Performance-centric Register File Design for GPUs using Racetrack Memory

Shuo Wang[1], Yun Liang[1], Chao Zhang[1], Xiaolong Xie[1], Guangyu Sun[1], Yongpan Liu[2], Yu Wang[2], and Xiuhong Li[1]
[1]Center for Energy-Efficient Computing and Applications (CECA), EECS School, Peking University, China
[2]Department of Electronic Engineering, Tsinghua University, China
[1]{shvowang, ericlyun, zhang.chao, xiexl_pku, gsun, lixiuhong}@pku.edu.cn
[2]{ypliu, yu-wang}@tsinghua.edu.cn

Abstract— The key to high performance for GPU architecture lies in massive threading to drive the large number of cores and enable overlapping of threading execution. 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 costs so large amount of chip area that it cannot scale to meet the increasing demand of massive threading for GPU applications. Racetrack memory is a promising technology for designing large capacity register file on GPUs due to its high data storage density. However, without careful deployment of registers, the lengthy shift operation of racetrack memory may hurt the performance.

In this paper, we explore racetrack memory for designing high performance register file for GPU architecture. High storage density racetrack memory helps to improve the thread level parallelism, i.e., the number of threads that simultaneously execute. However, if the bits of the registers are not aligned to the ports, shift operations are required to move the bits to the ports. To mitigate the shift operation overhead problem, we develop a register file preshifting strategy and a compile-time managed register mapping algorithm. Experimental results demonstrate that our technique achieves up to 24% (19% on average) improvement in performance for a variety of GPU applications.

I. Introduction

Modern GPUs employ a large number of simple, in-order cores, delivering several TeraFLOPs peak performance. Traditionally, GPUs are mainly used for super-computers. More recently, GPUs have penetrated mobile embedded system markets. The system-on-a-chip (SoC) that integrates GPUs with CPUs, memory controllers, and other application-specific accelerators are available for mobile and embedded devices.

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 [5, 20, 18, 9, 19, 10]. 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 16KB L1 cache and 48KB shared memory.

The hardware resources on GPUs include (i) registers (ii) shared memory (iii) threads and thread blocks. A GPU kernel will launch as many threads concurrently as possible until one or more dimension of resource are exhausted. We use occupancy to measure the thread level parallelism (TLP). Occupancy is defined as the ratio of simultaneously active threads to the maximum number of threads supported on one SM. Though GPUs are featured with large register file, in reality, we find that the TLP is often limited by the size of register file. Table I gives the characteristics of some representative applications from Parboil[2] and Rodinia[3] benchmark suites on a Fermi-like architecture. The setting of the GPU architecture is shown by Table II. The maximum number of threads that can simultaneously execute on this architecture is 1,536 threads per streaming multiprocessor (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 II), only two thread blocks can run simultaneously. This leads to a very low occupancy as $\frac{2\times256}{1536} = 33\%$.

TABLE I. Kernel Description

Application	Register per thread	Occupancy	Register utilization
hotspot[3]	60	33%	93.75%
b+tree[3]	30	67%	93.75%
histo_final[2]	41	33%	64.06%
histo_main[2]	22	50%	51.56%
mri-gridding[2]	40	50%	93.75%

TABLE II. GPGPU-Sim Configuration

# Compute Units (SM)	15
SM configuration	32 cores, 700MHz
Resources per SM	Max 1536 threads, Max 8 thread blocks, 48KB shared memory, 128KB 16-bank register file(32768 registers)
Scheduler	2 warp schedulers per SM, round-robin policy
L1 Data Cache	16/32KB, 4-way associativity, 128B block, LRU replacement policy, 32 MSHR entries
L2 Unified Cache	768KB 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 [7, 17]. 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 [12], racetrack memory has shown about 28X higher density compared to SRAM [21]. Recent study has also enabled racetrack memory for GPU register file design [11]. However, prior work primarily focuses on optimizing energy efficiency, resulting in very small performance gain [11].

