Liang Y, Wang S. Performance-centric optimization for racetrack memory based register file on GPUs. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 31(1): 36–49 Jan. 2016. DOI 10.1007/s11390-016-1610-1

Performance-Centric Optimization for Racetrack Memory Based Register File on GPUs

Yun Liang *, *Member, CCF, ACM, IEEE*, and Shuo Wang

Center for Energy-Efficient Computing and Applications (CECA), School of Electrical Engineering and Computer Sciences, Peking University, Beijing 100871, China

E-mail: {ericlyun, shvowang}@pku.edu.cn

Received September 9, 2015; revised December 3, 2015.

Abstract The key to high performance for GPU architecture lies in its massive threading capability to drive a large number of cores and enable execution overlapping among threads. However, in reality, the number of threads that can simultaneously execute is often limited by the size of the register file on GPUs. The traditional SRAM-based register file takes up so large amount of chip area that it cannot scale to meet the increasing demand of GPU applications. Racetrack memory (RM) is a promising technology for designing large capacity register file on GPUs due to its high data storage density. However, without careful deployment of RM-based register file, the lengthy shift operations of RM may hurt the performance. In this paper, we explore RM for designing high-performance register file for GPU architecture. High storage density RM helps to improve the thread level parallelism (TLP), but if the bits of the registers are not aligned to the ports, shift operations are required to move the bits to the access ports before they are accessed, and thus the read/write operations are delayed. We develop an optimization framework for RM-based register file on GPUs, which employs three different optimization techniques at the application, compilation, and architecture level, respectively. More clearly, we optimize the TLP at the application level, design a register mapping algorithm at the compilation level, and design a preshifting mechanism at the architecture level. Collectively, these optimizations help to determine the TLP without causing cache and register file resource contention and reduce the shift operation overhead. Experimental results using a variety of representative workloads demonstrate that our optimization framework achieves up to 29% (21% on average) performance improvement.

Keywords register file, racetrack memory, GPU

1 Introduction

Modern GPUs employ a large number of simple, in-order cores, delivering several TeraFLOPs peak performance. Traditionally, GPUs are mainly used for supercomputers. More recently, GPUs have penetrated mobile embedded system markets. The system-on-chip (SoC) that integrates GPUs with CPUs, memory controllers, and other application-specific accelerators are available for mobile and embedded devices. The major SoCs with integrated GPUs available in the market include NVIDIA Tegra series with low power GPU, Qualcomm's Snapdragon series with Adreno GPU, and Samsung's Exynos series with ARM Mali GPU.

The key to high performance of GPU architecture lies in the massive threading to enable fast context switch between threads and hide the latency of function unit and memory access. This massive threading design requires large on-chip storage support[1-7]. The majority of on-chip storage area in modern GPUs is allocated for register file. For example, on NVIDIA Fermi (e.g., GTX480), each streaming multiprocessor (SM) contains 128 KB register file, which is much larger than the 16 KB L1 cache and 48 KB shared memory.

The hardware resources on GPUs include 1) registers, 2) shared memory, 3) and threads and thread blocks. A GPU kernel will launch as many threads

Regular Paper

Special Section on Computer Architecture and Systems with Emerging Technologies

This work was supported by the National Natural Science Foundation of China under Grant No. 61300005.

A preliminary version of the paper was accepted by the 21st Asia and South Pacific Design Automation Conference (ASP-DAC 2016).

*Corresponding Author

concurrently as possible until one or more dimensions of resource are exhausted. In this paper, we define thread level parallelism (TLP) as the number of thread blocks that can execute simultaneously on GPU. Though GPUs are featured with large register file, in reality, we find that the TLP is often limited by the capacity of register file. Table 1 gives the characteristics of some representative applications from Parboil and Rodinia benchmark suites on a Fermi-like architecture. The setting of the GPU architecture is shown in Table 2. The maximum number of threads that can simultaneously execute on this architecture is 1 536 threads per SM. However, for these applications, there exists a big gap between the achieved TLP and the maximum TLP. For example, one thread block of application *hotspot* requires 15 360 registers. Given 32 768 registers budget (Table 2), only two thread blocks can run simultaneously. This leads to a very low occupancy as $\frac{2\times256}{1\,536} = 33\%$, where the occupancy is defined as the ratio of simultaneously active threads to the maximum number of threads supported on one SM.

Table 1. Kernel Description

Application	Source	Block Size	Number of Registers per Thread	TLP	Occupancy (%)	Register Utilization (%)
hotspot	Rodinia	256	60	2	33	93.75
b+tree	Rodinia	256	30	4	67	93.75
histo_final	Parboil	512	41	1	33	64.06
histo_main	Parboil	768	22	1	50	51.56
mri-gridding	Parboil	256	40	3	50	93.75

Table 2. GPGPU-Sim Configuration

Number of compute units (SM)	15
SM configuration	32 cores, 700 MHz
Resources per SM	Max 1 536 threads, Max 8 thread blocks, 48 KB shared memory, 128 KB 16-bank register file (32 768 registers)
Scheduler	2 warp schedulers per SM, round-robin policy
L1 data cache	16/32 KB, 4-way associativity, 128 B block, LRU replacement policy, 32 MSHR entries
L2 unified cache	768 KB size

To deal with the increasingly complex GPU applications, new register file design with high capacity is urgently required. Designing the register file using high storage density emerging memory for GPUs is a promising solution[8-9]. Recently, racetrack memory (RM) has attracted great attention of researchers because of its ultra-high storage density. By integrating multiple bits (domains) in a tape-like nanowire[10], racetrack memory has shown about 28x higher density compared with SRAM (static random access memory)[9]. Recent study has also enabled racetrack memory for GPU register file design[8]. However, prior work primarily focuses on optimizing energy efficiency, leading to very small performance improvement[8].

In this paper, we explore RM for designing high-performance register file for GPU architecture. High storage density RM helps to enlarge the register file capacity. This allows the GPU applications to run more threads in parallel. However, RM-based design presents a new challenge in the form of shifting overhead. More clearly, for RM, one or several access ports are uniformly distributed along the nanowire and shared by all the domains. When the domains aligned to the ports are accessed, bits in them can be read immediately. However, to access other bits on the nanowire, the shift operations are required to move those bits to the nearest access ports. Obviously, a shift operation induces extra timing and energy overhead.

To mitigate the shift operation overhead problem, we develop an optimization framework, which employs three different optimization techniques at the application, compilation, and architecture level, respectively. High storage density RM allows high TLP. However, high TLP may cause resource contention in cache and register file. Thus, at the application level, we propose to adjust the TLP for better performance through profiling. At the compilation level, we design a compile-time register mapping algorithm, which attempts to put the registers that are used frequently together close in the register file to reduce the shift operation overhead. Finally, at the architecture level, we design a preshifting mechanism to preshift the ports for the idle banks to further reduce the shift operation overhead.

The key contributions of this paper are as follows.

• *Framework*. We develop a performance-centric optimization framework for RM-based register file on GPUs.

• *Techniques*. Three optimization techniques at the application, compilation, and architecture level are developed.

• *Evaluation.* Experiments on a variety of applications show that compared with SRAM design, our RM-based register file design improves performance by 21% on average.

2 Background

In this section, we first review the GPU architecture using NIVIDIA Fermi architecture as an example. We then introduce the basics of racetrack memory. Finally, GPU register file designed based on racetrack memory is presented.

2.1 GPU Architecture

Modern GPUs achieve high throughput thanks to the massive threading feature. When an application is launched on a GPU, hundreds or thousands of threads can execute simultaneously. To support such a high thread level parallelism, GPU has multiple streaming multiprocessors (SMs) working in parallel. Each SM contains an instruction cache, warp scheduler, L1 data cache, register file, shared memory, and 32 small cores, called streaming processors (SPs), as shown in Fig.1. Within an SM, threads are often grouped as a warp, which usually contains 32 threads. At any given clock cycle, a warp that is ready for execution, is selected and then issued to SPs by the warp scheduler. The SPs in an SM work in the single instruction multiple data (SIMD) fashion and they share the on-chip memory resources including register file, shared memory and L1 data cache on an SM.

