VERTICAL MEMORY OPTIMIZATION FOR HIGH PERFORMANCE ENERGY-EFFICIENT GPU

by

Mengjie Mao

M.S. in Computer Science,

University of Science and Technology of China, 2012

B.S. in Computer Science,

South China University of Technology, 2009

Submitted to the Graduate Faculty of

the Swanson School of Engineering in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2016

UNIVERSITY OF PITTSBURGH

SWANSON SCHOOL OF ENGINEERING

This dissertation was presented

by

Mengjie Mao

It was defended on

March 24, 2016

and approved by

Hai Li, Ph.D., Associate Professor, Departmental of Electrical and Computer Engineering

Yiran Chen, Ph.D., Associate Professor, Departmental of Electrical and Computer Engineering

Alex K. Jones, Ph.D., Associate Professor, Departmental of Electrical and Computer Engineering

Zhi-Hong Mao, Ph.D., Associate Professor, Departmental of Electrical and Computer Engineering

Rami Melhem, Ph.D., Professor, Departmental of Computer Science

Dissertation Director: Hai Li, Ph.D., Associate Professor, Departmental of Electrical and Computer Engineering

VERTICAL MEMORY OPTIMIZATION FOR HIGH PERFORMANCE ENERGY-EFFICIENT GPU

Mengjie Mao, PhD

University of Pittsburgh, 2016

GPU heavily relies on massive multi-threading to achieve high throughput. The massive multi-threading imposes tremendous pressure on different storage components. This dissertation focuses on the optimization of memory subsystem including register file, L1 data cache and device memory, all of which are featured by the massive multi-threading and dominate the efficiency and scalability of GPU.

A large register file is demanded in GPU for supporting fast thread switching. This dissertation first introduces a power-efficient GPU register file built on the newly emerged *racetrack memory* (RM). However, the shift operators of RM results in extra power and timing overhead. A holistic architecture-level technology set is developed to conquer the adverse impacts and guarantees its energy merit. Experiment results show that the proposed techniques can keep GPU performance stable compared to the baseline with SRAM based RF. Register file energy is significantly reduced by 48.5%.

This work then proposes a *versatile warp scheduler* (VWS) to reduce the L1 data cache misses in GPU. VWS retains the intra-warp cache locality with a simple yet effective per-warp working set estimator, and enhances intra- and inter-thread-block cache locality using a thread block aware scheduler. VWS achieves on average 38.4% and 9.3% IPC improvement compared to a widely-used and a state-of-the-art warp schedulers, respectively.

At last this work targets the off-chip DRAM based device memory. An integrated architecture substrate is introduced to improve the performance and energy efficiency of GPU through the efficient bandwidth utilization. The first part of the architecture substrate,

thread batch enabled memory partitioning (TEMP) improves memory access parallelism. TEMP introduces thread batching to separate the memory access streams from SMs. The second part, *Thread batch-aware scheduler* (TBAS) is then designed to improve memory access locality. Experimental results show that TEMP and TBAS together can obtain up to 10.3% performance improvement and 11.3% DRAM energy reduction for GPU workloads.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

PREFACE

I would like to thank my advisors, Dr. Hai Li and Dr. Yiran Chen. Dr. Li and Dr. Chen are great researchers, they led me to the research world of GPU, non-volatile memory and neuromorphic computing. During the past four years, they gave me numerous insightful guidance, and their supports helped me overcome the difficulties during the study, and also helped me finish this dissertation. I would like to thank the committee members, Dr. Alex K. Jones, Dr. Zhi-Hong Mao, and Dr. Rami Melhem, for their help ameliorating for this dissertation.

I am very grateful to all the colleagues from EI-lab. It was my pleasure and honor to work with them during the past several years.

At last I would like to thank my family for their encouragement and support. I dedicate this PhD thesis to my wife, Xueying Zeng, who always stands with me and brings happy into my life everyday.