In this paper, we explore racetrack memory for designing high performance register file for GPU architecture. High storage density racetrack memory helps to enlarge the register file capacity. This allows the GPU applications to run more threads in parallel. However, racetrack-based design presents a new challenge in the form of shifting overhead. More clearly, for racetrack memory, 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 a register file preshifting strategy and a compile-time register mapping algorithm that optimizes the mapping of registers to the physical address in the register file. The preshifting strategy is implemented in hardware and tries to preshift the unaligned racetracks toward the corresponding access ports as much as possible.The mapping algorithm consists of two phases. In the first phase, our algorithm partitions the registers into groups. The registers in the same group have frequent inter-communication. Thus, we will align them to the same offset along multiple ports so that they can be accessed simultaneously via multiple ports without shifting overhead. In the second phase, our algorithm determines the mapping for each group. By partioning the registers into groups, this also helps to scale down the problem size.

The key contributions of this paper are as follows,

- Framework. We develop a register file design and compile-time

978-1-4673-9569-4/16/$31.00 ©2016 IEEE

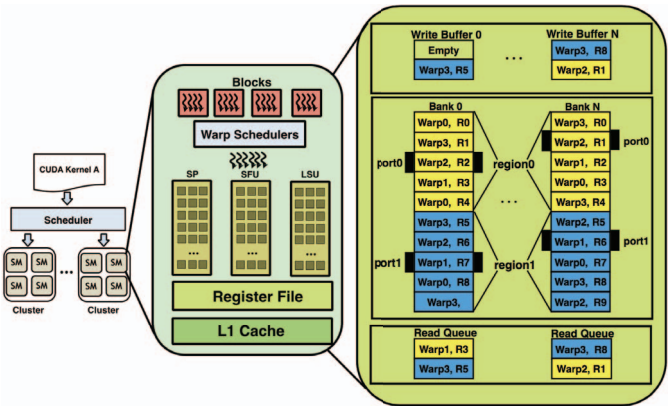


Fig. 1. GPU and register file architecture.

register mapping framework for GPUs using racetrack memory.

- Optimization techniques. We develop a architecture-level preshifting strategy and compile-time register mapping algorithms to minimize the number of shift operations of RM-based register file.
- We conduct experiments using a variety of applications and show that compared to SRAM design, our racetrack-based register file design improves performance by 19% on average.

II. Background and Motivation

A. GPU Architecture

A GPU is a highly parallel, multithreaded, many-core processor [19, 10]. The design of our baseline GPU is based on the GPGPU-Sim[1](version 3.2.2) as shown in Figure 1. The SMs are grouped into core clusters and each SM contains many streaming processors (SPs). Each SM coordinates the single-instruction multiple-data (SIMD) execution of its SPs. All the SPs in an SM share the computing and memory resource including warp scheduler, register file, special purpose units (SPUs), L1 cache, and shared memory, etc. The detailed setting of the GPU architecture used in this paper is shown in Table II.

The structure of the GPU kernel written in CUDA programming model is closely related to the GPU architecture. A GPU kernel is composed of many threads hierarchically organized into groups called *thread blocks*. At run-time, GPU thread block scheduler dispatchs the threads to SMs at the unit of thread blocks. The thread block scheduler tries to dispatch as many thread blocks as possible to each SM until the resource limitation is reached. The threads in a thread block are further grouped into *warps*, set of 32 threads. Warps are the basic scheduling units on GPU.

B. Racetrack Memory

More recently, racetrack memory is becoming increasingly popular due to its high storage density [12, 17, 15]. Compared with SRAM, Racetrack memory (RM) could achieve about 28X storage density while keeping comparable access speed [21]. Analogous to access 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

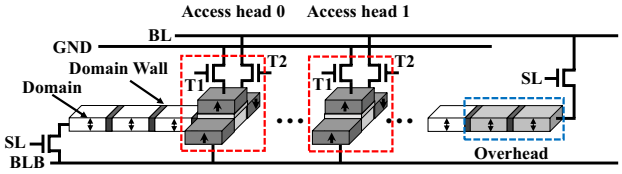


Fig. 2. Physical structure of a racetrack memory cell.