Warp is the thread scheduling unit. If a warp is stalled due to the long latency operation (e.g., memory operation), then the warp scheduler will switch the execution to other ready warps to hide the overhead. In order to fully hide the stall overhead caused by the long latency operations, GPU applications tend to launch as many threads as possible. However, the capacity of register file is limited, which constrains the TLP on the GPU. Furthermore, due to the chip area, power and energy constraints, it is very difficult to enlarge the register file using the current SRAM-based technology.

2.2 Basics of Racetrack Memory

Racetrack memory is an emerging non-volatile memory, which is also known as spintronic domain wall memory. Recently, it is becoming increasingly popular due to its high storage density[10-12] and low energy consumption. Compared with SRAM, racetrack

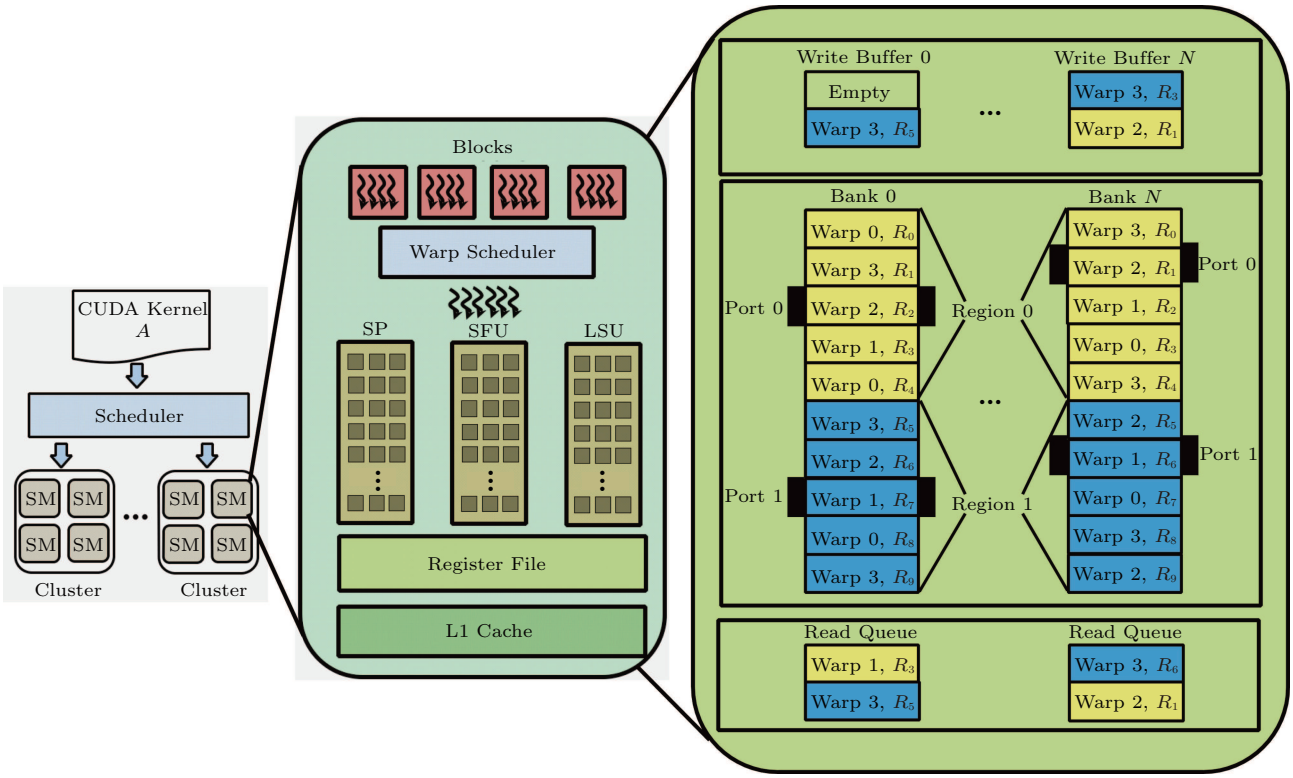


Fig.1. Architecture of GPU with an RM-based register file.

memory (RM) could achieve about 28x storage density while keeping comparable access speed[9]. Moreover, read and write operation could achieve about 2x and 5x energy reduction respectively[11]. In the following, we will present the details of the basic operations of racetrack memory.

Read and Write Operations. Analogous to accessing the head of hard disk, one access head in RM can access multiple bits. However, RM needs to shift the bit to the head if the bit is not aligned to the port. RM stores multiple bits in tape-like magnetic nanowires called cells. A cell is made of a nanowire (holding successive domains) to save bits and several access ports to access them, as shown in Fig.2. The "white bricks" represent the domains and those separations are domain walls used to isolate successive domains. Each access head has two transistors T_1 and T_2, as outlined by the red dash boxes. T_1 is used to read. The magnetization direction of each domain represents the value of the stored bit. If the direction is anti-parallel to the reference domain (grey bricks) in the access port, the value is "1"; else it will be "0". T_2 is used to write bits into a domain. When T_2 is on, a crosswise current is applied on the domain, and the required bit is shifted in just as a shift operation.

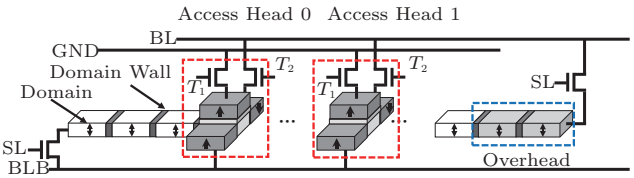


Fig.2. Physical structure of a racetrack memory cell.

Shift Operations. Shift operation is an unique operation of racetrack memory. The shift operation is based on a phenomenon called spin-momentum transfer caused by spin current. The shift current provided by shift control transistor (SL) drives all the domains in a racetrack cell left and right. Note that several overhead domains are physically assigned at either end of the cell, in order to save valid domains with stored bits when they are shifted out.

2.3 GPU Register File with Racetrack Memory

Multi-Bank RM-Based Register File. In order to support multi-read and multi-write operations, multi-bank SRAM register file is widely adopted in high-throughput processors. Compared with single-bank multi-ported SRAM register file, it achieves higher performance, lower power, and less area[9,12-13]. For instance, the NIVIDA Fermi architecture employs a 16-bank SRAM register file, where up to four read or write operations could be accomplished at the same time for register file. In this paper, we design RM-based register file following the multi-bank architecture.

Fig.1 shows the architecture of RM-based register file. It is 16-bank RM-based register file. For each bank, it has a read request queue and a write buffer. The read request queue is used to buffer read requests. The write buffer is used to resolve the conflict between the delayed write operation in RM and the immediate write-back operation in GPU pipeline's last stage. All the read and the write requests will be arbitrated by the request arbitrator to avoid bank conflicts and determine the sequence for the requests. At any given clock cycle, at most four read or write requests could be served at the same time, which means up to four banks can be accessed simultaneously.

Performance of RM-Based Register File. For SRAM-based register file, each register could be randomly accessed in any given clock cycle, while for the RM-based register file, the lengthy shift operations of RM would increase the register access latency. This latency could be up to tens of cycles according to the distance between the target domains and the corresponding access ports, and this could degrade the performance of GPU with RM-based register file. Therefore, we propose three optimization techniques in Section 4 to mitigate this problem.

3 Motivation

RM has high storage density, which means under the same chip area budget, larger register file capacity could be achieved. Moreover, larger register file capacity could allow more threads of GPU, unleashing performance improvement by larger TLP. Table 1 shows the maximum TLP of five benchmarks under different sizes of register file. In order to quantify the potential performance improvement by using RM-based register file, we compare the performance of GPU in five applications under the same chip area budget① for SRAM and ideal RM-based register file②. As shown in Fig.3, by using RM-based register file, we could potentially

①The chip area budget is set according to the area of a 16-bank 128 KB SRAM register file as shown in Table 2.