Mengjie Mao

Natick, MA

2016

1.0 INTRODUCTION

The functionality of GPU has extended from fixed graphic acceleration to general purpose computing including image processing, computer vision, machine learning and scientific computing. Because of its vast computational capability, GPU is widely deployed in various platforms ranging from hand-held devices to HPC systems. Modern GPU can support tens of thousands of concurrent threads executed in *single instruction multiple threads* (SIMT). The thread-level parallelism of GPU is being continuously improved; for example, over the past 5 years, the core count of the high-end NVIDIA GPU is increased by nearly 10×, allowing more than one hundred thousand of threads to run simultaneously in Maxwell architecture [3]. Such extreme multi-threading can effectively amortize the device memory access latency with the help of frequent hardware thread scheduling: if a thread suspends on waiting the data from memory, the hardware thread scheduler will almost always be able to switch to another thread from the huge thread pool. Nevertheless, massive parallelism in GPU also asserts unbearable pressure on the whole memory subsystem vertically ranging from the on-chip storage components to the off-chip device memory. The bottleneck of memory subsystem plays a crucial role limiting the continuous upscaling of GPU computation.

1.1 GPU REGISTER FILE

GPU *register file* (RF) is the nearest on-chip storage component to core. To support hardware thread switching, a large RF is indispensable. And as the thread-level parallelism of GPU keeps being scaled, the RF capacity is increased accordingly; for instance, compared to GTX 480, the RF size is increased by 4× in GTX 980, marking 512KB RF per *stream*

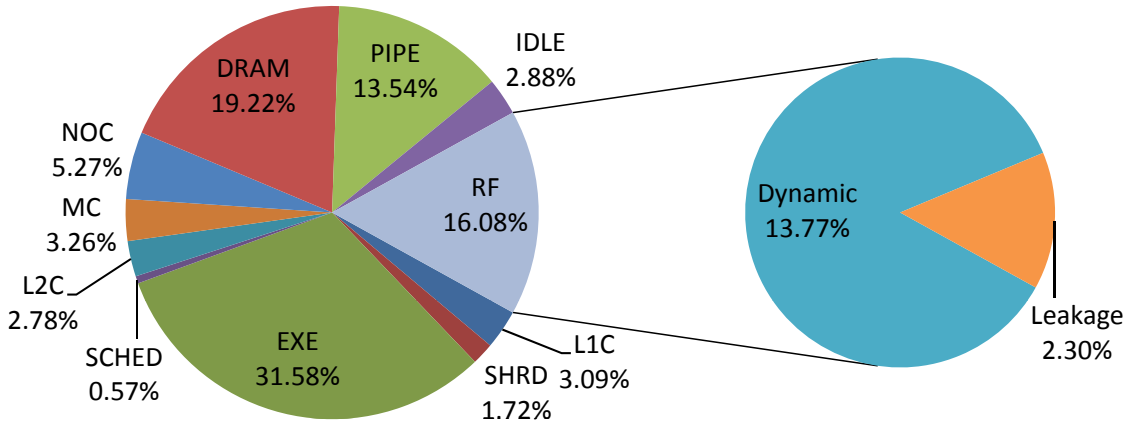

Figure 1: The power breakdown of GPU. The statistic is obtained from GPGPU-sim [1] coupled with GPUWattch [2]

multiprocessor (SM). Traditionally, GPU RF is built with SRAM and can consume up to 16% power of the whole GPU system. It is even comparable to the power consumption of DRAM or execution unit, as shown in Figure 1. As technology scales, the high leakage power and the large parametric variability of large on-chip SRAM based storage component have become the major obstacles impeding GPU power efficiency [4]. Substituting traditional memory technology with emerging memories [5, 6, 7, 8] appears as a promising approach to combat these scaling challenges in GPU designs, *e.g.*, using *spin-transfer-torque random access memory* (STT-RAM) or *embedded DRAM* to implement GPU RFs. Compared to their SRAM counterpart, these emerging memories generally offer smaller cell area or larger capacity with the same footprint, better scalability, and nonvolatility.