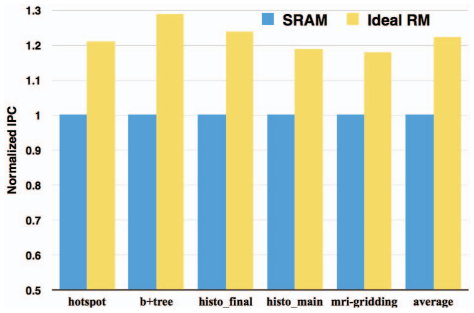


Fig. 3. GPU performance in different applications.

save bits and several access ports to access them, as shown in Figure 2. The "white bricks" represent the domains and those separations are domain walls used to isolate successive domains. Each access head has two transistors T1 and T2, as outlined by the red dash boxes. T1 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". T2 is used to write bit into a domain. When T2 is on, a crosswise current is applied on the domain, and the required bit is shifted in just as a shift operation.

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.

C. Motivation

GPUs employ massive threading to hide the latency of functional unit and memory operations. However, as shown by Table I, the GPU occupancy is often limited by the size of register file. Certainly, larger register file will help to unleash the power of massive threading for them. In Figure 3, we compare the performance of GPU in five applications under the same chip area budget for register file[1]. By doubling the register file size, we achieve up to 29% (on average 22%) performance improvement. The improvement is attributed by more occupancy enabled by larger register file. For example, for application mri-gridding, the occupancy is improved from 50% to 100% by doubling the register file size. Note that this improvement is the ideal performance improvement by assuming the same access delay for 128KB and 256KB register file and there is enough die area for enlarging register file size. In this paper, we will explore racetrack based register file design for GPU architecture. Racetrack memory has higher storage density and almost no access latency overhead after the capacity is increased. We will demonstrate that our design using racetrack memory can achieve the performance close to the ideal case.

III. Register Mapping Problem

A. Racetrack-based Register File

The architecture of our racetrack-based register file is shown in Figure 1. Similar to the SRAM register file design, it partitions the registers into multiple banks (e.g. 16). Each bank is equipped with multiple ports for simultaneous reading and writing. Each entry in a bank stores the registers for a warp. Warps are the scheduling units in GPUs. All the threads in a warp all executed together in a lock style. Thus, the same index registers of the threads in the same warp are stored in an entry in a bank. Each entry uses a number of racetrack stripes to store the registers. One warp is often allocated dozens of registers. These registers are spread across the banks as follows,

$$Bank\_id = (Warp\_id + Register\_id) \mod Num\_Bank \quad (1)$$

[1]The area budget is set according to the area of a 16-bank 128KB register file as shown in TABLE II.

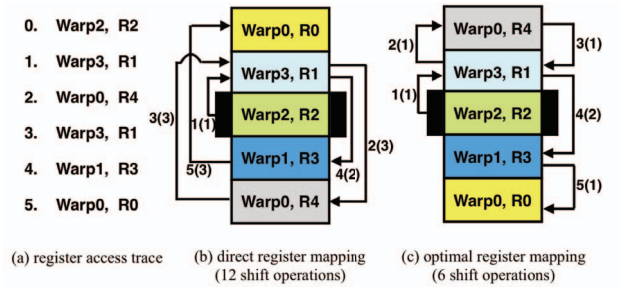


Fig. 4. Comparison of different register mapping. For subfigure (b) and (c), each arrow is associated with $m(n)$, where m represents the m^{th} register access in the trace, and n represents the shift operations.

where Num_Bank refers to the total number of banks of the register file, and $Bank\_id$ refers the allocated *bank id* of the register.

Multiple access ports are uniformly distributed along the racetracks in a bank. Each access port can only serve the register access requests within its private region as shown by Figure 1. Let us use N_d to denote the number of entries a bank could contain, and N_p to denote the number of ports a bank has. Then, $\lceil \frac{N_d}{N_p} \rceil$ is the number entries allocated to each port. The racetrack stripes within each bank have to be shifted simultaneously. Multiple access ports in a bank can be used to access multiple registers that are aligned to them simultaneously. More clearly, given N_p ports, N_p registers can be accessed simultaneously. However, to access other registers in a bank, shift operations are required to move the racetrack stripes to the nearest access ports. Obviously, a shift operation induces extra timing and energy overhead.

