Improving Thread-level Parallelism in GPUs Through Expanding Register File to Scratchpad Memory

CHAO YU, YUEBIN BAI, QINGXIAO SUN, and HAILONG YANG, Beihang University

Modern Graphic Processing Units (GPUs) have become pervasive computing devices in datacenters due to their high performance with massive thread level parallelism (TLP). GPUs are equipped with large register files (RF) to support fast context switch between massive threads and scratchpad memory (SPM) to support inter-thread communication within the cooperative thread array (CTA). However, the TLP of GPUs is usually limited by the inefficient resource management of register file and scratchpad memory. This inefficiency also leads to register file and scratchpad memory underutilization. To overcome the above inefficiency, we propose a new resource management approach *EXPARS* for GPUs. *EXPARS* provides a larger register file logically by expanding the register file to scratchpad memory. When the available register file becomes limited, our approach leverages the underutilized scratchpad memory to support additional register allocation. Therefore, more CTAs can be dispatched to SMs, which improves the GPU utilization. Our experiments on representative benchmark suites show that the number of CTAs dispatched to each SM increases by 1.28× on average. In addition, our approach improves the GPU resource utilization significantly, with the register file utilization improved by 11.64% and the scratchpad memory utilization improved by 48.20% on average. With better TLP, our approach achieves 20.01% performance improvement on average with negligible energy overhead.

CCS Concepts: • **Computer systems organization** → **Single instruction, multiple data**; • **Software and its engineering** → *Compilers*;

Additional Key Words and Phrases: GPU, register file, scratchpad memory, resource utilization

ACM Reference format:
Chao Yu, Yuebin Bai, Qingxiao Sun, and Hailong Yang. 2018. Improving Thread-level Parallelism in GPUs Through Expanding Register File to Scratchpad Memory. *ACM Trans. Archit. Code Optim.* 15, 4, Article 48 (November 2018), 24 pages.
https://doi.org/10.1145/3280849

1 INTRODUCTION

During the past decade, modern Graphic Processing Units (GPUs) have been widely used for high performance computing due to their massive thread level parallelism (TLP). For GPUs, instruction stalls and memory accesses are hidden by fast switching among massive concurrent threads, which is enabled by large register file (RF) holding contexts of all active threads. GPUs are also equipped with scratchpad memory (SPM) to support inter-thread communication within

New Paper, Not an Extension of a Conference Paper.
This work is supported by the National Key Research and Development Program of China under Grant No. 2016YFB1000503 and the National Science Foundation of China under Grants No. 61572062 and No. 61502019.
Authors' addresses: C. Yu, Y. Bai, Q. Sun, and H. Yang, Beihang University, School of Computer Science and Engineering, Beijing, 100191, China; emails: {yuchao, byb, sunqingxiao, hailong.yang}@buaa.edu.cn.

ACM 1544-3566/2018/11-ART48
https://doi.org/10.1145/3280849

cooperative thread array (CTA). To explore the high TLP of GPUs, resources should be carefully managed for better utilization. Recent works [9, 10, 35, 38, 41] have focused on improving the TLP of GPUs by exploiting the management of register file to support more active threads. WarpMan [38] uses a warp-level resource management to reduce register file fragmentation, thus more warps can be dispatched to the streaming multiprocessor (SM) even when the available resource is not enough for a full CTA. Yoon et al. [41] propose a virtual-thread architecture that can dispatch more CTAs to make full use of the register file and scratchpad memory without considering other restrictions such as the number of CTAs and thread slots. Jeon et al. [10], Jatala et al. [9], and Vijaykumar et al. [35] adopt time sharing mechanisms for warps to use the same registers at different time with increased TLP. However, all above works have taken register file and scratchpad memory as two separate units of completely different purposes.

This work is motivated by the observation that the maximum number of CTAs is determined by the most dominant GPU resources such as register file, whereas other resources such as scratchpad memory are usually underutilized. For instance, given a SM with 32K (32K*4B = 128KB) register file and 48KB scratchpad memory, if a CTA needs 5K register file and 2KB scratchpad memory, then the maximum number of CTAs can be dispatched to a SM concurrently is 6 due to the limited register file. Therefore, the remaining 2K register file and 36KB scratchpad memory is underutilized. If we can use the scratchpad memory to expand the capacity of register file, then up to 8 CTAs can be dispatched to a SM concurrently and the utilization of register file and scratchpad memory are both improved to 100%. Although equipping each SM with a larger register file can improve both the TLP and scratchpad memory utilization of GPUs, larger register file means more energy consumption [5, 6, 17, 29] and researchers even explore to reduce the size of register file [17].

In this article, we propose a new GPU resource management approach *EXPARS*, that expands the register file to scratchpad memory to provide a larger register file logically. Instead of considering register file and scratchpad memory as two separate components, our approach is able to use scratchpad memory as register file transparently when the available register file becomes limited. By expanding register file to scratchpad memory, TLP can be effectively increased, which in turn improves the performance of computation intensive applications. However, higher TLP with more active threads may lead to severe contention on cache and global memory for memory intensive applications. To address this issue, we use a *Lazy Two-Level Warp Scheduler* (LTLWS) in our resource management approach to dynamically reduce the maximum number of active warps in each SM, which is effective to mitigate the performance degradation due to resource contention.

In general, our work makes the following contributions:

- We propose a new resource management mechanism that expands register file to scratchpad memory, which improves the utilization of both register file and scratchpad memory in addition to increased TLP.
- We design a prefetching mechanism that uses co-design of compiler annotation and hardware extension to alleviate the bandwidth mismatch between operand collector and scratchpad memory.
- We develop a lazy two-level warp scheduler to dynamically determine the optimal maximum number of active warps at runtime, which can effectively mitigate the resource contention due to increased TLP.
- We evaluate our approach with representative benchmark suites. The evaluation results demonstrate that *EXPARS* can significantly improve the utilization of register file and scratchpad memory by 11.64% and 48.20% on average, respectively. It also improves the application performance by 20.01% on average with negligible energy overhead.

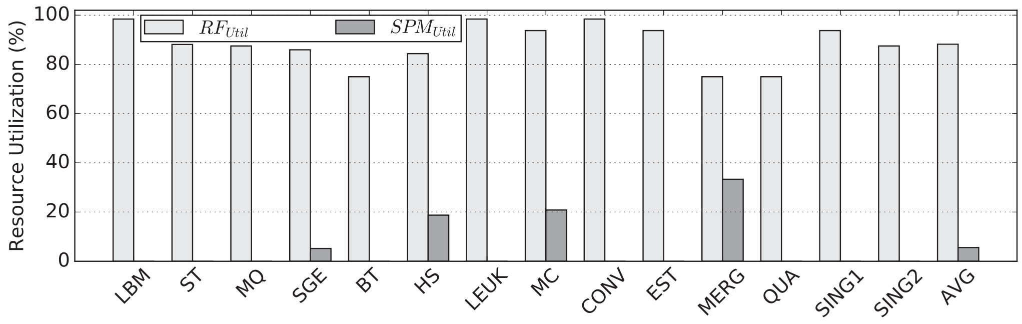

Fig. 1. Resource utilization of register file and scratchpad memory on baseline GPU.

The rest of the article is organized as follows: Section 2 introduces the motivation of this article and the challenges should be addressed. Section 3 presents the main ideas of our approach. The experimental setup and evaluations of our approach are described in Section 4. Related works are discussed in Section 5. Section 6 concludes the article.

2 MOTIVATION AND CHALLENGES

2.1 Resource Underutilization

In GPUs, every 32 threads of a kernel are organized into a warp, and warps are further grouped into CTAs. Before a CTA can be dispatched, it requires resource allocation such as register file, scratchpad memory, thread slot, and CTA slot. Modern GPUs are equipped with a large register file to store contexts of the massive threads; however, it is usually not fully utilized due to the granularity of resource allocation for CTA. Figure 1 shows the resource utilization of register file and scratchpad memory on baseline GPU (experimental setup see Section 4.1). The utilization of register file is high across all applications with 88.21% on average, which becomes the dominant factor to determine the maximum number of CTAs that can reside on a SM. However, we can see that none of these applications can fully utilize the register file. This is because the remaining register file is not sufficient to support one more CTA. Meanwhile, we also observe that the utilization of scratchpad memory is usually low with 5.58% on average. The large amount of underutilized scratchpad memory is the perfect candidate to expand the limited register file.

To expand the register file to scratchpad memory, register allocation strategy for CTA needs to be changed. For instance, if part of the registers for a CTA can be allocated in scratchpad memory, then the register file requirement of that CTA is satisfied and more CTAs can be dispatched. Figure 2 shows the comparison of overall resource utilization (combining register file and scratchpad memory) and the number of CTAs resided on each SM between baseline and oracle approach that expands register file to scratchpad memory. In Figure 2, the number of CTAs per SM increases by 29.84% on average and the overall utilization improves from 65.68% to 88.26% on average.

2.2 Challenges in Expanding Register File

Expanding the register file to scratchpad memory presents several difficult challenges that are elaborated as follows.

Challenge 1: *Coordination of register placement.* To expand the register file to scratchpad memory, the first thing to determine is how to manage the register allocation for CTAs between the register file and scratchpad memory. One approach is to allocate the registers in scratchpad memory whenever the register requirement of a CTA is beyond the available capacity of the register file, which we refer as *horizontal approach* in Figure 3(a). The benefit of the *horizontal approach*

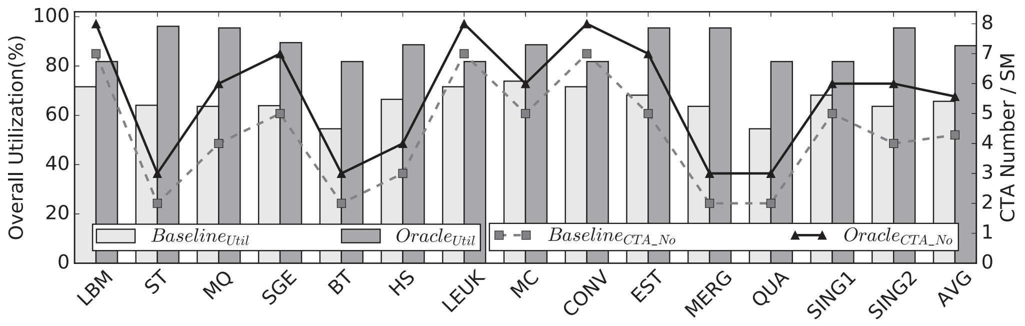

Fig. 2. Comparison of the overall resource utilization and CTA number between baseline and oracle approach.