Intuitively, the access frequency of RF is much higher than that of other memory components, *e.g.*, on-chip cache, because nearly every instruction must access more than one register. The dynamic energy is expected to contribute a larger portion of the total RF energy consumption. We profile prevalent GPU benchmark suites [1, 9, 10, 11] and observe that the access frequency of RF is nearly 10× ∼30× higher than that of L1 cache. Figure 1 shows the power breakdown of RF at the right of Figure 1. The RF is built on SRAM and the bank level parameters of RF is extracted from CACTI [12] enhanced with GPU RF

model. The emerging memory candidates, however, all suffer from high write energy and slow write speed. Directly replacing traditional memory cells in the GPU RF with emerging memory cells may offset the benefits of high density and low leakage power offered by the emerging memory technologies.

1.2 GPU L1 DATA CACHE

While the RF of modern GPU keeps its size increasing reasonably, the other major on-chip storage component, L1 data cache almost keeps its size stable [13, 3] due to the design considerations including access speed, thermal and area. Cache hierarchy has been introduced to GPU designs to reduce the memory access latency by taking advantages of data locality. Frequent thread switching requires a large number of active threads pending for schedule, these active thread, however, incurs extremely small cache quota per thread [14]. Moreover, the increased GPU concurrency results in severer low per-thread L1 data cache quota [14]. As such, conventional cache hierarchy becomes less efficient in retaining the frequently reused data even the inherent data locality of the applications is high [15].

Prediction of the re-reference interval of cache block plays an important role in general cache replacement policy: based on the re-reference intervals, the position of the cache block insertion can be dynamically adjusted to approach ideal cache hit rate [16]. In GPU applications, however, even perfect prediction of re-reference interval cannot necessarily achieve a high cache hit rate. Because of high volume of thread contents on small cache capacity, the cache blocks of a thread can be easily evicted by those of other threads. Such situation becomes harmful to system throughput, if the cache blocks are re-useful but the re-reference interval is not short enough to keep the data in cache. Moreover, programming model of modern GPU often employs a hierarchical thread organization [17, 18], resulting in various levels of cache locality. Therefore, although the intra-thread cache locality has been well studied and supported by many replacement policies in CPU domain, retaining the cache localities among the GPU threads still remains as a very challenging topic.

Besides the replacement policies, thread scheduling is considered as another critical approach improving the cache efficiency [19, 20, 15, 21]. Through the rearrangement of thread execution order, the re-reference interval of the data of each thread can be dynamically reconfigured, and the cache quota can also adapt to the needs of the running threads. Both intra- and inter-thread cache localities are required in GPU execution. However, most prior arts primarily focus on capturing the intra-thread locality. A comprehensive scheduler design that is capable of exploring multiple different cache localities in GPU applications is necessary.

1.3 GPU DEVICE MEMORY

The low per-thread cache quota is incapable of filtering the huge volume of memory accesses to off-chip DRAM device memory[19], causing tremendous pressures on the DRAM device memory. The DRAM bandwidth is not increased proportional to the increasing of concurrent threads. From Fermi to Maxwell architecture, the DRAM bandwidth is only increased by two times when the number of concurrent threads is increased by more than eight times [13, 3]. On the other hand, since the DRAM protocol, GDDR5 [22] does not evolve accordingly, the increased bandwidth comes from the double-boosted clock rate [23]; consequently, the power consumption of the device memory is aggravated.

Increasing memory access parallelism and locality via memory scheduling is one of the primary architectural approaches to improve memory efficiency in multi-core systems [24, 25]. Unfortunately, most existing memory scheduling algorithms are incapable to handle the extremely high volume of memory accesses in GPU [26, 27] since their implementation cost is generally unaffordable [28].