②The ideal RM-based register file is RM-based register file when the shift operation overhead is zero.

achieve up to 29% (on average 22%) performance improvement. The improvement is attributed to larger TLP enabled by larger register file capacity. For example, the TLP of application mri-gridding is increased from 3 to 6 by doubling register file capacity. Note that this improvement is the ideal performance improvement by assuming the same access delay for SRAM and RM-based register file and the shift operation overhead of RM is zero. In this paper, an optimization framework for RM-based register file on GPUs is proposed targeting at increasing performance. We will demonstrate that by employing our proposed framework to optimize RM-based register file, the achieved performance improvement is close to the ideal case.

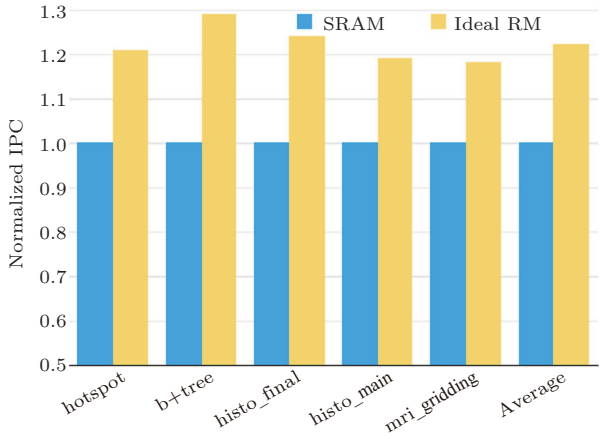


Fig.3. GPU performance in different applications.

4 Optimization Framework

A preliminary version of this paper was reported in [14]. In this paper, we propose an optimization framework consisting of three performance-centric optimization techniques at the application, compilation, and architecture level.

4.1 Framework Overview

We develop an RM-based register file optimization framework on GPU as shown in Fig.4. It employs three performance-centric optimization techniques at the application, compilation, and architecture level. At the application level, we determine the TLP that does not cause contention in neither register file nor cache through profiling. At the compilation level, we design a register mapping algorithm to reduce the number of shift operations by analyzing the register access trace. Finally, at the architecture level, we design a preshifting mechanism for the register file request arbitrator to preshift the ports for the idle banks. In the following, we will describe the details for each technique.

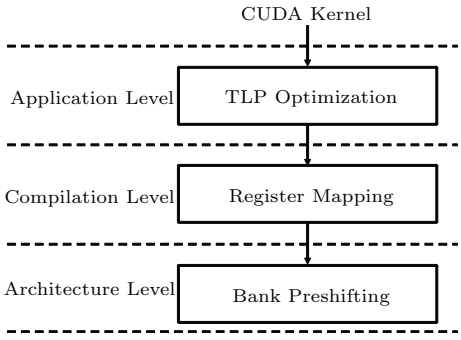


Fig.4. Optimization framework.

4.2 TLP Optimization

By using RM-based register file, we can support high TLP on the GPU. However, the previous study[15] shows that due to the contention in caches, the maximum TLP is not always the optimal choice for performance. Furthermore, using the RM-based register file design, high TLP could lead to long shift distance, which results in worse performance. Therefore, in this subsection, we adjust TLP by modelling the resource contention and shift operation overhead.

Cache Contention. RM-based register file allows more threads blocks to be executed on each SM, which increases the TLP of GPU. However, higher TLP may cause cache contention and further degrades GPU performance. To illustrate the performance degradation problem caused by cache contention, we show the performance and L1 data cache miss rate of two typical kernels under different TLP numbers in Fig.5. Note that to exclude the impact from the lengthy shift operations of RM, we set the shift operation overhead to zero. From Fig.5(a), we can see that the performance of kernel *calculate_temp* monotonically increases as the number of thread blocks increases, while the cache miss rate remains almost the same. This means that for kernel *calculate_temp*, the increased TLP will not aggravate cache contention, thus the best performance is achieved by setting TLP to be the maximum number of thread blocks per SM. As for kernel *split_sort*, however, the maximum TLP cannot lead to the best performance as shown in Fig.5(b). The best performance is achieved when the number of thread blocks per SM is set to be 5 but not 6. This is due to the aggravated cache contention caused by higher TLP, which is reflected by the increased cache miss rate. Therefore,

based on the analysis of these two kernels, we conclude that because of the cache contention overhead caused by increased TLP, the maximum TLP cannot always lead to the best GPU performance.

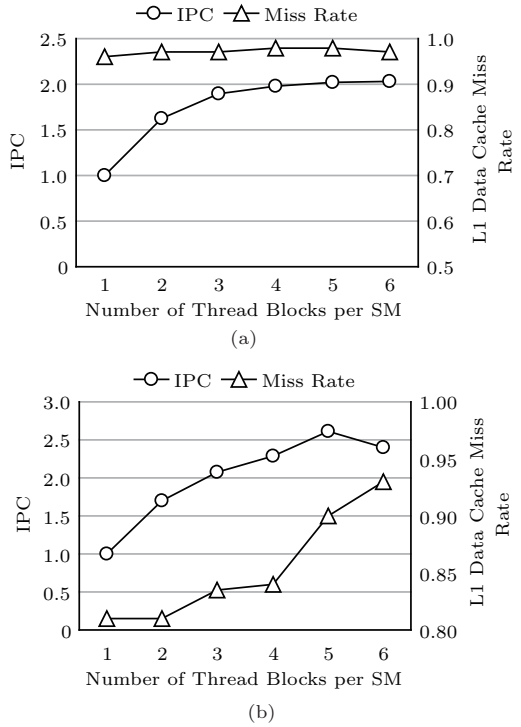


Fig.5. Performance and L1 data cache miss rate of two kernels. Both IPC and cache miss rate values are normalized according to the results when the number of thread blocks per SM is 1. (a) *calculate_temp*. (b) *split_sort*.

Shift Operation Overhead. Higher TLP will lead to more thread blocks present on GPU cores, which requires more registers allocated in register file. Due to the tape-like structure of RM, the maximum shift distance will be longer for RM. As Fig.6 shows, when the TLP is increased from 2 to 4, the number of warp registers per bank is doubled, thus making maximum shift distance increased from 4 to 8. The increased maximum shift distance enlarges the RM working space, thereby the read/write operations may be further delayed by more shift operations. These delayed read/write operations will cause performance degradation for GPU. Therefore, in order to determine the TLP to achieve optimal performance, we should also take into account of the shift operation overhead.

TLP Optimization. To optimize the TLP to achieve the best performance, we need to model both the cache contention and the shift operation overhead. Because the shift operation overhead is present in the register file while the cache contention problem exists in L1 data cache, we assume the shift operation overhead model is independent of cache contention model. Therefore, we model them separately and then combine them together to optimize TLP.

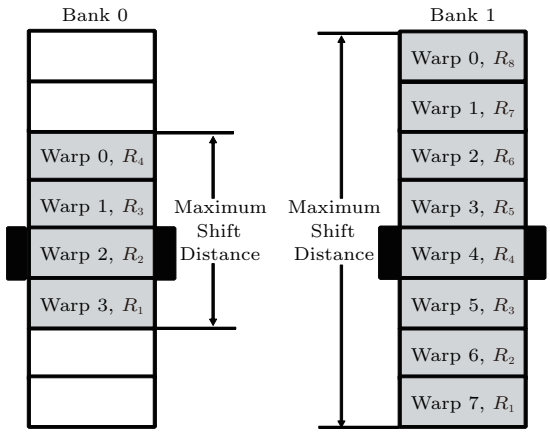


Fig.6. Shift operation overhead caused by increased TLP.

First, the cache contention is modeled by the variable $P_{\text{cache}}(N_{\text{TLP}})$, which represents the performance (IPC) under different TLPs but without shift operation overhead, and N_{TLP} is defined as the TLP value. We obtain the $P_{\text{cache}}(N_{\text{TLP}})$ value by using a profiling technique. More clearly, we profile the performance of each benchmark under different TLP values. To make the shift operation overhead to be zero, we set the number of access ports of RM to be the same as the number of data domains of racetrack, and thus no shift operation is needed for read/write operations.