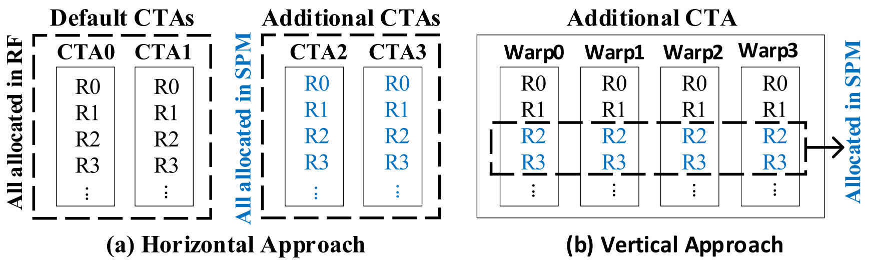

Fig. 3. Different register allocation approaches.

is that the number of CTAs originally allocated in register file (default CTAs) remains unchanged. The operands of default CTAs are accessed directly from the register file, with the performance equal to the baseline approach on GPU. In addition, the *horizontal approach* is easy to implement and the performance can be guaranteed no worse than the baseline approach if schedule the default CTAs with higher priority. However, this approach can cause pipeline stalls in the operand reading stage, especially when all source operands of instructions are allocated in scratchpad memory, which degrades the pipeline utilization severely.

To reduce the probability that all source operands need to be read from scratchpad memory, our approach distributes register allocations between the register file and scratchpad memory at per-register basis instead of at CTA or warp basis used in *horizontal approach*. We refer our approach for register allocation as *vertical approach* shown in Figure 3(b). With *vertical approach*, only a portion of registers of each warp/thread is allocated in scratchpad memory, and the rest are still allocated in the register file, which can efficiently mitigate the pipeline stalls as well as improve resource utilization.

Challenge 2: *Alleviating the bandwidth mismatch between operand collector and scratchpad memory.* Unlike register file, which can service multiple architectural register access per cycle, the scratchpad memory can only service one architectural register access per cycle in the baseline GPU architecture without bank conflicts (see Section 3.3.2). That will lead to bandwidth mismatch between operand collector and scratchpad memory. Although the *vertical approach* can cut down the number of access to scratchpad memory during instruction execution, it cannot eliminate all access to scratchpad memory for instruction operands. In addition, if there is more than one instruction waiting for operands to be fetched to collector units, and at least one operand of each of these instructions resides in scratchpad memory, then all these operands need to be read from scratchpad memory to collector units sequentially. Obviously, the sequential access to scratchpad memory leaves the SIMD execution unit underutilized due to the lack of ready instructions.

To alleviate the bandwidth mismatch, *EXPARS* proposes a small capacity operand cache (OC) to hold the required registers resided in scratchpad memory. Before instructions can be issued, their operands stored in scratchpad memory are first fetched to OC. The OC has multiple banks so that multiple operands can be read in a single cycle, which is similar to the register file. With the help of OC, accessing operands in scratchpad memory is as fast as register file. To avoid overhead of operand prefetching for each instruction, we use a batch mechanism to prefetch operands for a bundle of instructions, similar to the region-based preloading mechanism [17]. However, the difference is that our prefetching mechanism only preloads a small portion of operands stored in scratchpad memory while Reference [17] needs to preload all operands for all threads from global memory. When the first instruction of a bundle is to be scheduled, all registers of that bundle are already resided in either the register file or OC through our prefetching mechanism.

3 EXPARS METHODOLOGY

In this section, we describe *EXPARS*, an application-transparent mechanism to increase TLP of GPUs by expanding register file to scratchpad memory. If the available register file is not enough to support one more CTA, then part of the scratchpad memory is used as register file so that more concurrent CTAs can be dispatched.

3.1 Design Overview

To effectively increase TLP by exploiting the above ideas, *EXPARS* employs a compiler-hardware co-design. Figure 4 depicts the design overview of *EXPARS* on each SM. All the colored components are extended or modified by *EXPARS* to fulfill the idea of expanding register file to scratchpad memory. We will walk through each component to give a detailed description on how *EXPARS* works.

When the GPU assembler compiles the Parallel Thread Execution (PTX) instructions of a kernel into the low-level GPU Shader ASSembly (SASS) instructions (❶), it divides the instructions into bundles and analyzes the live registers of each bundle. After that, the *Compiler Assisted Annotation* component inserts an annotation before each bundle of instructions and generates the corresponding *.cubin* (❷). At runtime, the kernel in the generated *.cubin* is launched into GPU for execution (❸) and a large number of CTAs are created. Then the *CTA scheduler* analyzes the maximum number of CTAs per SM and dispatches these CTAs to SMs that have enough resources (register file and scratchpad memory, etc.) (❹). When a CTA is to be dispatched to a SM, the *Resource Allocator* analyzes the current resource utilization of the SM, determines where each register should be allocated, and allocates resources for that CTA. If there are registers allocated in scratchpad memory, then the *Register Allocation Table* is updated (❺) to record the allocation information. After the needed resources are allocated, all warps of that CTA are put into the warp pool (❻) and wait to be scheduled. Warps whose bundles are to be executed next and having registers allocated in scratchpad memory are put in the prefetching queue. The first warp in prefetching queue is checked in every cycle by *Register Prefetcher* (❼) to see if all registers allocated in scratchpad memory for the next bundle of that warp (❽) can be accommodated in the remaining space of Operand Cache (❾). If there is enough space, then the needed registers are fetched from scratchpad memory to *Operand Cache* (❿) and that warp is moved from prefetching queue to schedulable queue.

Because not all warps can be scheduled during each cycle, *EXPARS* uses a two-level [5] (TL)-based warp scheduler to schedule warps in schedulable queue (active warps) with greedy-then-oldest (GTO) or loose-round-robbin (LRR) algorithm (⓫). If all instructions of a warp bundle have been scheduled, then that warp is moved to the prefetching queue again to check if all live

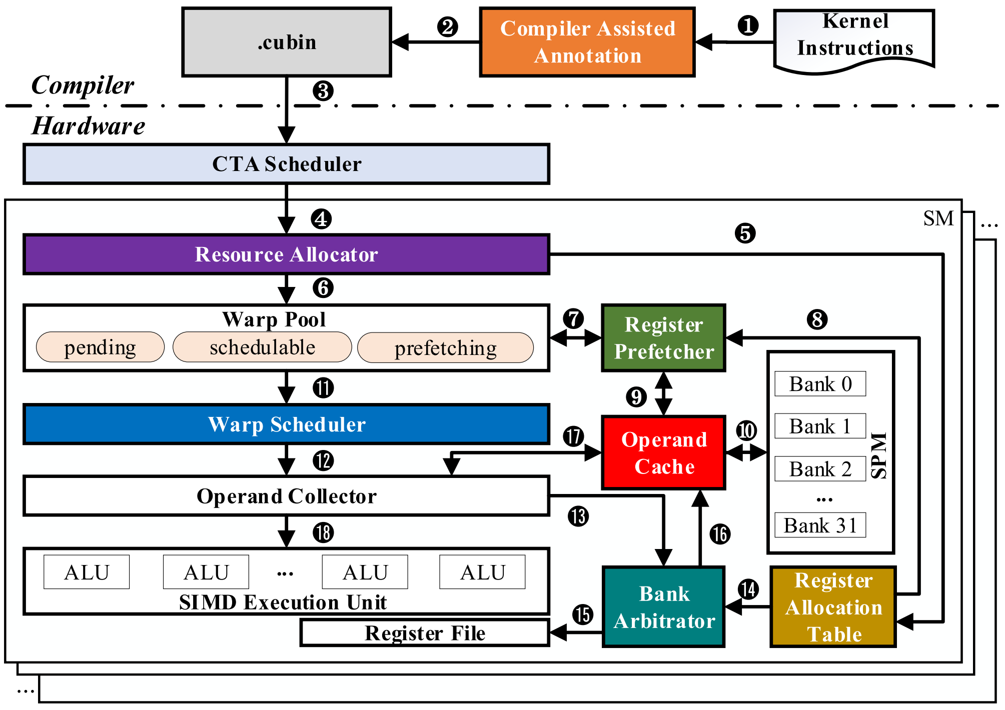

Fig. 4. Design overview of EXPARS. The colored components are extended beyond baseline.

registers of next bundle are ready. When an instruction is scheduled, a collector unit in operand collector is allocated for it (⓬) to read operands. Then the warp ID and architectural register index are sent to *Bank Arbitrator* (⓭) to fetch data either from register file or operand cache. The bank arbitrator checks the warp ID and architectural register index according to *Register Allocation Table* (⓮) to see whether the register is in register file or scratchpad memory. If the register is in register file, then it is read from register file directly (⓯). Otherwise, the read request is sent to operand cache (⓰) and the corresponding data is fetched to operand collector (⓱). After all operands are ready in collector unit, the instruction can be dispatched to the SIMD unit (⓲) for execution.

3.2 Compiler Assisted Annotation

EXPARS relies on compiler to get information about which registers to prefetch for a bundle of instructions. The first capability the compiler provides is to identify all instruction bundles. Although the region creation algorithm [17] can appropriately partition a kernel in its own case, it cannot be applied to *EXPARS* directly. The reason is that which registers will be allocated in register file or scratchpad memory is unknown at compile stage due to the *vertical approach* of register allocation used by *EXPARS*.

In this article, we use a modified version of region creation algorithm to define a bundle, which is based on five rules to identify boundaries of instruction bundles. *(1) barrier or fence operation*, a warp that is waiting at barrier or fence operation can resume execution only when other warps arrives at this point; *(2) long latency operation such as global load*, a long latency operation from a warp can block other warps from prefetching due to the limited capacity of OC; *(3) a bundle should not contain registers may cause line conflicts in OC* (see Section 3.3.4); *(4) the maximum number of registers in a bundle should not exceed a predefined value* to avoid occupying too much capacity of

Table 1. Parameters for TLP Calculation

Parameter	Description
R	Register file capacity on a SM
S	Scratchpad memory capacity (in words, four bytes) on a SM
R_{CTA}	Register file capacity required by a CTA
S_{CTA}	Scratchpad memory capacity (in words, four bytes) required by a CTA
τ	Threshold of maximum # registers allocated in scratchpad memory for a CTA, $0 < \tau < 1$
CTA_{RF}	Number of CTAs with all registers are allocated in register file
CTA_{Mix}	Number of CTAs with part of registers allocated in scratchpad memory
CTA	Maximum number of CTAs to be dispatched on a SM

OC; *(5) a bundle should not span beyond a basic block*, which avoids a single bundle belonging to different control flows.

To record the register information of each bundle, we define a new operation *PREF*, which is accompanied with a bit-vector. The length of bit-vector is equal to the maximum number of registers that GPU functions can use, which is 63 for Fermi architecture [24] and 255 for Maxwell architecture [26]. Each bit in the bit-vector represents an architectural register with the corresponding index. The *PREF* instruction is inserted at the start of each bundle by compiler. Prior work [29] with similar technique shows that code size and performance overhead are negligible with the inserted annotations.

To further reduce the ratio of live registers in a bundle to be allocated in scratchpad memory, we sort the register declarations of each kernel according to the weight of each register. The weight of each register is estimated at compile time by counting the reference number of that register. If a register is referenced frequently, then it means this register is likely to be a hot register and should be allocated at register file. Otherwise, it can be allocated in scratchpad memory.