When we double the capacity of register file from 128KB to 256KB using racetrack memory, we create two racetrack arrays. Each racetrack array uses the architecture shown in Figure 1. Finally, to improve the write efficiency, similar to prior work [11], we employ a small write buffer for each bank as shown in Figure 1.

B. Problem Formulation

Our compile-time managed register mapping is implemented based on GPGPU-Sim [1] compilation and run-time system. We analyze the assembly-like PTX (Parallel Thread Execution) code, the intermediate representation used in NVIDIA CUDA Compiler. Each PTX instruction can access up to 4 registers. We generate the register access trace (sequence of register access) generated by executing the GPU application on the target architecture. Then, based on the register access trace, we optimize the mapping of the registers to the physical address (e.g. entry) within a bank at compile-time. The original PTX code does not support physical register allocation, and we implement our register mapping algorithm by employing the register allocation framework proposed in [18].

Figure 4 (a) shows a snippet of the register file access trace of $bank0$. There are totally 5 registers (R0 - R4) accessed in the bank and the trace contains 6 accesses to the registers. If we map the registers to the bank simply in an ascending order of register index as shown in Figure 4 (b), the racetracks need 12 shift operations to access the registers for this trace. The optimal mapping of register is shown in Figure 4 (c). By exchanging the positions of register $R0$ and $R4$, 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 physical address in racetrack register file to reduce the shift operations and ensure high throughput.

In the following, we formulate the register mapping problem. We observe that the register access trace 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,

$$m(R_i) = p(R_i) \times \frac{N_d}{N_p} + o(R_i) \tag{2}$$

where $0 < p(R_i) < N_p$ and $0 \leq o(R_i) < \frac{N_d}{N_p}$. 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, $\forall 0 \leq i, j \leq N_r - 1, m(R_i) \neq m(R_j)$.

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

$$S = \sum_{0 \leq i,j \leq N_r - 1} C_{ij} \cdot d(m(R_i), m(R_j)) \tag{3}$$

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

$$d(m(R_i), m(R_j)) = |o(R_i) - o(R_j)| \tag{4}$$

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.*

IV. Optimization Techniques

A. Register Mapping Algorithm

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

A.1 Register Grouping Algorithm

We split the objective function (Equation 3) into two parts based on $o(R_i)$ and $o(R_j)$ as follows,

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

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,

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

From Equation 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 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

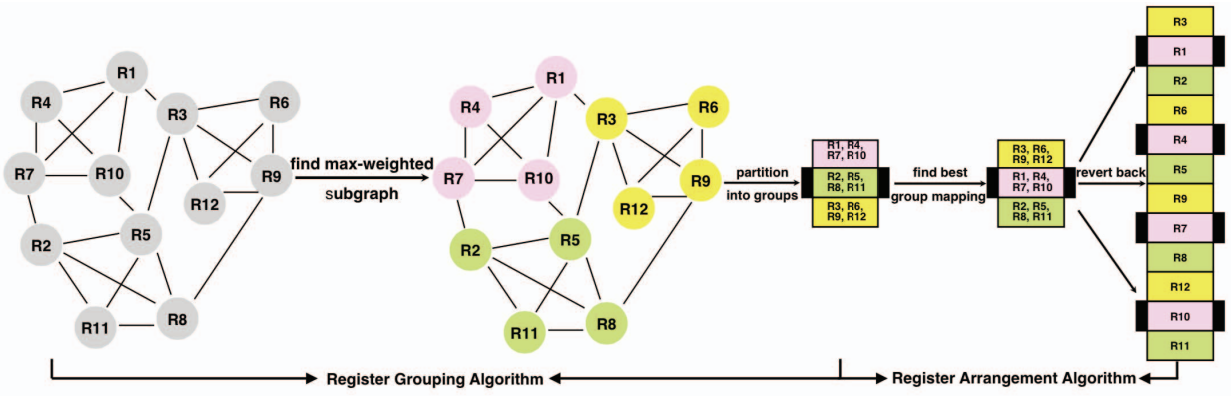


Fig. 5. Illustration of register mapping algorithm.