Second, the shift operation overhead is modeled by the variable $\Delta P_{\text{shift}}(N_{\text{TLP}})$, which represents the amount of performance (IPC) reduction when shift operation overhead is considered, and it is calculated according to the following equation:

$$\Delta P_{\text{shift}}(N_{\text{TLP}}) = P_{\text{ideal}}(N_{\text{TLP}}) - P_{\text{RM}}(N_{\text{TLP}}). \quad (1)$$

In this equation, $P_{\text{ideal}}(N_{\text{TLP}})$ represents the performance (IPC) when both cache contention and shift operation overheads are zero. We get the value of $\Delta P_{\text{shift}}(N_{\text{TLP}})$ by employing a profiling method. To be more specific, we profile the performance results of $P_{\text{ideal}}(N_{\text{TLP}})$ and $P_{\text{RM}}(N_{\text{TLP}})$ under different TLP numbers. And then we calculate $\Delta P_{\text{shift}}(N_{\text{TLP}})$ according to (1). Note that, to exclude the cache contention overhead during profiling, we set the L1 data cache to be hit at all the time.

Last, the performance of a benchmark considering both cache contention and shift operation overhead is calculated as

$$P(N_{\text{TLP}}) = P_{\text{cache}}(N_{\text{TLP}}) - \Delta P_{\text{shift}}(N_{\text{TLP}}). \quad (2)$$

To optimize TLP to achieve the best performance, we calculate each $P(N_{\text{TLP}})$ according to (2), and then find the TLP value which leads to the best performance. Overall, only a few times of profiling are needed for each benchmark③.

4.3 Register Mapping Optimization

In this subsection, we design a compile-time register mapping algorithm, which first analyzes the register access trace and then attempts to put the registers that are used frequently together close in the register file to reduce the shift operation overhead. Fig.7(a) shows a snippet of the register file access trace of a bank shown in Fig.7(b). There are totally five registers (R_0∼R_4) accessed in the bank and the trace contains six accesses to the registers. If we map the registers to the bank simply in an ascending order of register index as shown in Fig.7(b), the racetracks need 12 shift operations to access the registers for this trace. The optimal mapping of register is shown in Fig.7(c). By exchanging the positions of registers R_0 and R_4, the shift operations can be reduced to 6. The warps in GPUs access the register file almost every clock cycle. Hence, we need to carefully map the registers to the physical address in racetrack register file to reduce the shift operations and ensure high throughput.

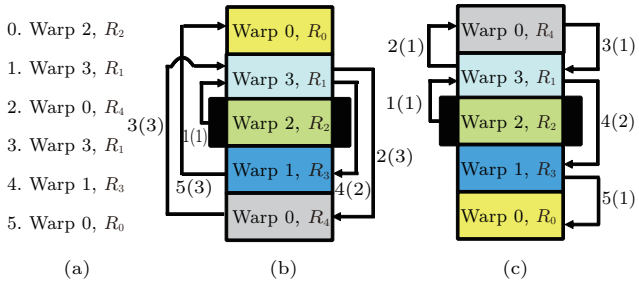


Fig.7. Comparison of different register mappings. (a) Register access trace. (b) Direct register mapping. (c) Optimal register mapping. For (b) and (c), each arrow is associated with $m(n)$, where m represents the m-th register access in the trace, and n represents the number of shift operations.

4.3.1 Problem Formulation

In the following, we formulate the register mapping problem. We observe that the register access traces of different banks are disjoint, and thus different banks can be modelled independently. Hence, next we will discuss the register mapping for one bank. The same techniques can be repeated for different banks.

Let $\mathcal{T}$ be the register access trace (sequence of register access) generated by executing the GPU application on the target architecture. We define a *move* from register R_i to R_j if R_j is accessed immediately after the access of R_i. We use C_{ij} to represent the number of *move* from register R_i to R_j in the trace, where $i, j = 0, 1, 2, ..., N_r - 1$, and N_r is the number of registers in a bank. C_{ij} can be easily derived by traversing the trace.

We define a mapping function $m(R_i)$ that maps register R_i to a physical address in the register file as follows:

$$m(R_i) = p(R_i) \times \frac{N_d}{N_p} + o(R_i),$$

where $0 < p(R_i) < N_p$ and $0 \leqslant o(R_i) < \frac{N_d}{N_p}$. In other words, $p(R_i)$ determines which port's region R_i is mapped to and $o(R_i)$ determines the offset of R_i in the region. Each physical entry can only be allocated for one register. Thus, $\forall 0 \leqslant i, j \leqslant N_r - 1, m(R_i) \neq m(R_j)$.

Then, we define the number of shift operations required for trace $\mathcal{T}$ as follows:

$$S = \sum_{0 \leqslant i,j \leqslant N_r - 1} C_{ij} \times d(m(R_i), m(R_j)), \quad (3)$$

where $d(m(R_i), m(R_j))$ represents the shift operation needed from the physical address of R_i to R_j within a bank. The function $d(m(R_i), m(R_j))$ is defined as follows:

$$d(m(R_i), m(R_j)) = |o(R_i) - o(R_j)|. \quad (4)$$

Problem 1 (Shift Operation Minimization). *Given the register access trace $\mathcal{T}$, find a mapping of the registers to the physical address in the bank (e.g., $m(R_i)$) such that S is minimized.*

We solve Problem 1 in two phases. In the first phase, we develop a register grouping algorithm that partitions the registers into groups. The size of each group is N_p (the number of ports). In the second phase, we develop a register arrangement algorithm that optimizes

③The number of recurrence is bounded by the maximum TLP supported on the GPU. The maximum TLP is a hardware limit for any generation of GPUs. For our platform, the maximum TLP is 8, which means at most 8 times of occurrence are needed for each benchmark.

the arrangement of registers within a bank. The first phase determines the $p(R_i)$ part and the second phase determines the $o(R_i)$ part in $m(R_i)$ for each register, respectively.

4.3.2 Register Grouping Algorithm

We split the objective function (3) into two parts based on (4) as follows:

$$\begin{aligned} S &= \sum_{0\leqslant i,j\leqslant N_r-1, o(R_i)\neq o(R_j)} C_{ij} \times d(m(R_i), m(R_j)) + \\ &\quad \sum_{0\leqslant i,j\leqslant N_r-1, o(R_i)=o(R_j)} C_{ij} \times d(m(R_i), m(R_j)) \\ &= \sum_{0\leqslant i,j\leqslant N_r-1, o(R_i)\neq o(R_j)} C_{ij} \times |o(R_i) - o(R_j)| + \\ &\quad \sum_{0\leqslant i,j\leqslant N_r-1, o(R_i)=o(R_j)} C_{ij} \times |o(R_i) - o(R_j)| \\ &= \sum_{0\leqslant i,j\leqslant N_r-1, o(R_i)\neq o(R_j)} C_{ij} \times |o(R_i) - o(R_j)|. \end{aligned} \tag{5}$$

That is, the moves between two registers that share the same offset to ports do not require shift operations. This is because these registers can be accessed simultaneously via multiple ports in racetrack-based register file without extra shift overhead.

Furthermore, given the register access trace $\mathcal{T}$, the total number of moves is a constant. We define it as follows:

$$\begin{aligned} Q &= \sum_{0\leqslant i,j\leqslant N_r-1} C_{ij} \\ &= \sum_{0\leqslant i,j\leqslant N_r-1, o(R_i)\neq o(R_j)} C_{i,j} + \\ &\quad \sum_{0\leqslant i,j\leqslant N_r-1, o(R_i)=o(R_j)} C_{i,j}. \end{aligned} \tag{6}$$

From (5) and (6), we find that $\sum_{o(R_i)=o(R_j)} C_{i,j}$ is negative correlated with S. Thus, we can minimize S by maximizing $\sum_{o(R_i)=o(R_j)} C_{i,j}$. The intuition behind this is that if there are frequent moves between R_i and R_j (e.g., $C_{i,j}$ is high), then we should align R_i and R_j to the same offset along two ports so that they can be accessed simultaneously without shifting overhead. In our racetrack-based register file design, each bank is associated with N_p ports. Thus, we need to partition the registers into groups of size N_p. The registers in each group g have a high number of moves (e.g., $\sum_{R_i,R_j\in g} C_{i,j}$) between them.