3.3 Hardware Extension

The bottom part of Figure 4 shows the hardware components of *EXPARS* with the colored ones extended beyond baseline. When a kernel is launched to GPU, the hardware manages resources used by that kernel according to compiler annotations and resource status. The CTA scheduler first analyzes the maximum number of CTAs per SM can hold and dispatches CTAs to each SM. Resource allocator (RA) determines where registers should be allocated and stores the allocation results in register allocation table (RAT). When a bundle of a warp is to be scheduled, the register prefetcher (RP) preloads all live registers of that bundle allocated in scratchpad memory into operand cache (OC). Warp scheduler dispatches all schedulable warps to execute and bank arbitrator (BA) decides where the operands of the issued instructions are to be read based on RAT.

3.3.1 Determining the Max Number of CTAs per SM. To describe how the CTA scheduler determines the maximum number of CTAs per SM can hold with *EXPARS*, we define several notations shown in Table 1. There are two types of CTAs using *EXPARS*, CTA_{RF} and CTA_{Mix}. Since resource requirements of all dispatched CTAs cannot exceed the available capacity, the two types of CTAs have to meet the following conditions defined by Equations (1) and (2):

$$CTA_{RF} \times R_{CTA} + CTA_{Mix} \times R_{CTA} \times (1 - \tau) \leq R, \tag{1}$$

$$CTA_{RF} \times S_{CTA} + CTA_{Mix} \times (S_{CTA} + R_{CTA} \times \tau) \leq S. \tag{2}$$

On the baseline GPU, up to $\lfloor \frac{R}{R_{CTA}} \rfloor$ CTAs can be dispatched to each SM. To ensure that the number of CTAs dispatched by *EXPARS* is no less than that of baseline (CTA_{Lower}), CTA_{RF} and CTA_{Mix} need to satisfy Equation (3):

$$CTA_{RF} + CTA_{Mix} \geq CTA_{Lower} = \left\lfloor \frac{R}{R_{CTA}} \right\rfloor. \tag{3}$$

Based on Equations (1), (2), and (3), we can derive the relation in Equation (4):

$$\left\lfloor \frac{R}{R_{CTA}} \right\rfloor \leq CTA_{RF} + CTA_{Mix} \leq \left\lfloor \frac{R+S}{R_{CTA} + S_{CTA}} \right\rfloor. \tag{4}$$

Although *EXPARS* can improve the number of dispatched CTAs by expanding the register file to scratchpad memory, other factors such as maximum number threads per SM could also limit the actual value of *CTA*. We use the minimum value generated by above factors as the upper bound (CTA_{Upper}) of *CTA* in Equation (5):

$$CTA_{Lower} \leq CTA_{RF} + CTA_{Mix} \leq CTA_{Upper}. \tag{5}$$

To get the maximum number of CTAs that can be dispatched by *EXPARS* to each SM, the sum of CTA_{RF} and CTA_{Mix} should be maximized in Equation (6):

$$\begin{aligned} f(x) &= CTA_{RF} + CTA_{Mix}, \\ CTA &= \max f(x). \end{aligned} \tag{6}$$

The above problem can be transformed into a linear programming problem, which explores the maximum value of Equation (6) satisfying Equations (1), (2), and (5). The results can be divided into three cases as shown in Equations (7), (8), and (9).

(1) If $CTA_{Lower} \geq CTA_{Upper}$, then

$$\begin{cases} CTA_{RF} = \left\lfloor \frac{R}{R_{CTA}} \right\rfloor, \\ CTA_{Mix} = 0. \end{cases} \tag{7}$$

(2) If $CTA_{Lower} < CTA_{Upper}$ and $\lfloor \frac{R}{R_{CTA}(1-\tau)} \rfloor \leq CTA_{Upper}$, then

$$\begin{cases} CTA_{RF} = \left\lfloor \frac{R}{R_{CTA}} \right\rfloor - \left\lfloor \lfloor \frac{R}{R_{CTA}(1-\tau)} \rfloor (1-\tau) \right\rfloor, \\ CTA_{Mix} = \left\lfloor \frac{R}{R_{CTA}(1-\tau)} \right\rfloor. \end{cases} \tag{8}$$

(3) If $CTA_{Lower} < CTA_{Upper}$ and $\lfloor \frac{R}{R_{CTA}(1-\tau)} \rfloor > CTA_{Upper}$, then

$$\begin{cases} CTA_{RF} = \left\lfloor \frac{R}{\tau R_{CTA}} \right\rfloor + CTA_{Upper} - \left\lfloor \frac{CTA_{Upper}}{\tau} \right\rfloor, \\ CTA_{Mix} = \left\lfloor \frac{CTA_{Upper}}{\tau} \right\rfloor - \left\lfloor \frac{R}{\tau R_{CTA}} \right\rfloor. \end{cases} \tag{9}$$

When a kernel is launched, the above conditions are evaluated and the corresponding values of CTA_{RF}, CTA_{Mix}, and *CTA* are calculated. Before such number (*CTA*) of CTAs is dispatched to each SM, all registers of the CTAs (CTA_{RF}) are allocated in register file, and at most τR_{CTA} registers of the CTAs (CTA_{Mix}) are allocated in scratchpad memory and the remaining registers of the CTAs (CTA_{Mix}) are allocated in register file. It should be noted that the CTAs will be scheduled using the default approach (baseline) if *CTA* is not larger than CTA_{Lower}, thus no extra overhead incurs.

3.3.2 Resource Allocator. The resource allocator is mainly responsible for coordinating the register allocation in register file and scratchpad memory. By expanding the register file to scratchpad memory, we can increase the number of CTAs resided in each SM whenever the register file becomes a dominant resource. Figure 5 shows how *EXPARS* coordinates the allocation of registers to increase the number of CTAs resided in each SM. Assuming that each SM has a 32K (128KB) register file and a 48KB scratchpad memory. Each CTA contains 10 warps and requires 10K register

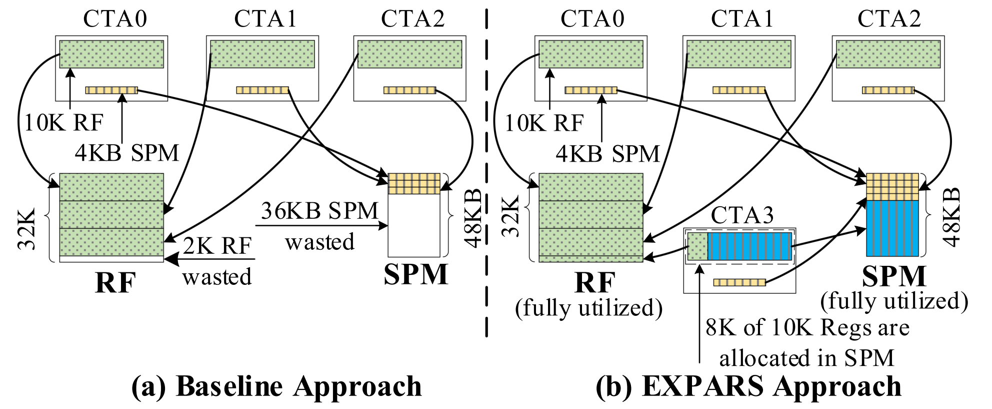

Fig. 5. Resource allocation of register file and scratchpad memory in *Baseline* and *EXPARS* approach.

file and 4KB scratchpad memory. Thus, at most three CTAs can be dispatched on each SM with the baseline resource management approach (shown in Figure 5(a) as the baseline), which leads to 2K register file and 36KB scratchpad memory unused. That is because the remaining 2K register file is not enough to accomodate a complete CTA.

The underutilized resources in Figure 5(a) can be reduced by using our resource allocation approach shown in Figure 5(b). To accommodate one more CTA, there should be 8K more register file. We can observe that there are 36KB scratchpad memory unused, of which we can use 32KB to accommodate the additional 8K register file allocation. In addition to the 8K register file, another 4KB scratchpad memory is also needed according to the resource requirement of a CTA, which is exactly our case in Figure 5(b) that at most 4 CTAs can reside on each SM with our approach.

In GPUs, scratchpad memory is shared at CTA granularity and each CTA allocates a consecutive region in the scratchpad memory. The scratchpad region allocated to each CTA is indexed according to the scratchpad memory base register (SBR) of each CTA, which records the base address for each scratchpad memory region. To store registers in scratchpad memory, another consecutive region in the scratchpad memory is allocated for each CTA. In this article, we use a two-way allocation policy for scratchpad memory, which allocates different resources (scratchpad and register) from top to bottom and from bottom to top, respectively. The free space of scratchpad memory only exists between the scratchpad regions and register regions. Figure 6 illustrates the two-way allocation policy for scratchpad memory allocation.

The scratchpad memory on our baseline GPU has 32 banks and the width of each entry is 32 bits. The 32 successive entries of the 32 banks make up a line. Thus, each line can store one architectural register, which contains 32 registers with the same register index for the 32 threads of a warp. Architectural registers belonging to the same warp are laid out consecutively by the order of register index. Warps belonging to the same CTA are laid out consecutively by the order of warp ID. Registers stored in scratchpad memory are accessed by addresses. Register address is computed based on SBR, warp ID, architectural register index, and so on (see Section 3.3.3). When an architectural register is to be accessed, 32 scratchpad memory addresses are generated for access. These 32 scratchpad memory addresses span across 32 distinct banks, thus they can be accessed simultaneously within a cycle without bank conflicts.

3.3.3 *Register Allocation Table.*

The purpose of the Register Allocation Table (RAT), shown in Figure 7, is to maintain metadata for CTAs to control the register prefetching process and find the addresses of registers allocated in scratchpad memory. The RAT maintains four variables,

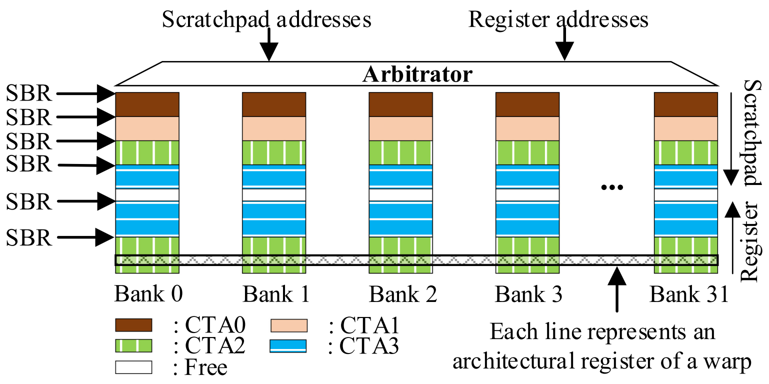

Fig. 6. Two-way allocation policy for scratchpad memory. CTA0 and CTA1 are default CTAs with all their registers allocated in register file, whereas CTA2 and CTA3 are additional CTAs enabled by *EXPARS*, with part of their registers allocated in scratchpad memory.