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         delete G' from G
11     group_id ← group_id + 1;
12     else
13         G'(V', E') ← FindMaxSubgraph(G', n);
14         foreach v_i ∈ V do
15             o(R_i) ← group_id;
16             p(R_i) ← port_num;
17             port_num ← port_num + 1;
18 FindMaxSubgraph(G = (V, E), Size){ V' ← ∅, E' ← ∅;
19 while |V'| ≤ Size do
20     w_d ← 0, w_e ← 0;
21     foreach v_t ∈ V do
22         w ← Σ_{v_i,v_j ∈ (V_s ∪ v_t), i≠j} C_{i,j};
23         if w_d ≤ w then
24             w_d ← w;
25             v_d ← v_t;
26     foreach e(v_m, v_n) ∈ V do
27         w ← Σ_{v_i,v_j ∈ (V_s ∪ (v_m,v_n)), i≠j} C_{i,j};
28         if w_e ≤ w then
29             w_e ← w;
30             V_e ← {v_m, v_n};
31     if w_d ≥ w_e and |V_s| ≤ size − 1 then
32         V' ← V' ∪ v_d;
33         V ← V − v_d;
34         else if w_e > w_d and |V'| ≤ size − 2 then
35             V' ← V' ∪ V_e;
36             V ← V − V_e;
37 E' = {e(v_i, v_j) | v_i, v_j ∈ V', v_i ≠ v_j};
38 return G' = (V', E');
39 }
```

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 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,

$$W(e(v_i, v_j)) = C_{i,j} + C_{j,i} \tag{7}$$

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 at $\sum_{\forall e \in G'} W(e)$ 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 (line 21-25) or two nodes (line 26-30) depending on which results in larger weight (line 31-36). Each time $FindMaxSubgraph$ 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' (line 7-9, 15-17). As a byproduct of this process, we assign group identifier for each group based on the sequence of when the group is formed. Figure 5 illustrates Algorithm 1 using an example. In this example, $N_p = 4$. We partition the registers into 3 groups with group size of 4.

A.2 Register Arrangement Algorithm

Let $\sigma \in \Omega$, then $\sigma[i]$ is the offset of the ith group. Algorithm 2 presents the details for register arrangement algorithm. It finds the best mapping that minimizes S (Equation 3) and returns $m(R_i)$ for each register. Figure 5 illustrates the register arrangement algorithm using an example. In this example, $N_r = 12$ and $N_p = 4$. By partitioning the registers into 3 groups, we can easily determine the mapping for groups through enumeration (3!) and then derive the mapping for all the registers.

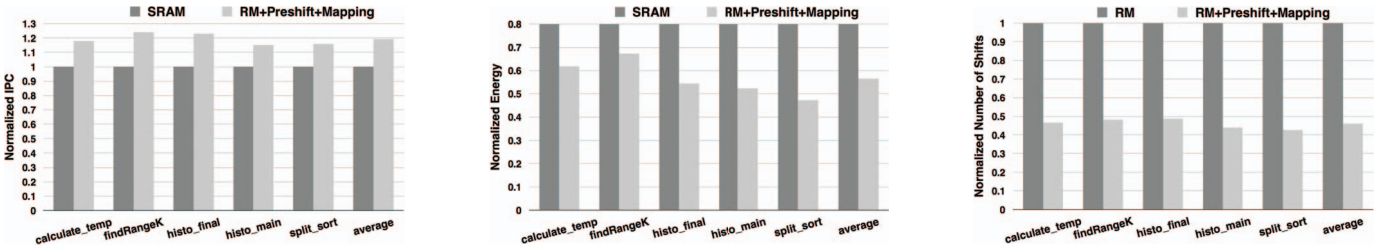
Algorithm 2: Register Arrangement Algorithm