We build an undirected graph $G = (V, E)$, where $v_i \in V$ represents register R_i, $|V| = N_r$. The edges are weighted using function W, where W is defined as follows,

$$W(e(v_i, v_j)) = C_{i,j} + C_{j,i}.$$

Problem 2 (Subgraph Weight Maximization). *Given the graph $G = (V, E)$, find a subgraph $G' = (V', E')$ where $V' \subseteq V$, $E' \subseteq E$, $|V'| = N_p$, such that $\sum_{\forall e\in G'} W(e)$ is maximized.*

The registers in subgraph G' form a group and we will distribute the N_p registers in G' across the N_p ports with the same offset.

Algorithm 1 describes the details of our register grouping algorithm. It partitions the groups into $\lceil \frac{N_r}{N_p} \rceil$ groups and each group has N_p registers. Algorithm 1 repeatedly calls function $FindMaxSubgraph$ to form a group from G. Function $FindMaxSubgraph$ is a heuristic to Problem 2. It finds the nodes in a group iteratively. It first finds the edge with the maximal weight. Then, in each iteration, it will complement the existing subgraph G' with either one node (lines 21∼25) or two nodes (lines 26∼30) depending on which results in larger weight (lines 31∼36). Each time when $FindMaxSubgraph$ is called, it returns a subgroup G' with size N_p. The registers in G' are distributed across the N_p ports with the same offset. We assign the port region ($P(R_i)$) for each register in G' (lines 7∼9, 15∼17). As a byproduct of this process, we assign group identifier for each group based on the sequence of the formed group. Fig.8 illustrates Algorithm 1 using an

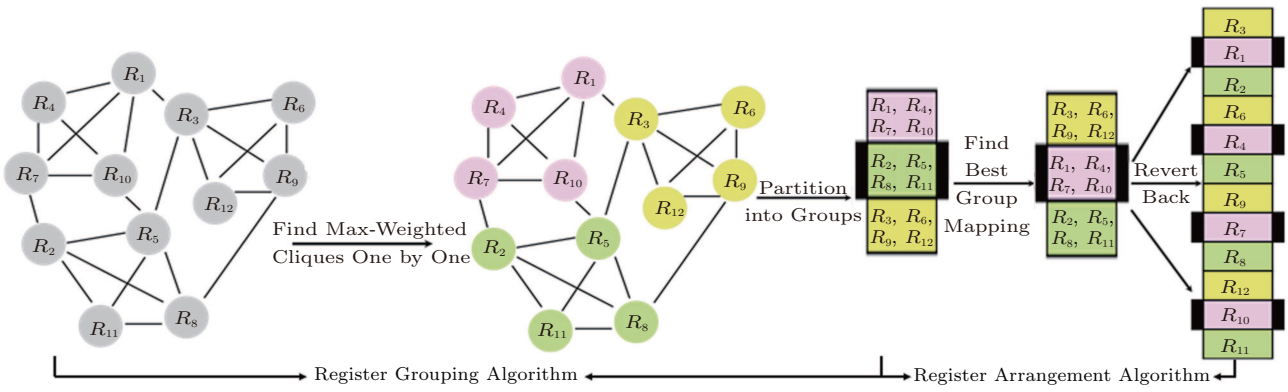


Fig.8. Illustration of register mapping algorithm.

example. In this example, $N_p = 4$. We partition the registers into three groups with the group size of 4.

Algorithm 1. Register Grouping Algorithm

```
   input  : G = (V, E)
   output: m(R_i)
1  for t ← 0 to ⌈N_r/N_p⌉ − 1 do
2      port_num ← 0;
3      group_id = 0;
4      if |V| ⩾ N_p then
5          G'(V', E') ← FindMaxSubgraph(G, N_p);
6          foreach v_i ∈ V' do
7              o(R_i) ← offset;
8              p(R_i) ← port_num;
9              PortNum ← port_num + 1;
10         end
11         Delete G' from G
12     end
13     group_id ← group_id + 1;
14     else
15         G'(V', E') ← FindMaxSubgraph(G', n);
16         foreach v_i ∈ V do
17             o(R_i) ← group_id;
18             p(R_i) ← port_num;
19             port_num ← port_num + 1;
20         end
21     end
22 end
23 FindMaxSubgraph(G = (V, E), Size){ V' ← ∅, E' ← ∅;
24 while |V'| ⩽ Size do
25     w_d ← 0, w_e ← 0;
26     foreach v_t ∈ V do
27         w ← Σ_{v_i,v_j∈(V_s∪v_t),i≠j} C_{i,j};
28         if w_d ⩽ w then
29             w_d ← w;
30             v_d ← v_t;
31         end
32     end
33     foreach e(v_m, v_n) ∈ V do
34         w ← Σ_{v_i,v_j∈(V_s∪(v_m,v_n)),i≠j} C_{i,j};
35         if w_e ⩽ w then
36             w_e ← w;
37             V_e ← {v_m, v_n};
38         end
39     end
40     if w_d ⩾ w_e and |V_s| ⩽ size − 1 then
41         V' ← V' ∪ v_d;
42         V ← V − v_d;
43         else if w_e > w_d and |V'| ⩽ size − 2 then
44             V' ← V' ∪ V_e;
45             V ← V − V_e;
46         end
47     end
48 end
49 E' = {e(v_i, v_j)|v_i, v_j ∈ V', v_i ≠ v_j};
50 Return G' = (V', E');
51 }
```

4.3.3 Register Arrangement Algorithm

Algorithm 1 determines the port region $p(R_i)$ for each register. In this subsection, we develop a register arrangement algorithm that determines the offset $o(R_i)$ for each register.

Given N_r registers in a bank, there exist $N_r!$ possible mappings of registers to the physical address in the register file. For example, using racetrack memory, we can increase the capacity of the register file from 128 KB to 256 KB. The racetrack-based register file is designed using 16 banks and 8 ports (N_p) per bank as this design gives the best trade-off among access latency, power and area as shown by prior studies[8-9,11-12]. Under this setting, we have $N_r = 64$. Obviously, it is intractable to enumerate all the possible mappings (64!).

The registers are partitioned into groups by Algorithm 1. More importantly, all the registers in the same group share the same offset. Therefore, we can determine the offset of the groups and assign the offset of the group for all the registers in it. By partitioning the registers into N_p groups, we reduce the size of mapping problem from N_r to $\frac{N_r}{N_p}$. In our setting, the problem size is reduced from 64! to 8!, which enables the enumeration for the best case.

Let Ω be the set of all the permutations of eight groups. Let $\sigma \in \Omega$, then $\sigma[i]$ is the offset of the i-th group. Algorithm 2 presents the details for register arrangement algorithm. It finds the best mapping that minimizes S ((3)) and returns $m(R_i)$ for each register. Fig.8 illustrates the register arrangement algorithm using an example. In this example, $N_r = 12$ and $N_p = 4$. By partitioning the registers into three groups, we can easily determine the mapping for groups through enumeration (3!) and then derive the mapping for all the registers.

Algorithm 2. Register Arrangement Algorithm

```
   input  : g(R_i), p(R_i)
   output : m(R_i)
1  S ← +∞;
2  foreach enumeration of groups σ ∈ Ω do
3      S' ← Σ_{0⩽i,j⩽N_r−1} C_{i,j} × |σ[g(R_i)] − σ[g(R_j)]|; if
       S' ⩽ S then
4          S ← S';
5          foreach register R_i do o(R_i) = σ[g(R_i)];
6      end
7  end
   // compute the final mapping function
8  foreach register R_i do m(R_i) ← p(R_i) × N_d/N_p + o(R_i);
```

4.4 Preshifting Optimization

GPU register file adopts a multi-bank structure. At any given clock cycle, only a small portion of register file banks could be accessed for read or write operations.