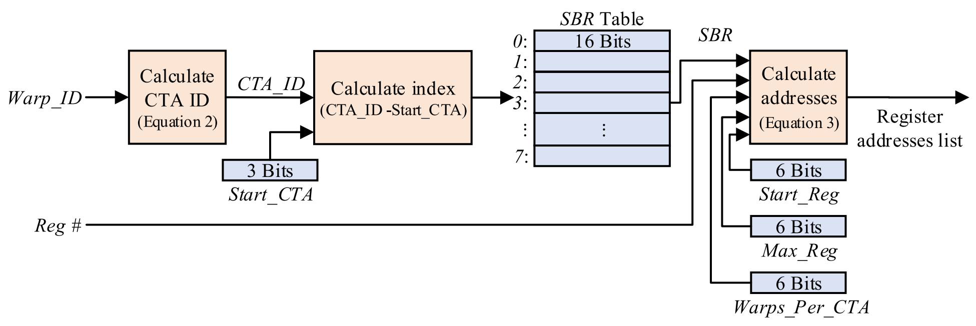

Fig. 7. Register allocation table.

Start_CTA, *Start_Reg*, *Max_Reg*, and *Warps_Per_CTA*, and one table, *SBR*. *Start_CTA*, which is equal to CTA_{RF} in Section 3.3.1, records the starting physical ID of CTAs that have registers allocated in scratchpad memory, and *Start_Reg* records the starting architectural register index allocated in scratchpad memory. *Start_Reg* is calculated by resource allocator using Equation (10):

$$Start\_Reg = \left\lfloor \frac{R - (CTA_{RF} \times R_{CTA})}{CTA_{Mix}} \right\rfloor. \tag{10}$$

Max_Reg records the maximum architectural register index of the CTA and *Warps_Per_CTA* records the warp number of each CTA. The *SBR* table records the values of scratchpad memory base registers, which are the base addresses of the consecutive regions allocated to CTAs to store their registers. The *SBR* table is indexed by relative CTA ID (*CTA_ID* minus *Start_CTA*). According to the baseline configuration in Table 2, because the maximum number of concurrent CTAs and threads in each SM is 8 and 1,536, respectively, *Start_CTA* takes 3 bits, *Warps_Per_CTA* takes 6 bits, and *SBR* table has eight entries (each entry is 16 bits in width to cover the 48KB scratchpad memory in each SM); because the maximum number of available architectural registers for each warp is 63, registers *Start_Reg* and *Max_Reg* need 6 bits each. In *EXPARS*, register declarations are sorted based on their reference weights in descending order by compiler. If *m* registers are allocated in scratchpad memory, then these *m* registers are the last *m* registers with least reference weights. This means the registers starting from *Start_Reg* in a CTA are all allocated in scratchpad

Table 2. Major Parameters of the Simulated System

Parameter	Configuration
System	15 SMs, 1.4GHz
SM	Fermi: 1536 threads/SM, 8 CTAs/SM; Maxwell: 2,048 threads/SM, 32 CTAs/SM
Warp scheduler	LRR/GTO/TL (2 schedulers per SM)
Registers / SM	Fermi: 32,678 (32,678*4B = 128KB); Maxwell: 65,536 (65,536*4B = 256KB)
Scratchpad / SM	Fermi: 48KB; Maxwell: 64KB
On-chip cache	L1: 16KB, L2: 768KB
GDDR memory	6 MCs, 16 banks, 924MHz, T_{RRD}=6, T_{RCD}=12, T_{RP}=12, T_{RC}=40, T_{CL}=12, T_{WR}=12

memory. Therefore, the RAT cannot only be used to determine whether a register is allocated in register file or scratchpad memory but also to calculate the register addresses in scratchpad memory. When allocating a scratchpad memory region for a CTA (*CTA_ID*), the value of *SBR* of that region is calculated using Equation (11). In Equation (11), *S* is the capacity of scratchpad memory:

$$SBR = S - (CTA\_ID - Start\_CTA + 1) \times (Max\_Reg - Start\_Reg + 1) \times Warps\_Per\_CTA \times 128. \tag{11}$$

When a warp (*Warp_ID*) accesses the register (*Reg*) allocated in scratchpad memory, Equation (12) is used first to calculate the CTA (*CTA_ID*) it belongs to. The *SBR* indexed by (*CTA_ID* − *Start_CTA*) can be extracted from the table. Then the first register address (*Address*) is calculated using Equation (13). Based on the first register address, the remaining 31 addresses are generated by increasing their preceding addresses by 4:

$$CTA\_ID = \left\lfloor \frac{Warp\_ID}{Warps\_Per\_CTA} \right\rfloor, \tag{12}$$

$$\begin{aligned} Address = SBR &+ (Max\_Reg - Start\_Reg + 1) \times (Warp\_ID \bmod \\ &Warps\_Per\_CTA) \times 128 + (Reg - Start\_Reg) \times 128. \end{aligned} \tag{13}$$

3.3.4 Register Prefetcher and Operand Cache. When a warp reaches a *PREF* instruction, the instruction decoder sends the register information of the instruction to the RP. The RP keeps the register information in the corresponding position of its register bit-vector table. The register bit-vector table maintained by RP is indexed by warp ID. Before a warp becomes schedulable, the RP guarantees all registers, which are parsed from the corresponding bit-vector, needed by the warp's next bundle are either in register file or OC. If the registers are allocated in the register file, then they are already there. Otherwise, registers need to be fetched from scratchpad memory to OC. In *EXPARS*, the warp pool maintains three queues (schedulable, prefetching and pending) to track the status of all supervised warps. If all registers of the next bundle of a warp are allocated in register file, then that warp will be put in the schedulable queue. Otherwise, it is put in the prefetching queue. Warps in the schedulable queue can be scheduled directly by scheduler for execution. When a bundle of a warp completes, if the next bundle contains registers allocated in scratchpad memory, it is moved to the front of prefetching queue.

In each cycle, the RP checks the head warp of the prefetching queue to see if there is enough free space in OC to hold all registers allocated in scratchpad memory for the next bundle of that warp. If there is enough space in OC, then the addresses of all registers are calculated by OC and these registers are fetched from scratchpad memory to OC. Otherwise, registers of previously completed bundles will be evicted from OC. Figure 8 shows the architecture of OC. The OC maintains a similar structure as the register file that multiple registers in different banks of OC can be accessed

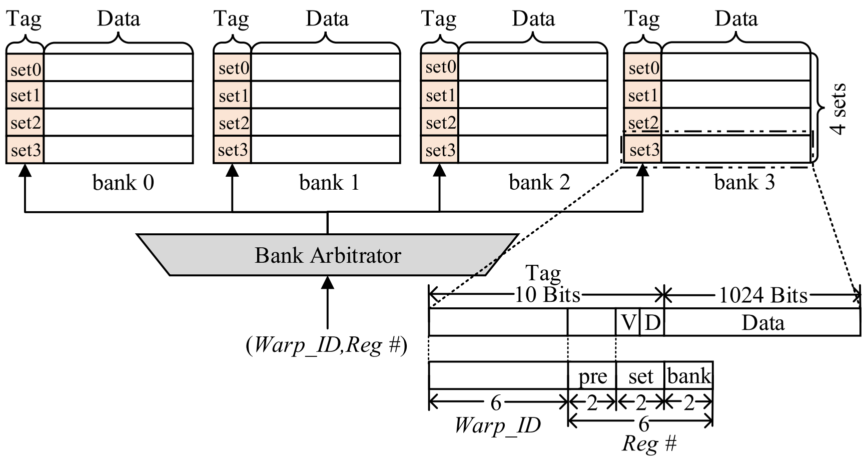

Fig. 8. Operand cache architecture.

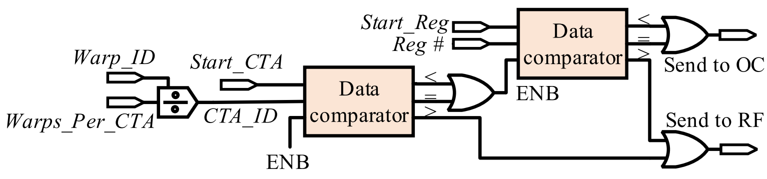

Fig. 9. Judgement logic of bank arbitrator.

simultaneously. In Figure 8, the OC has four banks and the four ways of each set are distributed in the four banks. Each line of OC is 1,024 bits to hold one architectural register for a warp. The tag, which takes 10 bits for each cache line, is composed of 6 bits of *Warp_ID*, the higher 2 bits of *Reg*, 1 bit valid flag, and 1 bit dirty flag. To preserve the compiler assigned bank information, registers prefetched from scratchpad memory are put to the same bank of OC as the register file. After all registers required by next bundle are ready in OC or register file, the warp is moved to the schedulable queue and waits to be scheduled. If the number of active warps in schedulable queue has reached the maximum value, then that warp will be moved to pending queue.

3.3.5 Bank Arbitrator. The bank arbitrator (BA) contains a read request queue for each register file bank to hold all access requests (*Warp_ID*, *Reg#*) until they are granted. During each cycle, the BA selects a group of non-conflicting accesses to send to the register file. To read registers allocated in scratchpad memory, we extend the BA with a judgement logic (illustrated in Figure 9). When an access request arrives, the BA checks the RAT to decide whether the needed register is allocated in register file or scratchpad memory. The BA first computes the *CTA_ID* according to *Warp_ID* using Equation (12). If *Start_CTA* is greater than *CTA_ID*, then it means all registers of that CTA are allocated in register file and the access request will be sent to register file. If *Start_CTA* is not greater than *CTA_ID* and the architectural register index *Reg#* is not less than *Start_Reg*, then the access request will also be sent to register file. Otherwise, the access request will be sent to the OC. When a register access request will be sent to the OC, the bank arbitrator first determines the OC bank of requested register according to the lowest 2 bits of *Reg#*, then sends the request to the corresponding set of the determined bank according to the middle 2 bits of *Reg#* (illustrated in Figure 8). Because all registers allocated in scratchpad memory are serviced by the OC during the execution of a bundle, the tag lookup can be eliminated during operand reading.

However, when the content of an architectural register changes or an architectural register should be evicted during the prefetching stage, the corresponding tag will be looked up.

3.3.6 A Lazy Two-Level Warp Scheduler. EXPARS can improve TLP by enabling more CTAs per SM through expanding register file to scratchpad memory. However, previous works [5, 13, 19, 43] have shown that higher TLP does not always mean higher performance due to resource contention. To alleviate the contention, we propose a *Lazy Two-Level Warp Scheduler* (LTLWS), which is inspired by Reference [19], to control the maximum number of schedulable warps (active warps) during runtime. LTLWS is designed based on the *two-level* (TL) scheduler that schedules warps into three queues such as schedulable queue, prefetching queue, and pending queue with its outer level scheduler. Warps in the schedulable queue can be scheduled for execution, whereas warps stalled by long latency operations or exceeding the maximum number of active warps are queued in the pending queue. Warps with registers to be preloaded for the next bundle of instructions are put in the prefetching queue.