Instead of dedicated hardware redesign of memory scheduler, *memory partitioning* based on OS memory management is another viable approach to improve memory usage efficiency. memory partitioning usually attempts to divide and allocate memory resources to threads, each of which accesses its associated memory partition exclusively to reduce inter-thread interference [29, 28, 30, 31]. Using memory partitioning to improve DRAM efficiency in

GPU is feasible and even attractive because: 1) The pageable unified memory address space in heterogeneous system allows OS to freely allocate the pages accessed by GPU; 2) The near-homogeneous multi-threading in GPU can guarantee the memory access fairness and parallelism if the running threads are evenly dispatched to SMs. This statement, however, may not be valid in multi-core systems due to the different bandwidth requirements of heterogeneous threads [31].

However, existing memory partitioning mechanisms of multi-core systems cannot be directly applied to the GPU domain. For instance, *memory bank partitioning* [29, 28, 30, 31] aims the multi-program environment where only limited number of threads are executed simultaneously. Each thread owns and accesses the assigned banks exclusively. In GPU applications, however, the number of the threads could be several orders of magnitude greater than that of the available banks. Each thread cannot own a (or a few) memory bank(s) exclusively. Moreover, all the threads in a GPU application share an unified address space, hindering the separation of memory access streams from different threads and making the implementation of memory partitioning very difficult.

1.4 DISSERTATION CONTRIBUTIONS

This dissertation tackles the bottleneck of memory subsystem in GPU. Three comprehensive solutions are proposed for three storage components:

- This dissertation introduces a GPU RF design with the emerging *racetrack memory* (RM) [32]. RM have extremely small cell area [33, 34], low read and write energy, and nonvolatility. However, the sequential access mechanism, which means that every read or write access requires sequentially shifting the target cell to the nearby access port on the track, results in a long and location-dependent access latency of RM. Hence, a series of techniques are introduced to alleviate the adverse impact of the sequential access of the RM based RF on the GPU performance as:
  1. A *register remapping* scheme is proposed to reorganize logical-physical register mapping to reduce the average shift distance in RW accesses;

2. A *RM aware warp scheduling (RMWS)* algorithm is developed to hide the shift delay by dynamically prioritizing the ready warps based on the current access port location on the RM tracks;
3. A *new write buffer* structure is designed to improve the scheduling efficacy of the RMWS and save the energy of the RM based RFs through eliminating of the unnecessary RF accesses;
4. A novel *warp-register mapping* technique is proposed to support efficient warp scheduling by removing the schedule hazard among multiple RMWS schedulers.

Our simulation results show that compared to SRAM based GPU RF design with normal warp scheduler, the application of RW based RF significantly reduce the energy consumption of RF by 48.5% while the system performance is kept stable.

- This dissertation then proposes a *versatile warp scheduler* (VWS) to reduce the cache miss rate in GPU [35]. Not only can VWS outperform the existing works on preserving intra-thread locality with much more efficient implementation, but also retain inter-thread cache locality for higher system throughput. The major contributions of VWS are:
  1. Three major types of cache localities are identified in GPU, and their impacts are quantitatively analyzed on cache usage efficiency;
  2. VWS is the first thread scheduler that is capable of retaining both intra- and inter-thread cache localities;
  3. An efficient working set estimator is designed to guide the thread scheduling decision for better intra-thread locality; a thread structure-aware scheduling policy and a workload dispatching mechanism are also introduced for better inter-thread locality.