For instance, in NIVIDIA Fermi architecture, up to 4 out of 16 register file banks could be accessed simultaneously. In this paper, we define the banks that are being accessed as busy banks, while the banks which are not being accessed as idle banks. Fig.1 shows that each bank of register file owns a read request queue and a write buffer. The read requests and write requests to be served in each bank are all stored in the corresponding queue or buffer. Because the requests in each queue or buffer are served in a first-in first-out (FIFO) fashion, for a busy bank, the top request in the read request queue or write buffer will be poped out and then served. It is needed to note that the write requests have higher priority than the read requests, and thus the write requests will always be served before the read requests.

Because only 4 out of 16 banks of register file could be served in any given clock cycle and the next accessed register can be accurately determined by knowing the top requests of each queue or buffer, we could take advantage of the idle banks to reduce shift operation overhead by preshifting. A simple motivation example is shown in Fig.9. As the figure shows, bank 0∼bank 4 are busy banks serving the read/write requests, while the remaining idle banks are preshifting towards registers which are going to be accessed in the next. This preshifting scheme could potentially reduce the number of shift operations when the idle banks are carefully deployed.

Preshifting Strategy. With the knowledge of the top requests in each queue and buffer, we could determine the next warp register to be served in the following for each bank, which leaves us a good opportunity to preshift the idle banks in advance before they are appointed to be busy banks. Therefore, we propose an idle bank preshifting mechanism shown in Fig.10.

First, in any given clock cycle, after the four busy banks are determined by the register file arbitrator, each bank will be checked about whether it needs a shift operation. If it is a busy bank, it will be read/written or shifted depending on whether it needs shift, while if the bank is an idle bank, it will also be checked about whether it needs a shift operation. If it needs a shift operation to align the register to corresponding access port, the bank will shift one step towards access port; otherwise it will do nothing. Overall, this opportunistic preshifting policy makes the idle banks shift as much as possible to keep the ready requests to be served as early as possible, which could potentially reduce the shift operation overhead.

Hardware Overhead of Preshifting Overhead. The preshifting strategy is implemented as a control logic in register file arbitrator. This control logic only needs to know idle/busy state and the current access port position of each bank. The hardware implementation for this control logic is simple because only a few comparators and control state flip-flops are required. Then, since the preshifting logic is dedicated for one bank, the routing resource overhead is limited. Thus, the hardware overhead incurred by incorporating our proposed preshifting strategy for register arbitrator is negligible.

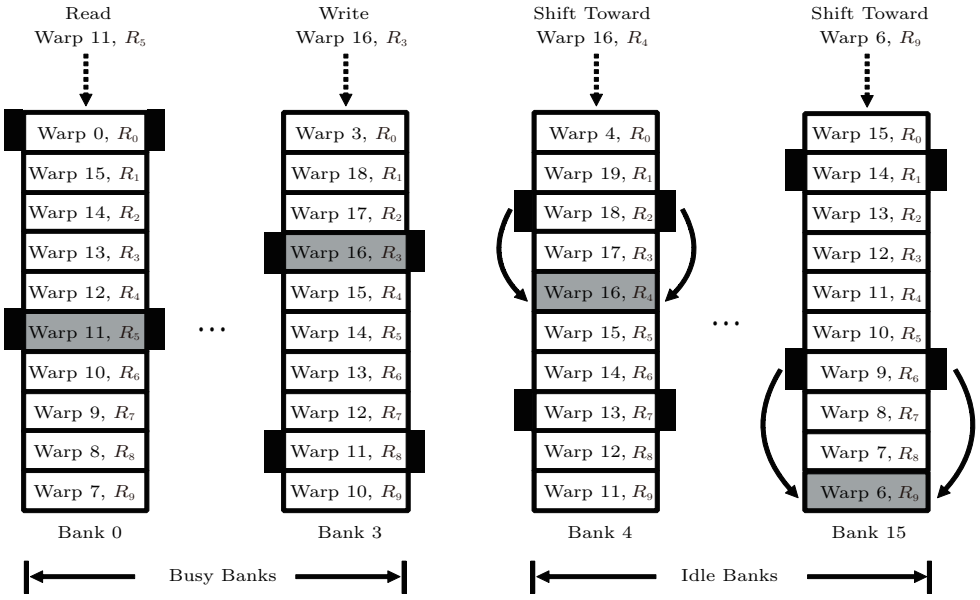


Fig.9. Example of opportunistically preshifting idle banks.

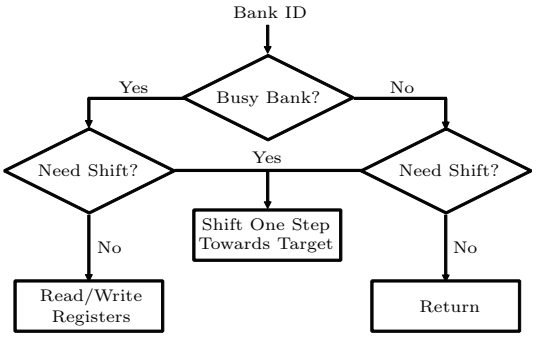


Fig.10. Flow of opportunistic preshifting idle banks.

5 Experimental Evaluation

We implement our racetrack-based register file design based on GPGPU-sim. We evaluate our technique using applications from benchmark suites Rodinia and Parboil as shown in Table 1. We extend GPGPU-sim with racetrack memory model using a circuit-level racetrack memory simulator[9]. The design parameters of racetrack-based register file are shown in Table 3. Racetrack memory incurs long write latency. To mitigate this problem, similar to prior study[8], we employ a write buffer to improve the writing efficiency. The write buffer is 2 KB and each bank has two entries as shown in Fig.1.

In the following, we perform three sets of experiments. First, we demonstrate the performance improvement of our racetrack-based register file. Second, we show the energy results. Third, we compare the area scalability of SRAM and racetrack memory.

5.1 Performance Results

Fig.11(a) shows the performance improvement of racetrack-based register file over default SRAM design. Our proposed optimization framework achieves up to 29% (21% on average) performance improvement. The reasons for the improvement are mainly two folds. First, high-density racetrack-based register file doubles the cache capacity (256 KB) compared with conventional SRAM design (128 KB). The increased register file enables the applications to execute more threads in parallel. Fig.12 depicts the GPU occupancy improvement, showing that the occupancy is increased from 46.6% to 80.0% on average. Second, the optimization techniques used in our optimization framework have greatly alleviated the shift operation overhead, leading to a promising performance improvement.

In Section 4, we propose a framework which embraces three optimization techniques, TLP optimization, register mapping, and preshifiting optimization. In order to quantify the respective portions of contribution from these proposed techniques, we break down the contributions and show the results in Fig.11(d). It shows that RM-based register file without using the proposed method contributes to 18% performance improvement, while our optimization framework contributes to 82% performance improvement. To further break down the contributions from each technique used in our framework, we show that the TLP optimization, register mapping, and preshifting optimization techniques contribute to 8%, 54%, and 20% performance improvement respectively.

Fig.11(b) shows the average number of shift operation has been reduced to 41% of the original number of shift operations by employing our optimization framework. To differentiate the contributions from the three optimization techniques, we show the contribution breakdown in Fig.11(e). As shown in the figure, we can see that preshifting idle banks, TLP optimization, and register mapping contribute to 17%, 30%, and 53% number of shift operation reduction respectively, which matches the performance improvement contribution result we have concluded from Fig.11(d).