Warps in the schedulable queue are scheduled by the inner level scheduler of LTLWS using either GTO (LTLWS-GTO) or LRR (LTLWS-LRR) algorithm. Different from Reference [5], where the number of entries (active warps) for the inner level scheduler is fixed, our approach dynamically adjusts the number of active warps during runtime, which achieves higher TLP without degrading the performance due to severe resource contention. When a kernel is launched, the maximum number of CTAs dispatched to each SM can be calculated using the approach presented in Section 3.3.1. Initially, the maximum number of active (schedulable) warps for LTLWS is not set and all warps with registers ready in register file or OC for the next instruction bundles are put in the schedulable queue.

The instructions issued for each warp are recorded until the first warp finishes execution. We define W_{Max} as the number of warps involved in the LTLWS scheduler and $Inst_i$ as the number instructions issued for warp i (W_i) when the first warp finishes execution. The $Inst_{Max}$ is defined as the number of instructions issued from the first completed warp. To be optimal, the maximum number (W_{Opt}) of active warps that can be scheduled by the inner level scheduler of LTLWS is obtained using Equation (14). After W_{Opt} is obtained, the maximum number of active warps for LTLWS is set to W_{Opt}. Then, LTLWS degenerates into TL, which introduces negligible overhead, and the kernel keeps on running to completion with up to W_{Opt} warps can be scheduled in each cycle:

$$W_{Opt} = \left\lfloor \frac{\sum_{i=1}^{W_{Max}} Inst_i}{Inst_{Max}} \right\rfloor. \tag{14}$$

3.3.7 Hardware Cost. EXPARS requires a small OC, which has the same number of banks with register file, for each SM to cache registers allocated in scratchpad memory. By default, it is set to 2KB and partitioned into four sets, and each set has four lines distributed in four banks (illustrated in Figure 8) if the register file has four banks. Each line needs a 10-bit tag, thus, the total tags for each SM take 20B storage. Besides, there also need a RAT for each SM to maintain metadata for CTAs to control the register prefetching process and find the addresses of registers allocated in scratchpad memory. The RP maintains a register bit-vector table, which supports 48 warps with 63 registers per warp in our baseline configuration, and its storage overhead is 378B. The RAT contains a *SBR* table to record base addresses of scratchpad memory regions allocated to CTAs to store registers. In our baseline configuration, the *SBR* table has eight entries and each entry needs 2B and the total storage overhead of *SBR* table for each SM is 16B. In addition, the RAT also maintains one 3-bit variable *Start_CTA* and three 6-bit variables *Start_Reg*, *Max_Reg* and

Warps_Per_CTA. Also, with LTLWS scheduler, each warp requires one 4B counter to record the number of executed instructions and there is also a 6-bit variable to record the optimal number of active warps. In addition to above storage cost, *EXPARS* needs an arithmetic circuit to determine the maximum number of CTAs using Equations (7), (8), and (9) for each SM before CTAs are dispatched to SMs. Meanwhile, the modified bank arbitrator needs two additional comparator circuits to judge the position of needed registers.

3.4 Compiler-Only Approach vs. EXPARS Approach

Although the compiler can also spill registers to scratchpad memory, the compiler cannot determine how many and which architectural registers should be spilled to scratchpad memory. That is because whether the TLP is limited by register file or not depends not only on required register file and scratchpad memory but also on CTA size. For example, assuming that each thread of a kernel needs 20 registers and each CTA needs 10KB scratchpad memory. If the CTA size is 512, then the TLP will be limited by register file in the baseline architecture and registers can be spilled to scratchpad memory to increase TLP. If the CTA size is changed to 256, then the TLP of the kernel will be limited by scratchpad memory and no registers need to be spilled to scratchpad memory; otherwise, unnecessary performance loss will occur. However, the compiler is usually unaware of the CTA size, which may be changed with the input. Thus, it is infeasible to rely on compiler-only approach to spill registers to scratchpad memory efficiently, whereas *EXPARS* spills registers to scratchpad memory efficiently to increase TLP according to runtime information. Besides, even if the CTA size can be passed to the compiler with certain methods, the challenge of bandwidth mismatch discussed in Section 2.2 still needs to be addressed by the hardware.

4 EVALUATION

4.1 Experimental Setup

We model *EXPARS* with GPGPU-Sim v3.2.2 [1]. Table 2 presents the major parameters of the simulated system. Except for Section 4.6, all results are generated using the Fermi [24] configuration, which is the mostly targeted configuration for GPU research even in recent publications [9, 10, 12, 16, 31, 35, 41]. Note that although the simulations are based on Fermi architecture, the principles behind *EXPARS* are also applicable to newer architectures such as Kepler, Maxwell and Pascal. The evaluation of *EXPARS* for newer GPU architecture is presented in Section 4.6. Since the PTX assembler (ptxas) is closed-source for CUDA applications, we implement the compiler extension based on the PTXPlus, which relies on ptxas for register allocation, of GPGPU-Sim compilation system. We use GPUWatch [21] to model the GPU power consumption. The die area overhead of *EXPARS* is estimated by using CACTI v7.0 [37] with 40nm technology. The overhead including logic, operand cache, and register allocation table is considered. The total area overhead is about 0.861mm^2 for all 15 SMs, which is only 0.163% of the die area of the GTX 480.

We evaluate *EXPARS* on several widely adopted benchmark suites including Rodinia [2], Nvidia CUDA-SDK [25], and Parboil [30] to cover a wide range of application domains. This article mainly targets on register-sensitive workloads such as workloads in References [9, 10, 38]. For each selected workload from these benchmark suites, we report the number of warps per CTA, the number of CTAs per SM, the register file, and scratchpad memory usage per CTA in Table 3. Since PTX-Plus is currently only compatible with CUDA Compute Capability less than 2.0, all the workloads are compiled with CUDA 4.2 and GCC 4.4 for Compute Capability 1.3. We set τ to be 0.8 based on empirical studies and the size of OC is 2KB. We compare *EXPARS* with two other approaches: *Baseline* and *Warpman* [38]. *Baseline* represents the default GPU implementation, whereas *Warpman* is the state-of-the-art approach to improve the utilization of register file as well as increase

Table 3. Benchmarks Description

Application	Kernel	Abbr.	CTAs/SM	Warps/CTA	SPM/CTA	RF/CTA
lbm [30]	performStreamCollide_kernel	LBM	7	4	0	4608
stencil [30]	block2D_hybrid_coarsen_x	ST	2	16	0	14,436
mri-q [30]	ComputeQ_GPU	MQ	4	8	0	7,168
sgemm [30]	mysgemmNT	SGE	5	4	512	5,632
b+tree [2]	findRangeK	BT	2	16	0	12,288
hotspot [2]	calculate_temp	HS	3	8	3,072	9,216
leukocyte [2]	GICOV_kernel	LEUK	7	6	0	4,608
MonteCarlo [25]	MonteCarloOneBlockPerOption	MC	5	8	2,048	6,144
convolutiontexture [25]	convolutionRowsKernel	CONV	7	6	0	4,608
EstimatePiInlineP [25]	initRNGP17curandState	EST	5	8	0	6,144
mergeSort [25]	mergeSortSharedKernel	MERG	2	16	8,192	12,288
quasirandomGenerator [25]	quasirandomGeneratorKernel	QUA	2	12	0	12,288
singleAsianOptionP [25]	initRNGP17curandState	SING1	5	8	0	6,144
singleAsianOptionP [25]	generatePathsIfEvPT_P17curandStante	SING2	4	8	0	7,168

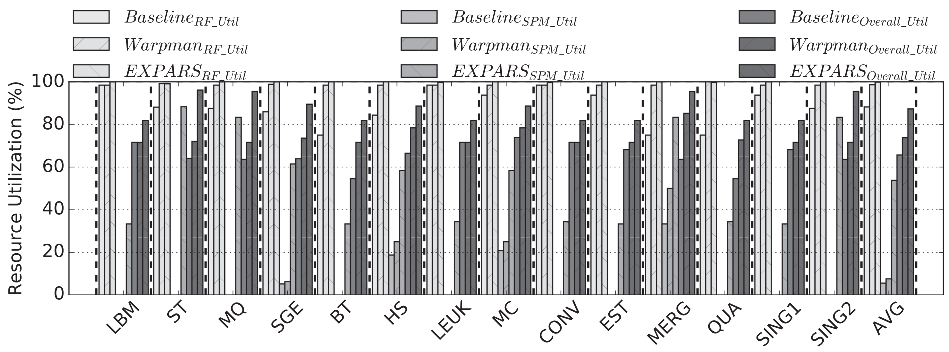

Fig. 10. Comparison of resource utilization among *Baseline*, *Warpman*, and *EXPARS*.

the TLP by scheduling threads at the warp level. We evaluate each approach with three different warp schedulers: LRR, GTO, and TL.

4.2 Resource Utilization

Figure 10 compares the resource utilization of *Baseline*, *Warpman*, and *EXPARS*. We show the register file utilization, scratchpad memory utilization, and overall resource utilization for each benchmark. The overall resource utilization is the total utilization of register file and scratchpad memory. It is observed in Figure 10 that *EXPARS* can make full utilization (100%) of the register file for almost all benchmarks. The average register file utilization of *Baseline*, *Warpman*, and *EXPARS* is 88.21%, 98.62%, and 99.85%, respectively. The high register file utilization can be attributed to the *vertical approach* of register allocation in *EXPARS*, which uses the allocation granularity smaller than *Warpman* and thus leads to less resource fragmentation in most cases.

In addition, the average utilization of scratchpad memory has also been significantly improved, which increases from 5.58% (*Baseline*) to 53.78% (*EXPARS*). Whereas, *Warpman* can only improve the average utilization of scratchpad memory to 7.59%. Moreover, with *EXPARS*, the overall resource utilization for each benchmark is more than 80%, whereas for *Warpman*, only one

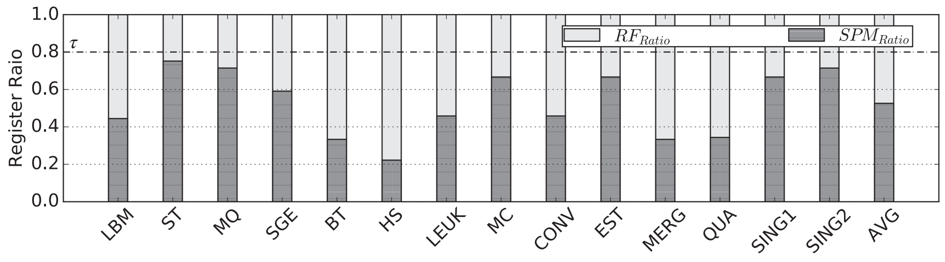

Fig. 11. The ratio of registers allocated in register file and scratchpad memory of each mix CTA.

