessing, crossbar transmission, arbiter, and bank stealing if enabled. We also consider the extra read and write operations on each stolen write. For all other non-RF components, we use GPUWattch [27] to calculate the power.

VI. EXPERIMENTAL RESULTS
We first show the overall performance improvement by our proposed techniques with different banking numbers. In all the results, straight RF stands for the traditional RF without the proposed techniques and the proposed RF stands for the RF equipped with the proposed techniques. As a case study for a better understanding of our proposed techniques, we choose representative design points for detailed analysis. We also sweep different RF sizes to demonstrate the scalability.

A. Design Space Study
For a thorough study, we apply our techniques to RF designs with a fixed capacity of 128 KB but different numbers of banks (4, 8, 16, and 32). The results given in Fig. 11 are averaged on all the benchmarks, including performance, area, area efficiency, and energy efficiency. The area efficiency is calculated as normalized performance over area, while the energy efficiency is calculated as normalized performance over energy.

From Fig. 11(a) and (b), we see that RF designs with our bank stealing techniques always achieve better performance over the straight RFs with the same configuration. The RFs with the same configurations have almost the same area due to the negligible hardware cost for bank stealing. The overall performance improvement ranges from 8% to 13%. Given the fact that we only make change on a single GPGPU unit, this improvement is significant.

The most important observation is that the proposed RF can often outperform the straight RFs equipped with more banks. For instance, the proposed 8-bank RF outperforms the straight RFs with 16 banks or 32 banks. It exhibits the effectiveness of our proposed techniques that adequately eliminate the potential bank conflicts. It is more efficient than purely adding more banks. As a result, our proposed techniques are much more area-efficient, achieving nearly 25% and 50% area savings over the 16- and 32-bank straight RFs. At the same time, the performance is also superior with 7.2% and 5.7% improvement over the 16- and 32-bank straight RFs.

We also compare the area efficiency and energy efficiency between proposed RFs and straight RFs in Fig. 11(c) and (d), respectively. All the results demonstrate that our proposed RF is superior in terms of performance, area, and energy efficiency. Because the RF size keeps increasing in modern GPGPU, the much improved area efficiency with better RF performance will be beneficial for the future scaling.

B. Case Study
As a case study, we choose the RF design that exhibits the highest area efficiency from the proposed RFs (eight bank).
Accordingly, we choose eight-bank and 16-bank straight RF designs for comparison. The 16-bank straight RF acts as the baseline and all the results are normalized to it.

Fig. 12 shows the performance comparison in detail. From the bottom bar in Fig. 12, we see that the straight 8-bank RF degrades the performance by nearly 5% compared with the 16-bank straight RF design. That is mainly due to the increased bank conflicts.

1) Evaluation on Read Stealing: In Fig. 12, we see that the average performance gets improved by around 7% over the straight 8-bank RF by read stealing. That means an averaged better performance of 2% than the straight 16-bank RF. Read stealing alone can provide better performance than the baseline straight 16-bank RF design except for M.Car out of the 14 benchmarks. The experiment results confirm that the read stealing can effectively reduce the potential conflicts and improve the overall GPGPU performance.

It is worth mentioning that we have done the same experiments using different types of warp schedulers, including round-robin, two-level, and so on. The conclusion stays the same to our results using GTO. Although the read stealing may occasionally modify the instruction flow, the experiments verify that this scheme is generally applicable and not sensitive to the host scheduling policies.

2) Evaluation on Write Stealing: We further apply write stealing on eight-bank straight RF. The results are stacked on top of read stealing, manifesting the total performance gain by the combined schemes shown in Fig. 12.

We find that the two schemes are complementary to each other on performance. They remove the conflicts and reduce the latency, which is essential to improve the RF throughput. When the two bank stealing techniques are applied together, the enhanced 8-bank RF can outperform the straight 16-bank RF by around 7%. Specifically, we see that it outperforms the straight 16-bank design for all the benchmarks.

3) RF Access Latency: We investigate the RF access latency by applying our proposed schemes on straight RF design. We take the latency as the indicator for the impact of conflicts, as it is a more direct measurement of RF throughput.

Fig. 13 shows the latency penalty measured in pipeline cycles when accessing the RF. Compared with Fig. 4(a) on the 16-bank straight RF, we find that the straight 8-bank RF nearly doubles the latency penalty due to aggravated conflicts. Instead, our proposed 8-bank RF effectively shortens the operand latency by reducing the interread and interwrite penalties by around 58% and 88%, respectively. The write stealing is more effective to remove the interwrite conflicts, while the read stealing largely depends on the prediction of the potential interread conflicts in the next cycle.