5.2 Energy Results

Fig.11(c) shows the energy consumption of RM-based register file with and without our optimization framework, and the value of energy consumption is normalized according to the SRAM-based register file energy consumption. As shown in Fig.11(c), the energy consumption has decreased by 33%∼53% (on average 43%) by employing our proposed optimization techniques. In order to quantify how much our proposed

Table 3. SRAM, RM, and Write Buffer Operating Parameters

Register File	Capacity (KB)	Number of Banks	Read Latency (ns)	Write Latency (ns)	Shift Latency (ns)	Read Energy (pJ)	Write Energy (pJ)	Shift Energy (pJ)	Leakage (mW)
SRAM	128	16	0.31	0.31	-	218.88	57.28	-	12.31
RM	256	16	0.28	1.24	0.61	117.12	173.22	56.16	7.95
Write buffer	2	-	0.15	0.16	-	15.60	14.60	-	1.12

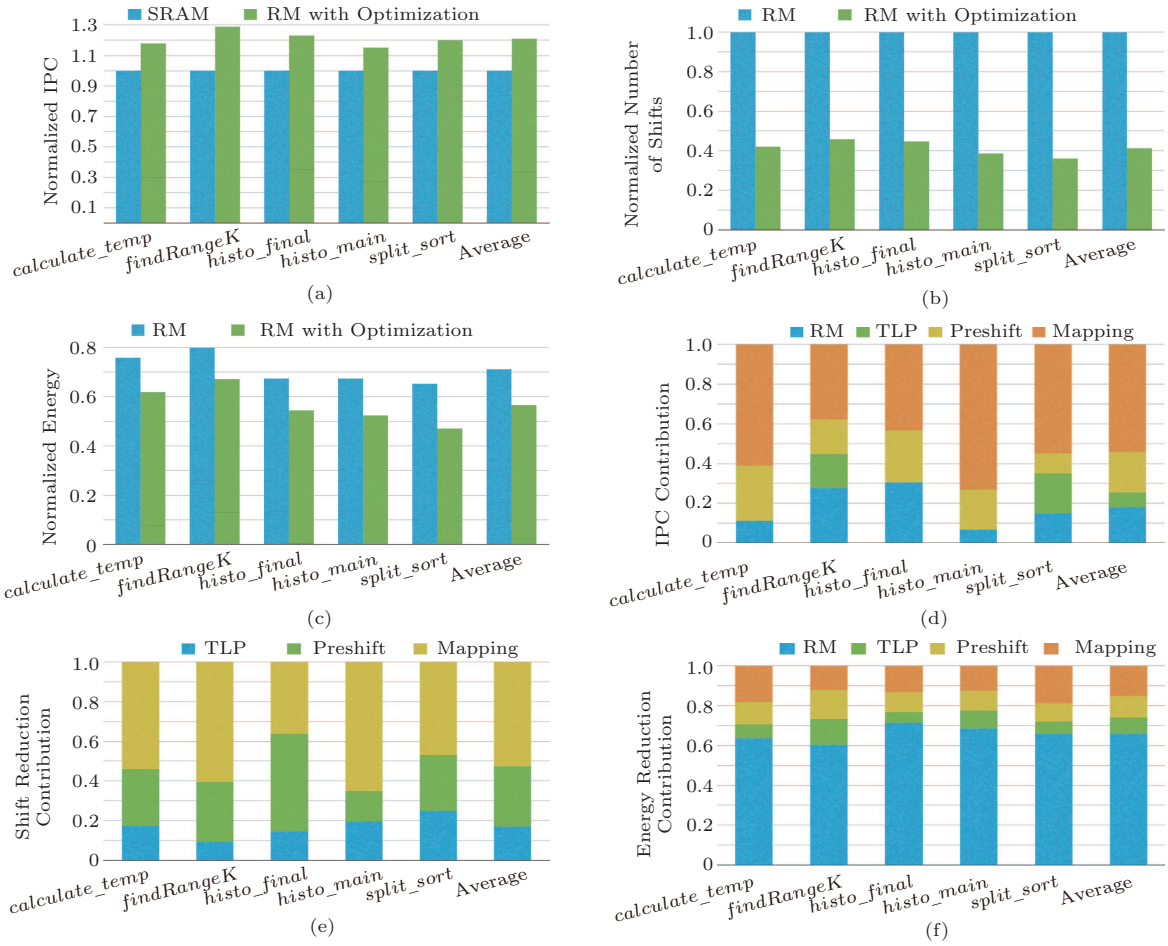


Fig.11. Analysis of performance, number of shifts, and energy in different applications. (a) Performance. (b) Shift number. (c) Energy. (d) Performance breakdown. (e) Shift number breakdown. (f) Energy breakdown.

method contributes to the reduced energy consumption, we break down the energy contribution and show the results in Fig.11(f). As we can see, the emerging RM technology is the main factor bringing energy consumption reduction, which contributes to 66% energy reduction. Moreover, 44% energy consumption reduction comes from employing our proposed optimization framework, where preshifting idle banks, TLP optimization, and register mapping contribute to 11%, 8%, and 15% energy reduction respectively.

5.3 Area Results

RM has high storage density compared with SRAM. Thus, after doubling the register file size from 128 KB to 256 KB by employing RM-based register file, the chip area of RM-based register is only 55% of the area of SRAM-based register file. This indicates that within the same chip area budget, RM-based register file could provide more register file capacity for GPU and thus higher TLP. Therefore, RM-based register file is a promising area-efficient solution for future GPU register file.

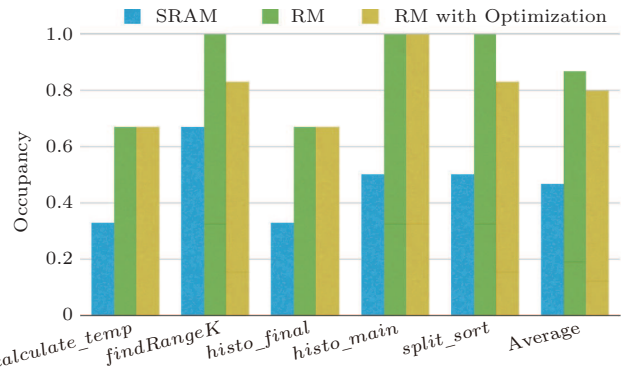


Fig.12. GPU occupancy in different applications.

6 Related Work

Recently, emerging memory technologies have attracted a lot of attention in both industry and academic area. Jog *et al.* presented a technique to improve STT-RAM based cache performance for CMP[16]. Topics about using STT-RAM to architect last level cache have been studied in [17-18]. Optimization techniques at architecture and compilation level were proposed for racetrack memory[11]. Chen *et al.* proposed a data placement strategy to minimize the shift operations for racetrack-based memory in general CPU[19]. However, GPU and CPU have distinct architecture. Thus, these techniques cannot be directly applied to GPU.

Emerging Memory Technology for GPU Cache. There are a few proposals that explore emerging memory for GPU caches by utilizing its high storage density and power efficient features. TapeCache[20], a cache designed with racetrack memory, demonstrated the benefits of racetrack-based cache design for GPUs. Venkatesan *et al.* presented a detailed racetrack memory based cache architecture for GPGPU cache hierarchies[12], and their experiment results show substantial performance and energy improvement.

Emerging Memory Technology for GPU Register File. GPUs demand a large size of register file for thread context switch. Racetrack memory is a good candidate for designing GPU register file[8,12]. Mao *et al.* presented a racetrack memory based GPGPU register file[8]. They proposed to use a hardware scheduler and write buffer to reduce energy consumption. Besides the racetrack memory, Jing *et al.* proposed an energy-efficient register file design for GPGPU based on eDRAM and also developed a compiler-assisted register allocation optimization technique[13,21]. However, all the previous work using emerging technology for GPU register file mainly focuses on optimizing area, power and energy, resulting in no or little performance improvement. In contrast, we focus on improving performance for GPU applications by fully taking advantage of the high storage density of racetrack memory. We also propose a novel compile-time register mapping algorithm to minimize the shift operations.