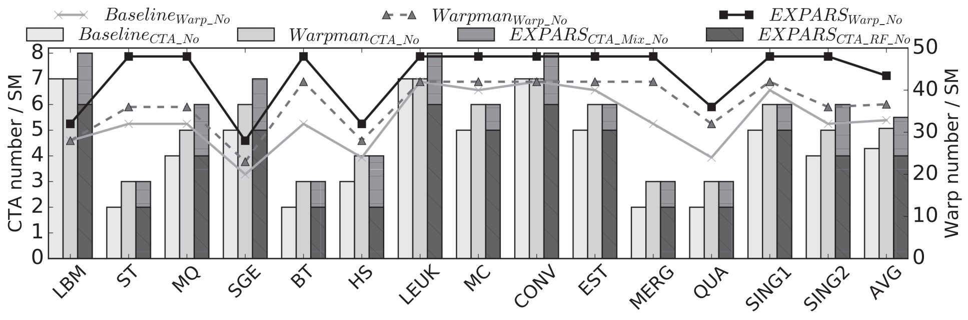

Fig. 12. Comparison of TLP among *Baseline*, *Warpman*, and *EXPARS*.

benchmark exceeds 80% (*MERG*). In general, the average utilization of overall resource for *Baseline*, *Warpman*, and *EXPARS* is 65.68%, 73.80%, and 87.28%, respectively. The result shows that *EXPARS* can effectively improve the resource utilization on GPUs.

As described in Section 3.3.1, The value of CTA_{RF} and CTA_{Mix} are calculated by *EXPARS* under the restriction of threshold τ. Figure 11 demonstrates the ratio of registers allocated in register file and scratchpad memory of each mix CTA in each benchmark. We can see that all benchmarks satisfy the threshold τ (0.8 in our experiments) and the average ratio of registers allocated in register file of each mix CTA is 47.39%, which means that nearly half of registers in each mix CTA can remain allocated in register file.

4.3 Thread Level Parallelism

Figure 12 shows the TLP comparison among *Baseline*, *Warpman*, and *EXPARS*, which includes the maximum number of CTAs per SM and the maximum number of warps per SM. In Figure 12, the maximum number of CTAs for all benchmarks has increased with *EXPARS* compared to *Baseline* and *Warpman*. We notice that with *EXPARS*, for *LBM*, *HS*, *LEUK*, and *CONV*, the number of CTAs whose registers are entirely allocated in register file (CTA_{RF}) is one less than *Baseline*. The reason is that for these benchmarks, either the register file utilization in *Baseline* (see Figure 10) or the register requirement for each CTA (see Table 3) is relatively high. There is not enough register file for increasing the number of CTAs while satisfying the threshold τ. Therefore, with *EXPARS*, the number of CTA_{RF} is reduced to free enough register file for dispatching more CTAs. On average, the maximum number of CTAs for *Baseline*, *Warpman*, and *EXPARS* is 4.29, 5.07, and 5.50, respectively.

However, when comparing the maximum number of warps, we can see that *EXPARS* can dispatch more warps for all benchmarks than *Baseline* and *Warpman*. With *Warpman*, although it can dispatch one more CTA for each SM when resources are not fully utilized, the additional CTA is a partial CTA and only part of the warps can be allocated with resources to execute on each

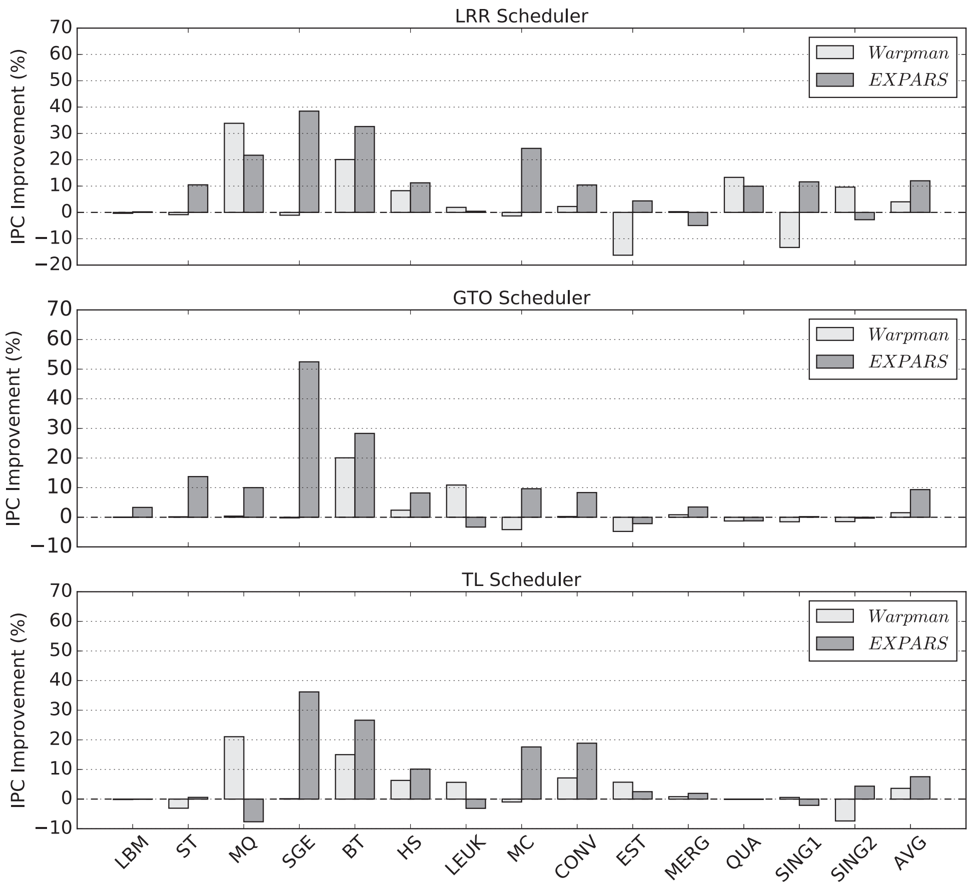

Fig. 13. Comparison of performance improvement between *Warpman* and *EXPARS* with different warp schedulers. The results are normalized to *Baseline*.

SM. Different from *Warpman*, *EXPARS* can dispatch more CTAs through expanding register file to scratchpad memory, and all CTAs dispatched to each SM by *EXPARS* are full CTAs. This means all warps of dispatched CTAs can get the required resources to execute on SMs. In the cases of *ST*, *MQ*, and *SING2*, *EXPARS* can dispatch 48 warps, which reaches the warp limit (see Table 2), whereas *Baseline* and *Warpman* can only dispatch at most 32 and 36 warps, respectively. In addition, 10 out of the 14 benchmarks with *EXPARS* can make full use of the 1,536 thread slots, whereas none of the *Baseline* and *Warpman* approach can achieve that large number of warps. On average, the maximum number of warps for *Baseline*, *Warpman,* and *EXPARS* is 32.86, 36.64, and 43.43, respectively.

4.4 Performance Improvement

In this section, we evaluate the performance impact of *EXPARS* with results shown in Figure 13. To avoid the bias toward a particular scheduling strategy, we evaluate *EXPARS* with three different state-of-the-art warp schedulers (LRR, GTO, and TL). As seen from Figure 13, *EXPARS* achieves better performance improvement over *Warpman* for most benchmarks due to the increased TLP. For *BT*, its performance improvement with *EXPARS* using all three schedulers is about 30%, whereas with *Warpman* the performance improvement is less than 20%. Moreover, *SGE* achieves about 52% and 40% performance improvement using GTO and LRR/TL scheduler, respectively, with *EXPARS*.

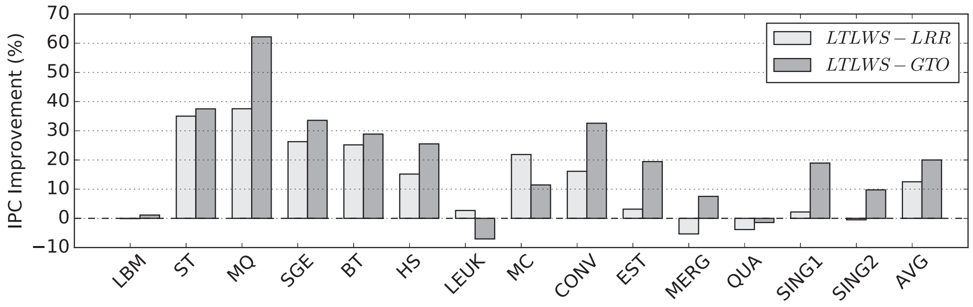

Fig. 14. Performance improvement of *EXPARS* with two optimized warp schedulers. The results are normalized to *Baseline*.

Note that for benchmarks *EST* and *SING1* using LRR scheduler, *Warpman* even losses performance significantly compared to *Baseline*. The reason is that although *Warpman* can dispatch a partial CTA to improve TLP, scheduling partial warps of CTA may destroy the intra-CTA locality, and thus causes performance degradation, especially when the LRR scheduler is used. On average, *EXPARS* achieves 12.01%, 9.34%, and 7.55% performance improvement with LRR, GTO, and TL scheduler, respectively.

It is observed that although the TLP of *MQ* is better with *EXPARS* than *Warpman*, the performance is, on the contrary, better with *Warpman* using LRR/TL scheduler. The reason is that higher TLP may lead to severe contention on shared resources such as cache and memory bandwidth. The above reason also applies to *LEUK* with GTO/TL scheduler, and *MERG* and *SING2* with LRR scheduler. To alleviate the resource contention caused by the increased TLP, we use the *LTLWS* scheduler to optimize the warp scheduling. *Warpman* can use the SM-dueling approach [18] or a simple static threshold to avoid the same resource contention problem; however, it is not evaluated in this article. Figure 14 shows the results of performance improvement using LTLWSs (LTLWS-LRR and LTLWS-GTO). Both LTLWS-LRR and LTLWS-GTO can effectively alleviate the resource contention. Especially for benchmark *MQ*, it achieves 37.54% and 62.18% performance improvement when using LTLWS-LRR and LTLWS-GTO, respectively. On average, LTLWS-LRR and LTLWS-GTO improve performance by 12.53% and 20.01%, respectively. We have also expanded the register file of both *Baseline* and *Warpman* by 2KB, which is the size of operand cache, and we find that the increased capacity has no effect on the performance of *Baseline* and the performance improvement of *Warpman* is less than 1% on average.