From this plot, we see that the fetching latency can be treated as a performance indicator, albeit not linearly. For applications with significantly reduced operand latency, the proposed RF usually achieves better performance than the straight 16-bank RF (Q.Gen, S.qrng, and Tpacf). In a word, our proposal is effective on reducing the operand latency penalty with fine-grained control of the precious RF resources.

We also notice an increasing utilization to 54% by the proposed 8-bank RF with bank stealing against 23% in the straight...
16-bank RF. This is mainly because the greatly reduced number of banks and more balanced usage of the RF banks that accelerate the operand fetching for better performance.

4) More Evaluation on Write Stealing: In general, write stealing can eliminate the interwrite conflicts. This is under the assumption that the temporary register can find a slot to write back to the destination register before the first dependent access. However, this is not always true. Whenever there is pending RAW or WAW hazard, the value resided in the temporary register has to be written back incurring pipeline stall. In Fig. 14, we find that the majority of registers manage to write back without any intervention to the execution pipeline. First, 83% of the registers are normally written without stealing. Therefore, the power overhead introduced by write stealing can be limited. For the registers that do require stealing, over 25% of the registers are written back within one cycle after stealing. Another 50% of the registers are written back successfully by waiting several more cycles but still before its first dependent access. Compared with the baseline, where 17% of the registers are written back conflicting with reads, only 2.5% writes are left that negatively stall the pipeline.

5) Evaluation on Power and Energy Efficiency: We show the power number and use the metric perf/power to quantify the power efficiency and perf/energy for the energy efficiency. The power results reported in this section are chip-wide power counting in all the GPGPU core units.

Fig. 15 shows the power results. From Fig. 15, we confirm that RF designs using fewer banks consume less power. For instance, the straight 8-bank RF consumes 6% less power than the straight 16-bank RF. This is because fewer banks slightly increase the dynamic banking power but reduce the crossbar power and the leakage, as explored in Fig. 3(b). However, using traditional designs, fewer banks are inferior in terms of the energy efficiency when performance is accounted.

C. Scalability Study

Finally, we demonstrate the scalability of our techniques on RF designs in newer GPGPU architectures. For example, the Kepler doubles the RF capacity, allows 255 registers per thread with ISA support, and simplifies the register scoreboard hardware with compiler information, as documented in [2]. Regarding the VLSI design, we believe that there are not too many changes as the basic building block, i.e., the SRAM array is hard to change. Therefore, our scalability study focuses on the bank organization and capacity increase by applying our techniques onto RFs with different capacities (128, 256, and 512 KB per SM). We also increase the number of thread blocks and schedulers accordingly to represent the higher pressure from increasing TLP. For each RF size, we select a pair of optimal RF designs with and without our schemes by considering a balanced performance and area.

Then, we compare their area and energy efficiency in Fig. 16. We find that the most area and energy-efficient configuration is generally not the design with the largest number of banks due to the marginal performance gains at the cost of excessive power and area overhead. The optimal point varies with the RF capacities meaning that the traditional RF design is generally not the design with the largest number of banks but can deliver comparable performance to the traditional RFs with a larger number of banks. This promises more silicon area savings with technology scaling. Our proposed bank stealing can effectively reduce the conflicts with varying RF sizes. We find a steady improvement from 40% to 80% in the area efficiency and significantly better energy efficiency by 15%–28% over the traditional design. The results demonstrate the superiority for our schemes and well justify them for the better scalability in the future GPGPUs.
VII. RELATED WORK

There has been an ample amount of work studying efficient RF structures in different microprocessors, including conventional CPUs, very long instruction word, and streaming processors such as GPGPUs.

For example, in general microprocessors, SRAM–dynamic random access memory (DRAM) hybrid macrocell [28] and 3T1D embedded DRAM [29] have been applied on L1D caches in general microprocessors. In the GPGPU domain, a variety of new RF implementations, such as hybrid SRAM–DRAM RF [7], spin transfer torque random access memory (STT)-magnetic random access memory (MRAM) RF [30], and embedded dynamic random access memory (eDRAM)-based RF [17], [31], [32], have been proposed. As these new types of memory generally compromise performance, these works focus on smart strategies to maximally recover the performance loss.