```
   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)]|;
4      if S' ≤ S then
5          S ← S';
6          foreach register R_i do o(R_i) = σ[g(R_i)];
   // compute the final mapping function
7  foreach register R_i do m(R_i) ← p(R_i) · N_d/N_p + o(R_i);
```

B. Register File Preshifting Strategy

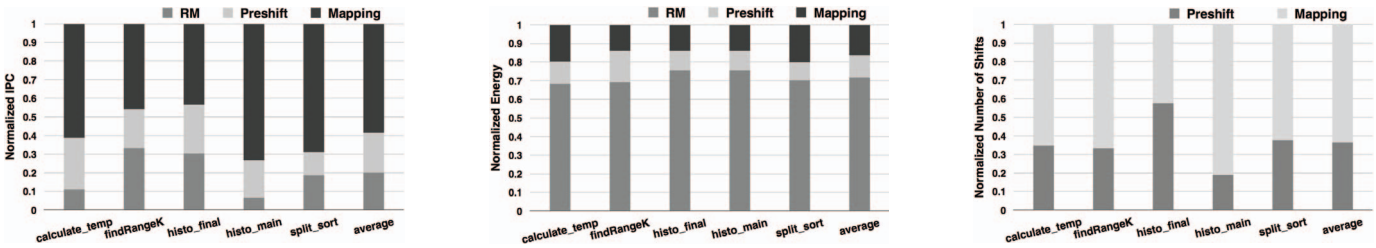
In this Section we also propose a preshifting strategy which is implemented in hardware to further reduce the shift operation overhead.

GPU register file adopts a multi-banked 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 regard the banks that is being accessed as *busy banks*, while the banks which are not being accessed as



(a) performance (b) energy (c) shift number

Fig. 6. Performance, energy, and shift number in different applications.



(a) performance breakdown (b) energy breakdown (c) shift number breakdown

Fig. 7. GPU perfromance, energy, and shift breakdown in different applications.

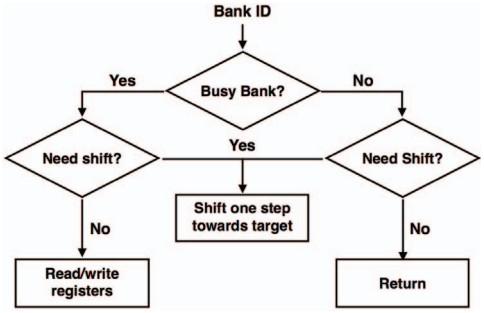


Fig. 8. The flow of opportunistic register file preshifing strategy.

idle banks. Each bank of register file owns a read request queue and write buffer which is shown in Figure 1. 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, when a bank is appointed to be *busy bank* to serve a request, 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 request has higher priority than the read request, so the write request will always be first served when there is a read and write request conflict.

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 a *idle bank* preshifting mechanism shown in Figure 8. In the following, we will use the register file of NIVIDA fermi architecture as an example to explain the preshifting mechanism.

First, in any given clock cycle, after the 4 *busy banks* are determined by the register file arbitrator, each bank will be checked wether it needs shift. If it is a *busy bank*, it will be read/written when it doesn't need a shift or be shifted toward target register when it does need a shift. While if the bank is an *idle bank*, it will also be checked whether it needs a shift operation or not. If it needs to shift to align the register to corresponding access port, the bank will shift one step towards access port, otherwise it will do nothing.

V. Experimental Evaluation

We implement our racetrack-based register file design based on GPGPU-sim[1]. We evaluate our technique using applications from benchmark suites Rodinia[3] and Parboil [2] as shown by Table I. We extend GPGPU-sim with racetrack memory model using a circuit-level racetrack memory simulator [21]. The design parameters of racetrack-based register file are shown in Table III. Racetrack-memory incurs long write latency. To mitigate this problem, similar to prior study [11], we employ a write buffer to improve the writing efficiency. The write buffer is 2KB and each bank has two entries as shown by Figure 1.

TABLE III. SRAM and RM operating parameters

Register File	SRAM	RM	Write Buffer
capacity	128KB	256KB	2KB
bank #	16	16	-
read latency	0.31ns	0.28ns	0.15 ns
write latency	0.31ns	1.24ns	0.16ns
shift latency	-	0.61ns	-
read energy	218.88pJ	117.12pJ	15.6pJ
write energy	57.28pJ	173.22pJ	14.6pJ
shift energy	-	56.16	-
leakage	12.31mW	7.95mW	1.12mW

A. Performance Results

Figure 6a shows that the performance improvement of our proposed RM based register file design over the default SRAM design is up to 24% (19% on average). The reasons for the improvement are two folds. First, under the same chip area budget, high-density RM based register file could enable larger capacity (256KB) compared to default SRAM design (128KB). The increased capacity of register file enables the applications to execute more threads in parallel as to make GPU achieve high occupancy as shown in Figure 9, where the occupancy is increased from 46.6% to 86.8% on average. Second, compared with the direct register mapping[2] our proposed register file preshifting strategy and register mapping technique have greatly reduced the number of shift operations by 54% on average shown in Figure 7c, which has largely alleviated the lengthy shift operation overheads.