4.5 Energy Consumption

There are two components may cause additional energy overhead: *Scratchpad Memory* and *Operand Cache*. First, because part of the scratchpad memory is used as register file, the number of scratchpad memory access is increased to fetch register data stored in it for operand collector. Second, *EXPARS* introduces an operand cache between scratchpad memory and operand collector to address the bandwidth mismatch. Each register stored in scratchpad memory is first fetched from scratchpad memory to operand cache before it can be read by operand collector, thus the number of both scratchpad memory and operand cache accesses increases. When there is not enough free space in operand cache, registers of inactive warps need to be evicted, and the eviction also increases the number of both scratchpad memory and operand cache accesses. Although the two components can generate additional energy consumption compared to the *Baseline*, the performance improvement with *EXPARS* offsets the energy cost and reading operands from the small operand cache instead of the large register file can also reduce energy consumption to a certain

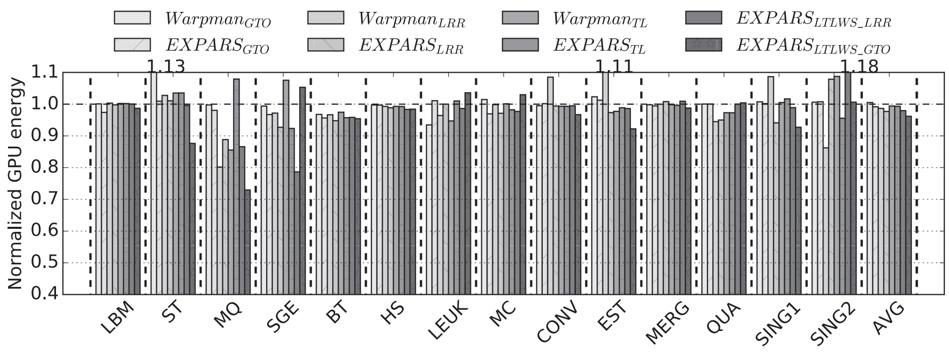

Fig. 15. The total energy consumption of *EXPARS* with different warp schedulers. The results are normalized to *Baseline*.

Table 4. The Average Capacity of On-chip Memory (Register File and Scratchpad Memory) Required to Maximize TLP for All Benchmarks in Rodinia [2] and Parboil [30] on the Fermi and Maxwell Architectures

Benchmark suite		Average required RF		Average required SPM	
		Fermi	Maxwell	Fermi	Maxwell
Rodinia	All	1.60×	1.30×	0.57×	0.57×
	Register-Sensitive	1.86×	2.07×	0.46×	0.61×
Parboil	All	1.64×	1.27×	0.37×	0.37×
	Register-Sensitive	1.82×	1.94×	0.43×	0.34×

extend, thus the overall energy consumption actually reduces apparently. As shown in Figure 15, on average, the energy consumption with *EXPARS* is less than both *Baseline* and *Warpman*. Especially when the LTLWS-LRR scheduler is used with *EXPARS*, *MQ* reduces energy consumption by 27%, whereas *ST*, *EST* and *SING1* achieve about 10% energy saving. In general, *EXPARS* introduces negligible energy consumption compared to *Baseline* and *Warpman*. We also note that the energy consumption can be reduced further by appropriately optimizing the replacement strategy of OC through operand-liveness analysis. We leave this exploration to future work.

4.6 Evaluation for Advanced Architecture

To evaluate the effectiveness of applying *EXPARS* to advanced architecture, we first report the impact of on-chip memory (register file and scratchpad memory) capacity on TLP on Fermi and Maxwell architectures. To get the register and shared memory usage, we compile all benchmarks in Rodinia [2] and Parboil [30] for Fermi (with "-arch=sm_13 –resource-usage" option) and Maxwell (with "-arch=sm_52 –resource-usage" option) architectures with the recent released CUDA compiler.[1] Then, we calculate the capacities of register file and scratchpad memory benchmarks would require if there are no register file and scratchpad memory size constraints. Table 4 shows the average capacity of register file and scratchpad memory needed to maximize TLP for all benchmarks in Rodinia [2] and Parboil [30] on Fermi and Maxwell architectures. This experiment illustrates that the larger register file on Maxwell architecture does not alleviate the problem that the TLP of benchmarks, especially register-sensitive ones, is limited by the insufficient register file. The

[1]The maximum amount of registers that GPU functions can use are 63 and 255 for Fermi and Maxwell architecture, respectively.

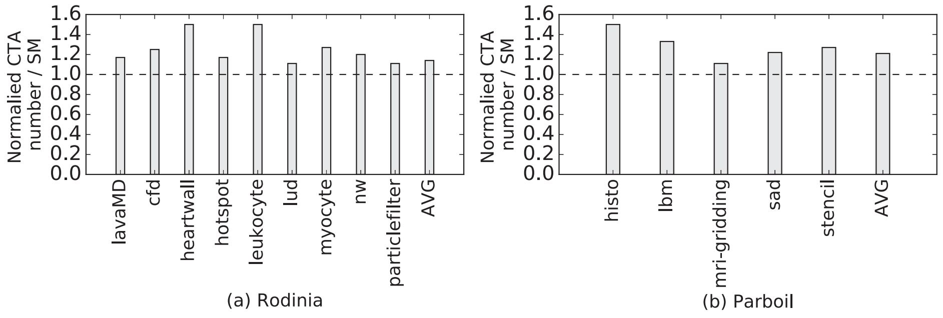

Fig. 16. The TLP of register-sensitive benchmarks in Rodinia and Parboil using the resource allocation strategy in *EXPARS* on Maxwell architecture. The results are normalized to *Baseline*.

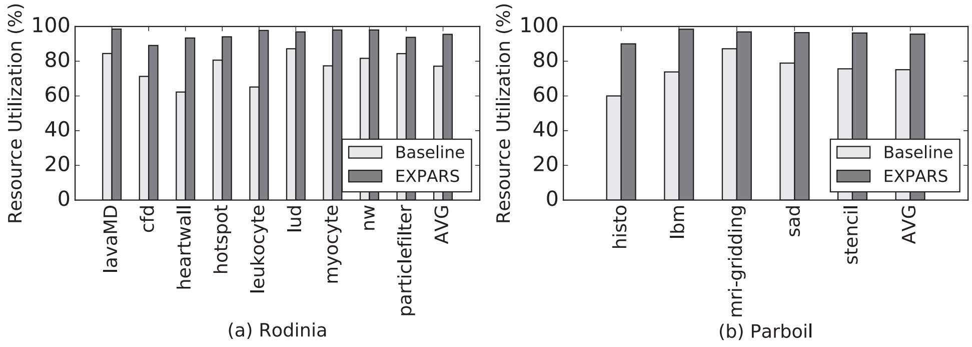

Fig. 17. The overall resource utilization of register-sensitive benchmarks in Rodinia and Parboil on Nvidia Maxwell architecture.

reason for that is more registers can be provided to each kernel on the Maxwell architecture and the CUDA compiler employs more optimization technologies to make full use of the provided registers for higher performance. Similar results have been observed in References [22, 29]. Table 4 also shows that the required capacity of register file is far more than the required capacity of scratchpad memory to maximize the TLP on average, and the utilization of scratchpad memory is quite low even when the TLP is maximized. The results imply that the capacity of register file still remains as one of the major factors limiting TLP even on Maxwell architecture. Thus, it is still effective to expand register file to the underutilized scratchpad memory using *EXPARS* to improve both the TLP and utilization of on-chip memory.

To further understand the advantages of applying *EXPARS* to advanced architecture, we use the resource allocation strategy in *EXPARS* to see how the TLP and overall utilization of on-chip memory are improved on Maxwell architecture. Figure 16 shows the normalized TLP of all register file sensitive benchmarks in Rodinia and Parboil. We can see that the normalized TLPs of Rodinia and Parboil are improved to 1.14 and 1.21, respectively. In Figure 17, we can see that the overall utilization of on-chip memory for Rodinia and Parboil is increased from 77.13% to 95.48% and from 75.11% to 95.61% on average, respectively. From above results, we can conclude that both Fermi and advanced architecture have similar problem that the capacity of register file is one of the major factors limiting the TLP, whereas the scratchpad memory is usually underutilized, which makes *EXPARS* still effective to address such problem.

5 RELATED WORK

GPU resource underutilization: Yang et al. [40] propose software and hardware approaches to maximize the utilization of scratchpad memory by allocating and de-allocating scratchpad memory dynamically. Xiang et al. [38] present a fine-grained warp-level resource management approach by dispatching one partial CTA to alleviate the resource fragmentation problem. Jatala et al. [9] introduce a resource sharing mechanism to minimize register and scratchpad wastage and resources are shared between CTAs through a time-multiplexing manner. Vijaykumar et al. [35] propose a software-hardware codesign resource virtualization framework to enable dynamic and fine-grained control over resource management. Yoon et al. [41] breakthrough the scheduling limit to propose a virtual thread architecture to assign CTAs up to the capacity limit, which is complementary to *EXPARS*. GPUDuet [22] is a recent work that leverages context switching to achieve higher levels of TLP. There are two key differences between GPUDuet and *EXPARS*. First, if the TLP is limited by register file, GPUDuet switches out stalled warps to underutilized on-chip resources (L1 D-cache and scratchpad memory) and launches new warps. Whereas *EXPARS* expands register file to underutilized scratchpad memory to increase the logic capacity of register file and accomodate more CTAs. Second, in GPUDuet, no warp in extra CTAs can be scheduled until one active warp is switched out or finished. Whereas *EXPARS* can schedule all warps in all launched CTAs. Erez et al. [27, 28] put forward novel solutions (*SIMD lane permutation* and *dual-path execution*) to the control flow divergence problem to improve SIMD resource utilization. There are also works [3, 4, 42] targeting at improving resource utilization of multitasking GPUs.

GPU register file management: Through register lifetime analyzing, Jeon et al. [10] propose a register file virtualization mechanism that can dynamically release dead registers from one warp and allocate them to another warp to shrink the register file capacity. Hayes et al. [8] propose an unified on-chip memory allocation framework for compilers to offload register pressure to scratchpad memory. Xie et al. [39] propose a similar compiler-based framework to coordinate register allocation and spill variables to global memory and scratchpad memory to satisfy the register per-thread limit. Gebhart et al. [5] introduce a register file cache between the main register file and SIMD unit to reduce the number of accesses to the main register file. To minimize the energy consumed by the large register file, Kloosterman et al. [17] replace the large register file with a small operand staging unit to only cache live registers of active regions. To improve the hit rate of register file cache and tolerate large register file latency, Sadrosadati et al. [29] partition instructions into intervals and prefetch register working-set from the main register file to the register file cache at the beginning of each interval, which is similar to Reference [17]. In this article, we also adopt a similar register file cache (operand cache), the main difference is that *EXPARS* targets improved resource utilization and TLP, and *EXPARS* only prefetchs registers allocated in scratchpad memory to the operand cache before each instruction bundle can be scheduled.

GPU memory unification: Different applications have different requirements for different on-chip memories. Nvidia Fermi [24] and Kepler [15] are equipped with a 64KB unified on-chip memory for each SM that can be configured either as 48KB of scratchpad memory and 16KB cache or 16KB of scratchpad memory and 48KB cache. However, above method still needs to statically divide the unified memory into separate memories before launching a kernel in a coarse-grained way, which cannot make full use of the precious on-chip memories. Furthermore, *EXPARS* can be easily extended to spill registers to both scratchpad memory and L1 cache. Although Gebhart et al. [7] propose an unified on-chip memory structure that the capacity of register file, scratchpad memory, and L1 cache can be partitioned at runtime according to the requirement of applications in a fine-grained way, there are still two shortcomings. First, the unified structure lacks flexibility; register file is one of the main contributors to GPU energy consumption and various power saving technologies [11, 14, 23, 32–34] are proposed for register file to save energy, which can be hard to apply to the unified structure due to the different access characteristics between register file and L1 cache. Second, the unified structure increases bank conflicts between register file, scratchpad memory and L1 cache; they use software-managed hierarchical register file [6] to reduce the required bandwidth to the main register file, however, that technology focuses on energy efficiency and may lead to resource underutilization and suboptimal performance [29, 35]. Jing et al. [12] introduce an integrated architecture that enables the register file to support the function of cache, which also has above weaknesses.