Techniques exploring architectural modifications are intensively studied to improve performance or power. One study related to this paper in the CPU domain [16] proposes an area-efficient CPU RF by reducing ports and applies prefetching and delayed write back to sustain performance. They leverage queues with dozens of entries and heavily augment CPU pipeline. However, in GPGPU, it is impractical to apply their methods, because there is no register renaming stage and the operand width is too wide for queuing. Prior work also studies multicore cache [33], [34] or cache-conflict problem [35], [36] by a prediction mechanism. Other researchers are interested in hierarchical RF structures in the CPU domain, such as [37] and [38] for less complexity, and RF caching [30], [39], [40] for performance enhancement.

In the GPGPU domain, the work [4] aims to reduce the main RF access energy by introducing a small register cache for frequently accessed operands. To prevent the register cache from thrashing, they further propose a two-level thread scheduler to maintain a small set of active threads on deck. According to their configuration, they require an additional 6-KB fully associative cache per SM to keep the performance without loss. This might aggravate the already congested RF layout with degraded timing parameters. In contrast, our previous work [9] reduces the RF area and improves the performance by identifying the interread conflicts. This work steps forward by identifying the interwrite conflicts and proposing a combined structure to combat with them and enables further performance improvement. Another work in [3] proposes to reduce the dynamic and leakage power by identifying the inactive or unallocated registers, which is complementary to our study for further reduction on RF power. For better performance, cache in GPGPU is also intensively studied recently, such as [41]–[43]. Another work proposes a unified on-chip memory design [5], which is able to reconfigure the capacities of RF, shared memory, and cache to accommodate diverse applications. The unified structure merges all the on-chip storage that leads to longer interconnecting wires and longer data access latency.

VIII. CONCLUSION

In this paper, we propose efficient RF design with explicit conflict eliminating techniques to sustain the high RF throughput facing ever-increasing TLP. We first reveal how different alternatives affect the overall performance. To the bank conflict problem, we identify the causes and propose two effective techniques accordingly. The read stealing scheme explicitly exploits the available bandwidth in the highly banked RF structure to eliminate conflicts and leverages the massive parallelism in GPGPU to access the RF more intelligently. As a complement, the write stealing scheme leverages the idle banks to further hide the latency penalty on unnecessary operand waiting. Their combination improves the performance using a half or even quarter number of banks with smaller area and energy budget, which imply that our proposed RF design with bank stealing can sustain the fast-increasing capacity of RF scaling in the future GPGPUs.

The RF importance to performance has been widely acknowledged. Always keeping low overhead in mind, our proposal explicitly leverages fine-grained control of the existing resources without adding costly buffers or other hierarchical memory design. Our proposed bank stealing techniques are generally applicable to other banked RF structure. The microarchitectural investigation enables a low cost and a more efficient architecture with the acceptable complexity.

REFERENCES


Jingjie Zhang is currently pursuing the bachelor’s degree with the Department of Electrical Engineering, Fudan University, Shanghai, China. He has been an Undergraduate Research Assistant with the Advanced Computer Architecture Laboratory, Shanghai Jiao Tong University, Shanghai, since 2015. His current research interests include signal processing, computer aided design, and computer architecture.

Li Jiang received the B.S. degree in computer science and technology from Shanghai Jiao Tong University, Shanghai, China, in 2007, and the Ph.D. degree from the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, in 2013. He has been an Assistant Professor with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, since 2014. His current research interests include computer architecture, computer aided design, VLSI testing, and fault tolerance, emphasized on emerging technologies and applications. Dr. Jiang was a recipient of the Best Doctoral Thesis Award at the Asian Test Symposium in 2014.

Chao Li (M’10) received the Ph.D. degree from the University of Florida, Gainesville, FL, USA, in 2014. He is currently an Assistant Professor with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China. His current research interests include computer architecture, energy-efficient systems, and green computing. Dr. Li has received best paper awards from the IEEE HPCA and the IEEE COMPUTER ARCHITECTURE LETTERS.

Xiaoyao Liang received the Ph.D. degree from Harvard University, Cambridge, MA, USA. He is currently a Professor and the Associate Dean of the Department of Computer Science and Engineering with Shanghai Jiao Tong University, Shanghai, China. He has been a Senior Architect or an IC Designer with NVIDIA, Intel, and IBM, USA. His current research interests include computer processor and system architectures, energy efficient and resilient microprocessor design, high throughput and parallel computing, general purpose graphic processing unit, hardware/software co-design for the cloud and mobile computing, IC design, VLSI methodology, FPGA innovations, and compiler technology.