To further quantify the respective portions of contribution from these propsed techniques, we break down the contributions and show the results in Figure 7a. It shows that RM-based register file, preshifting strategy, and proposed mapping algorithm contributes to 20%, 21%, and 59% performance improvement respectively. Moreover, we can see from the Figure 7c that preshifting strategy and the proposed mapping algorithm contributes to 37% and 64% number of shift oper-

[2]The direct mapping strategy is to directly map each virtual register to a physical address in register file according to its virtual register number

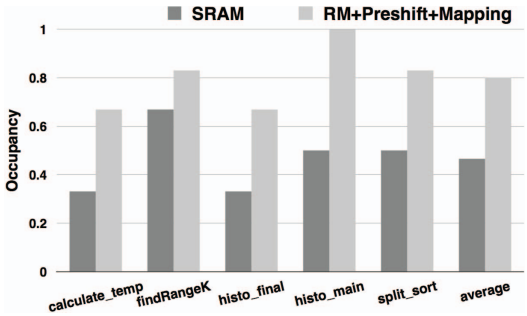


Fig. 9. GPU occupancy in different applications.

ation reduction on average, which matches the performance improvement contribution results we have concluded from Figure 7a.

B. Energy Results

Figure 6b shows that, compared with default SRAM design our proposed RM design reduces up to 53% (on average 43%) energy consumption. Furthermore, as shown in Figure 7b, RM-based register file, preshifting strategy, and mapping algorithm contributes to 72%, 12%, and 16% energy reduction respectively, which means that RM-based register file is major reason for the great energy reduction, thus indicating that RM-based register file is a promising solution for energy-efficient GPU register file design in the future.

C. Area Results

RM based register file saves 45% chip area over the SRAM based register file, and this large area reduction is mainly attributed to the high storage density of racetrack based memory, which indicates a good scalability of racetrack memory based register file for GPU.

VI. 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[8]. Topics about using STT-RAM to architect last level Cache has been studied in[13, 14]. Optimization techniques at architecture and compilation level are proposed for racetrack memory [15]. Chen et al. has proposed a data placement strategy to minimize the shift operations for racetrack based memory in general CPU[4]. 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 [16], 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 [17], 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 [17, 11]. Mao et al. presented a racetrack memory based GPGPU register file[11]. 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 [7, 6]. However, all the previous works using emerging technology for GPU register file mainly focus 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.

VII. Conclusion

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. 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 explore 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 develop an architecture-level preshifting strategy and a compile-time register mapping technique to minimize the shift operations to further improve performance. Our experiments show that our racetrack-based register file can achieve up to 24% (19% on average) performance improvement for a variety of GPU applications.

VIII. Acknowledgments

This work was supported by the National Science Foundation China (No. 61300005,61572045). We thank the anonymous reviewers for their feedback.

References

[1] GPGPU-Sim v3.2.2. https://github.com/gpgpu-sim/gpgpu-sim_distribution.

[2] Parboil Benchmark Suite. http://impact.crhc.illinois.edu/Parboil/parboil.aspx.

[3] Rodinia Benchmark Suite. http://www.cs.virginia.edu/~skadron/wiki/rodinia/index.php/Downloads.