GPU warp scheduling: GPU warp scheduling is a hot research point in recent years [19, 20, 31, 36, 43]. Lee et al. [20] first propose a profiling algorithm to find the critical warps and then schedule these critical warps more frequently than others. Yu et al. [43] analyze the pipeline efficiency and propose a stall-aware warp scheduler to determine the optimal number of CTAs at runtime. Lee et al. [19] propose an alternative scheduler based on GTO, which is the closest to ours among the above works, to dynamically determine the maximum number of CTAs assigned to each SM. Others [31, 36] propose schedulers mainly targeting on data locality. In this article, we propose a TL-based scheduler whose maximum number of active warps is determined at runtime.

6 CONCLUSION

In this article, we present a new resource management approach *EXPARS* for GPUs to increase TLP. When the register file is not enough for dispatching one more CTA, we expand the register file to scratchpad memory in a vertical approach so that more CTAs can be dispatched to SMs. In addition, we propose a lazy two-level warp scheduler to mitigate the resource contention due to the increased TLP. Our experiment results show that with *EXPARS* the number of CTAs dispatched to each SM increases by 1.28× on average. In addition, the register file utilization increases by 11.64%, and the scratchpad memory utilization increases by 48.20% on average. With better TLP, our approach achieves 20.01% performance improvement on average with negligible energy overhead.

ACKNOWLEDGMENTS

We thank our anonymous reviewers for their valuable feedback and suggestion. Hailong Yang is the corresponding author.

REFERENCES

[1] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'09)*. IEEE, 163–174.

[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'09)*. IEEE, 44–54.

[3] Quan Chen, Hailong Yang, Minyi Guo, Ram Srivatsa Kannan, Jason Mars, and Lingjia Tang. 2017. Prophet: Precise qos prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers. *ACM SIGARCH Comput. Architect. News* 45, 1 (2017), 17–32.

[4] Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. 2016. Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers. *ACM SIGARCH Comput. Architect. News* 44, 2 (2016), 681–696.

[5] 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 *ACM SIGARCH Computer Architecture News*, Vol. 39. ACM, 235–246.

[6] 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'11)*. ACM, 465–476.

[7] Mark Gebhart, Stephen W. Keckler, Brucek Khailany, Ronny Krashinsky, and William J. Dally. 2012. Unifying primary cache, scratch, and register file memories in a throughput processor. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'12)*. IEEE Computer Society, 96–106.

[8] Ari B. Hayes and Eddy Z. Zhang. 2014. Unified on-chip memory allocation for SIMT architecture. In *Proceedings of the 28th ACM International Conference on Supercomputing (ICS'14)*. ACM, 293–302.

[9] Vishwesh Jatala, Jayvant Anantpur, and Amey Karkare. 2016. Improving GPU performance through resource sharing. In *Proceedings of the 25th ACM International Symposium on High-Performance Distributed Computing (HPDC'16)*. ACM, 203–214.

[10] Hyeran Jeon, Gokul Subramanian Ravi, Nam Sung Kim, and Murali Annavaram. 2015. GPU register file virtualization. In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'15)*. ACM, 420–432.

[11] Naifeng Jing, Yao Shen, Yao Lu, Shrikanth Ganapathy, Zhigang Mao, Minyi Guo, Ramon Canal, and Xiaoyao Liang. 2013. An energy-efficient and scalable eDRAM-based register file architecture for GPGPU. In *ACM SIGARCH Computer Architecture News*, Vol. 41. ACM, 344–355.

[12] Naifeng Jing, Jianfei Wang, Fengfeng Fan, Wenkang Yu, Li Jiang, Chao Li, and Xiaoyao Liang. 2016. Cache-emulated register file: An integrated on-chip memory architecture for high performance GPGPUs. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. IEEE, 1–12.

[13] Onur Kayıran, Adwait Jog, Mahmut Taylan Kandemir, and Chita Ranjan Das. 2013. Neither more nor less: Optimizing thread-level parallelism for GPGPUs. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT'13)*. IEEE, 157–166.

[14] Onur Kayiran, Adwait Jog, Ashutosh Pattnaik, Rachata Ausavarungnirun, Xulong Tang, Mahmut T. Kandemir, Gabriel H. Loh, Onur Mutlu, and Chita R. Das. 2016. μC-States: Fine-grained GPU datapath power management. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT'16)*. IEEE, 17–30.

[15] Nvidia Kepler. 2012. NVIDIA's Next Generation CUDATM Compute Architecture: Kepler TM GK110. Retrieved from https://www.nvidia.com/content/PDF/kepler/NVIDIA-kepler-GK110-architecture-whitepaper.pdf.

[16] Farzad Khorasani, Hodjat Asghari Esfeden, Amin Farmahini-Farahani, Nuwan Jayasena, and Vivek Sarkar. 2018. RegMutex: Inter-warp GPU register time-sharing. In *Proceedings of the ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA'18)*. IEEE.

[17] John Kloosterman, Jonathan Beaumont, D. Anoushe Jamshidi, Jonathan Bailey, Trevor Mudge, and Scott Mahlke. 2017. Regless: Just-in-time operand staging for GPUs. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'17)*. ACM, 151–164.

[18] Jaekyu Lee and Hyesoon Kim. 2012. TAP: A TLP-aware cache management policy for a CPU-GPU heterogeneous architecture. In *Proceedings of the IEEE 18th International Symposium on*. IEEE, 91–102.

[19] Minseok Lee, Seokwoo Song, Joosik Moon, John Kim, Woong Seo, Yeongon Cho, and Soojung Ryu. 2014. Improving GPGPU resource utilization through alternative thread block scheduling. In *Proceedings of the 20th IEEE International Symposium on High Performance Computer Architecture (HPCA'14)*. IEEE, 260–271.

[20] Shin-Ying Lee and Carole-Jean Wu. 2014. CAWS: Criticality-aware warp scheduling for GPGPU workloads. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques (PACT'14)*. ACM, 175–186.

[21] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. 2013. GPUWattch: Enabling energy optimizations in GPGPUs. In *ACM SIGARCH Computer Architecture News*, Vol. 41. ACM, 487–498.

[22] Zhen Lin, Michael Mantor, and Huiyang Zhou. 2018. GPU performance vs. thread-level parallelism: Scalability analysis and a novel way to improve TLP. *ACM Trans. Architect. Code Optim.* 15, 1 (2018), 15.

[23] Majid Namaki-Shoushtari, Abbas Rahimi, Nikil Dutt, Puneet Gupta, and Rajesh K. Gupta. 2013. ARGO: Aging-aware GPGPU register file allocation. In *Proceedings of the 9th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. IEEE Press, 30.

[24] Nvidia. 2009. Nvidia's next generation cuda compute architecture: Fermi. Retrieved from https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[25] Nvidia. 2011. CUDA C/C++ SDK code samples. Retrieved from https://developer.nvidia.com/cuda-toolkit-40.

[26] Nvidia. 2014. Nvidia GeForce GTX 980 whitepaper. Retrieved from https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF.

[27] Minsoo Rhu and Mattan Erez. 2013. Maximizing SIMD resource utilization in GPGPUs with SIMD lane permutation. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*. ACM, 356–367.

[28] Minsoo Rhu and Mattan Erez. 2013. The dual-path execution model for efficient GPU control flow. In *Proceedings of the IEEE 19th International Symposium on High Performance Computer Architecture (HPCA'13)*. IEEE, 591–602.

[29] 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'18)*. ACM, 489–502.

[30] John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-Mei W. Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center Reliable High-perform. Comput.* 127 (2012).

[31] Abdulaziz Tabbakh, Murali Annavaram, and Xuehai Qian. 2017. Power efficient sharing-aware GPU data management. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'17)*. IEEE, 698–707.

[32] Jingweijia Tan and Xin Fu. 2015. Mitigating the susceptibility of GPGPUs register file to process variations. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'15)*. IEEE, 969–978.

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

[34] Jingweijia Tan, Shuaiwen Leon Song, Kaige Yan, Xin Fu, Andres Marquez, and Darren Kerbyson. 2016. Combating the reliability challenge of GPU register file at low supply voltage. In *Proceedings of the International Conference on Parallel Architectures and Compilation*. ACM, 3–15.

[35] Nandita Vijaykumar, Kevin Hsieh, Gennady Pekhimenko, Samira Khan, Ashish Shrestha, Saugata Ghose, Adwait Jog, Phillip B. Gibbons, and Onur Mutlu. 2016. Zorua: A holistic approach to resource virtualization in GPUs. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. IEEE, 1–14.

[36] Jin Wang, Norm Rubin, Albert Sidelnik, and Sudhakar Yalamanchili. 2016. Laperm: Locality aware scheduler for dynamic parallelism on gpus. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA'16)*. IEEE, 583–595.

[37] Steven J. E. Wilton and Norman P. Jouppi. 1996. CACTI: An enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits* 31, 5 (1996), 677–688.

[38] Ping Xiang, Yi Yang, and Huiyang Zhou. 2014. Warp-level divergence in GPUs: Characterization, impact, and mitigation. In *Proceedings of the 20th IEEE International Symposium on High Performance Computer Architecture (HPCA'14)*. IEEE, 284–295.

[39] Xiaolong Xie, Yun Liang, Xiuhong Li, Yudong Wu, Guangyu Sun, Tao Wang, and Dongrui Fan. 2015. Enabling coordinated register allocation and thread-level parallelism optimization for GPUs. In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'15)*. ACM, 395–406.

[40] Yi Yang, Ping Xiang, Mike Mantor, Norm Rubin, and Huiyang Zhou. 2012. Shared memory multiplexing: A novel way to improve GPGPU throughput. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT'12)*. IEEE, 283–292.

[41] Myung Kuk Yoon, Keunsoo Kim, Sangpil Lee, Won Woo Ro, and Murali Annavaram. 2016. Virtual thread: Maximizing thread-level parallelism beyond GPU scheduling limit. In *Proceedings of the 43rd ACM/IEEE International Symposium on Computer Architecture (ISCA'16)*. IEEE, 609–621.

[42] Chao Yu, Yuebin Bai, Hailong Yang, Kun Cheng, Yuhao Gu, Zhongzhi Luan, and Depei Qian. 2018. SMGuard: A flexible and fine-grained resource management framework for GPUs. *IEEE Trans. Parallel Distrib. Syst.* (2018). DOI: https://doi.org/10.1109/TPDS.2018.2848621.

[43] Yulong Yu, Weijun Xiao, Xubin He, He Guo, Yuxin Wang, and Xin Chen. 2015. A stall-aware warp scheduling for dynamically optimizing thread-level parallelism in GPGPUs. In *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS'15)*. ACM, 15–24.

Received May 2018; revised August 2018; accepted September 2018