7 Conclusions

The massive threading feature has enabled GPU to boost performance for a variety of applications. However, the number of simultaneously executing threads in GPUs is often constrained by the size of the register file and it could potentially be a serious bottleneck for performance. Moreover, the widely used SRAM-based register file does not scale well in power and area when the register file size is increased. In this work, we explored racetrack memory for designing high performance register file for GPUs. The high storage density feature of racetrack memory increases the register file capacity and subsequently enables more threads to execute in parallel. We also developed an optimization framework to minimize the shift operations to improve GPU performance. Our experiments showed that our racetrack-based register file can achieve up to 29% (21% on average) performance improvement for a variety of GPU applications.

Acknowledgments We thank the anonymous reviewers for their feedback.

References

[1] Gebhart M, Keckler S W, Khailany B, Krashinsky R, Dally W J. Unifying primary cache, scratch, and register file memories in a throughput processor. In *Proc. the 45th Annual IEEE/ACM International Symposium on Microarchitecture* (*MICRO*), Dec. 2012, pp.96-106.

[2] Li X, Liang Y. Energy-efficient kernel management on gpus. In *Proc. the Design Automation and Test in Europe* (*DATE*), Mar. 2016.

[3] Liang Y, Huynh H, Rupnow K, Goh R, Chen D. Efficient GPU spatial-temporal multitasking. *IEEE Transactions on Parallel and Distributed Systems*, 2015, 26(3): 748-760.

[4] Liang Y, Xie X, Sun G, Chen D. An efficient compiler framework for cache bypassing on GPUs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2015, 34(10): 1677-1690.

[5] Xie X, Liang Y, Li X, Wu Y, Sun G, Wang T, Fan D. Enabling coordinated register allocation and thread-level parallelism optimization for GPUs. In *Proc. the 48th Annual IEEE/ACM International Symposium on Microarchitecture* (*MICRO*), Dec. 2015.

[6] Xie X, Liang Y, Sun G, Chen D. An efficient compiler framework for cache bypassing on GPUs. In *Proc. the International Conference on Computer Aided Design* (*ICCAD*), Nov. 2013, pp.516-523.

[7] Xie X, Liang Y, Wang Y, Sun G, Wang T. Coordinated static and dynamic cache bypassing on GPUs. In *Proc. the 21st IEEE International Symposium on High Performance Computer Architecture* (*HPCA*), Feb. 2015, pp.76-88.

[8] Mao M,Wen W, Zhang Y, Chen Y, Li H H. Exploration of GPGPU register file architecture using domain-wall-shift-write based racetrack memory. In *Proc. the 51st Annual Design Automation Conference* (*DAC*), June 2014, pp.196:1-196:6.

[9] Zhang C, Sun G, Zhang W, Mi F, Li H, Zhao W. Quantitative modeling of racetrack memory, a tradeoff among area, performance, and power. In *Proc. the 20th Asia and South Pacific Design Automation Conference* (*ASP-DAC*), Jan. 2015, pp.100-105.

[10] Parkin S S P, Hayashi M, Thomas L. Magnetic domain-wall racetrack memory. *Science*, 2008, 320(5873): 190-194.

[11] Sun Z, Wu W, Li H. Cross-layer racetrack memory design for ultra high density and low power consumption. In *Proc. the 50th Annual Design Automation Conference* (*DAC*), May 2013, Article No. 53.

[12] Venkatesan R, Ramasubramanian S G, Venkataramani S, Roy K, Raghunathan A. Stag: Spintronic-tape architecture for GPGPU cache hierarchies. In *Proc. the 41st Annual International Symposium on Computer Architecture* (*ISCA*), Jun. 2014, pp.253-264.

[13] Jing N, Shen Y, Lu Y, Ganapathy S, Mao Z, Guo M, Canal R, Liang X. An energy-efficient and scalable eDRAM-based register file architecture for GPGPU. In *Proc. the 40th Annual International Symposium on Computer Architecture* (*ISCA*), Jun. 2013, pp.344-355.

[14] Wang S, Liang Y, Zhang C, Xie X, Sun G, Liu Y, Wang Y, Li X. Performance-centric register file design for GPUs using racetrack memory. In *Proc. the 21st Asia and South Pacific Design Automation Conference* (*ASP-DAC*), Jan. 2016.

[15] Kayiran O, Jog A, Kandemir M T, Das C R. Neither more nor less: Optimizing thread-level parallelism for GPGPUs. In *Proc. the 22nd International Conference on Parallel Architectures and Compilation Techniques* (*PACT*), Oct. 2013, pp.157-166.

[16] Jog A, Mishra A K, Xu C, Xie Y, Narayanan V, Iyer R, Das C R. Cache revive: Architecting volatile STT-RAM caches for enhanced performance in CMPs. In *Proc. the 49th Annual Design Automation Conference* (*DAC*), June 2012, pp.243-252.

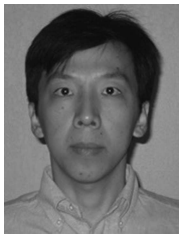
[17] Samavatian M H, Abbasitabar H, Arjomand M, Sarbazi-Azad H. An efficient STT-RAM last level cache architecture for GPUs. In *Proc. the 51st Annual Design Automation Conference* (*DAC*), May 2014, pp.197:1-197:6.

[18] Sun Z, Bi X, Li H H, Wong W F, Ong Z L, Zhu X, Wu W. Multi retention level STT-RAM cache designs with a dynamic refresh scheme. In *Proc. the 44th Annual International Symposium on Microarchitecture* (*MICRO*), Dec. 2011, pp.329-338.

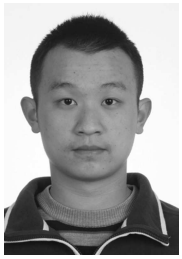
[19] Chen X, Sha E H M, Zhuge Q, Dai P, Jiang W. Optimizing data placement for reducing shift operations on domain wall memories. In *Proc. the 52nd Annual Design Automation Conference* (*DAC*), June 2015, pp.139:1-139:6.

[20] Venkatesan R, Kozhikkottu V, Augustine C, Raychowdhury A, Roy K, Raghunathan A. TapeCache: A high density, energy efficient cache based on domain wall memory. In *Proc. the International Symposium on Low Power Electronics and Design* (*ISLPED*), July 30-August 1, 2012, pp.185-190.

[21] Jing N, Liu H, Lu Y, Liang X. Compiler assisted dynamic register file in GPGPU. In *Proc. the International Symposium on Low Power Electronics and Design* (*ISLPED*), Sept. 2013, pp.3-8.



Yun Liang obtained his B.S. degree in software engineering from Tongji University, Shanghai, and his Ph.D. degree in computer science from National University of Singapore, in 2004 and 2010, respectively. He was a research scientist with Advanced Digital Science Center, University of Illinois Urbana-Champaign, Urbana, IL, USA, from 2010 to 2012. He has been an assistant professor with the School of Electronics Engineering and Computer Science, Peking University, Beijing, since 2012. His current research interests include graphics processing unit (GPU) architecture and optimization, heterogeneous computing, embedded system, and high level synthesis. Dr. Liang was a recipient of the Best Paper Award in International Symposium on Field-Programmable Custom Computing Machines (FCCM) 2011 and the Best Paper Award nominations in International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) 2008 and Design Automation Conference (DAC) 2012. He serves a technical committee member for Asia South Pacific Design Automation Conference (ASPDAC), Design Automation and Test in Europe (DATE), International Conference on Compilers Architecture and Synthesis for Embedded System (CASES), and International Conference on Parallel Architectures and Compilation Techniques (PACT). He is the TPC subcommittee chair for ASPDAC 2013.



Shuo Wang received his B.S. degree in electrical engineering from Nanjing Agricultural University, Nanjing, in 2013 and his M.S. degree in electrical engineering from the University of Southern California, Los Angeles, in 2014. Since 2015, he is a Ph.D. candidate in Center for Energy-Efficient Computing and Application (CECA), Peking University, Beijing. His current research interests include compilation techniques for heterogeneous computing platforms.