[4] Xianzhang Chen, Edwin H.-M. Sha, Qingfeng Zhuge, Penglin Dai, and Weiwen Jiang. Optimizing data placement for reducing shift operations on domain wall memories. In *Proceedings of the 52nd Annual Design Automation Conference (DAC)*, pages 139:1–139:6, 2015.

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

[6] Naifeng Jing, Haopeng Liu, Yao Lu, and Xiaoyao Liang. Compiler assisted dynamic register file in gpgpu. In *Low Power Electronics and Design (ISLPED), 2013 IEEE International Symposium on*, pages 3–8, Sept 2013.

[7] Naifeng Jing, Yao Shen, Yao Lu, Shrikanth Ganapathy, Zhigang Mao, Minyi Guo, Ramon Canal, and Xiaoyao Liang. An energy-efficient and scalable edram-based register file architecture for gpgpu. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pages 344–355.

[8] Adwait Jog, Asit K. Mishra, Cong Xu, Yuan Xie, Vijaykrishnan Narayanan, Ravishankar Iyer, and Chita R. Das. Cache revive: Architecting volatile stt-ram caches for enhanced performance in cmps. In *Proceedings of the 49th Annual Design Automation Conference*, pages 243–252.

[9] Yun Liang, H.P. Huynh, K. Rupnow, R.S.M. Goh, and Deming Chen. Efficient gpu spatial-temporal multitasking. *Parallel and Distributed Systems, IEEE Transactions on*, 26(3):748–760, 2015.

[10] Yun Liang, Xiaolong Xie, Guangyu Sun, and Deming Chen. An efficient compiler framework for cache bypassing on gpus. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 34(10):1677–1690, 2015.

[11] Mengjie Mao, Wujie Wen, Yaojun Zhang, Yiran Chen, and Hai (Helen) Li. Exploration of GPGPU Register File Architecture Using Domain-wall-shift-write Based Racetrack Memory. In *Proceedings of the 51st Annual Design Automation Conference (DAC'14)*, pages 196:1–196:6, 2014.

[12] Stuart S. P. Parkin, Masamitsu Hayashi, and Luc Thomas. Magnetic domain-wall racetrack memory. *Science*, 320:190–194, 2008.

[13] Mohammad Hossein Samavatian, Hamed Abbasitabar, Mohammad Arjomand, and Hamid Sarbazi-Azad. An efficient stt-ram last level cache architecture for gpus. In *Proceedings of the 51st Annual Design Automation Conference*, pages 197:1–197:6.

[14] Zhenyu Sun, Xiuyuan Bi, Hai (Helen) Li, Weng-Fai Wong, Zhong-Liang Ong, Xiaochun Zhu, and Wenqing Wu. Multi retention level stt-ram cache designs with a dynamic refresh scheme. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011.

[15] Zhenyu Sun, Wenqing Wu, and Hai Li. Cross-layer racetrack memory design for ultra high density and low power consumption. In *50th Design Automation Conference (DAC), 2013*, pages 1–6, May 2013.

[16] Rangharajan Venkatesan, Vivek Kozhikkottu, Charles Augustine, Arijit Raychowdhury, Kaushik Roy, and Anand Raghunathan. Tapecache: A high density, energy efficient cache based on domain wall memory. In *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*, pages 185–190.

[17] Rangharajan Venkatesan, Shankar Ganesh Ramasubramanian, Swagath Venkataramani, Kaushik Roy, and Anand Raghunathan. Stag: Spintronic-tape architecture for gpgpu cache hierarchies. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 253–264, 2014.

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

[19] Xiaolong Xie, Yun Liang, Guangyu Sun, and Deming Chen. An efficient compiler framework for cache bypassing on gpus. In *International Conference on Computer Aided Design (ICCAD 2013)*, pages 516–523.

[20] Xiaolong Xie, Yun Liang, Yu Wang, Guangyu Sun, and Tao Wang. Coordinated static and dynamic cache bypassing on gpus. In *21st IEEE International Symposium on High Performance Computer Architecture (HPCA 2015)*, pages 76–88.

[21] Chao Zhang, Guangyu Sun, Weiqi Zhang, Fan Mi, Hai Li, and Weisheng Zhao. Quantitative modeling of racetrack memory, a tradeoff among area, performance, and power. In *Design Automation Conference (ASP-DAC), 2015 20th Asia and South Pacific*, pages 100–105, Jan 2015.