  The simulation results show that compared to a widely-used thread scheduler, VWS can dramatically raise up the L1 data cache hits, and therefore improve the system performance by 38.4% on average. VWS also outperforms a state-of-the-art scheduler by 9.3%.
- At last this dissertation proposes to allocate memory banks to specific SMs for GPU memory access parallelism and locality enhancement. Specifically, a *thread batch enabled memory partitioning* (TEMP) is introduced to constrain the majority of the memory

accesses from a SM to the dedicated banks [36]. Thread blocks, which share the same set of pages with each other, are grouped into a thread batch. The thread batch is then dispatched to a SM as a whole and the pages accessed by the thread batch can be mapped to the banks dedicated to the SM by page coloring [37]. As a result, the memory access interweaving between SMs and banks is minimized and the memory access parallelism is maximized. The application of TEMP separates the memory access streams from different SMs, allowing further enhancement of memory access locality through optimizing the execution of the thread batches. *Thread batch-aware scheduling* (TBAS) is then proposed to prioritize thread batches in order to cluster the accesses to the row in a bank. The prioritization of thread batches also alleviates the congestion on memory controllers as well as the reply network from memory partitions to SMs. The major contributions of TEMP and TBAS are:

1. This is the pioneer work attacking DRAM bandwidth bottleneck in GPU applications running on heterogeneous shared memory systems utilizing memory partitioning;

2. Thread-data mapping in GPU applications is characterized. According to the characterization, TEMP is proposed to maximize memory access parallelism through the minimization of inter-SM interference of memory accesses;

3. TBAS is developed to improve memory access locality and mitigate the congestion on memory controller and reply network, further improving the efficacy of TEMP.

Experimental results show that after applying TEMP and TBAS, the performance of a set of GPU applications is improved by 10.3% while the DRAM energy consumption is reduced by 11.3%, w.r.t. the baseline without memory partitioning. The dissertation also evaluates the impact of the execution of heterogeneous CPU-GPU workloads on the integrated TEMP and TBAS substrate. The dissertation demonstrates that a simple yet effective solution is capable of addressing the interference incurred by CPU execution, ensuring TEMP and TBAS keep the high GPU throughput while barely deteriorating the CPU-side performance.

1.5 DISSERTATION ORGANIZATION

The rest of this dissertation is organized as follows. Chapter 2 introduces the background of GPU architecture. Chapter 3 first presents the array-level evaluation of RM based GPU RF, then describes the details on the architecture optimizations for RM based GPU RF, and gives the experimental results and the relevant analysis at the last. Chapter 4 analyzes the GPU cache locality, and then illustrates the details of VWS, at last presents the simulation results. Chapter 5 detailed the design of TEMP and TBAS, and shows the experiment results. Chapter 6 reviews the related works. Finally, Chapter 7 concludes the dissertation.

2.0 GPU ARCHITECTURE

Without loss of generality, we use NVIDIA Fermi [13] as an example to introduce the relevant knowledge of GPU architecture. A typical GPU architecture is shown in Figure 2. It consists of many SMs and *memory partitions*. SMs are connected to memory partitions with interconnection networks, *e.g.*, a two dimension mesh. Every SM holds a complete SIMD pipeline, which consists of instruction cache, warp scheduler, register files, shader cores, L1 data cache and scratchpad memory. Each memory partition has its own DRAM chips, dedicated memory controller, and L2 cache. All memory partitions are in the same memory address space so that a SM can access any memory partitions by address.

GPU programming models like CUDA [17] and OpenCL [18] define the workload offloaded to a GPU as a *kernel*. A kernel is highly multi-threaded where all the homogeneous threads are encapsulated in a *grid*. A grid is partitioned into *thread blocks*, each of which contains up to thousands of threads. During executions, each thread block is dispatched as a whole to a SM. A thread block in the SM can be further partitioned into fixed-size *warps* that are atomically scheduled by *warp scheduler* and executed by the shader cores in SIMD fashion. One SM in Fermi can concurrently run up to 48 warps. The shared scratchpad memory is employed in the SM to support intra-thread block data sharing. The L1 data cache is used to cache all other data and shared by all shader cores. Note that there is no hardware cache coherence protocol implemented in GPU. The L1 caches are write-evict write no-allocate for global data and write-back for per-thread private data. Therefore, the software can easily maintain the cache coherence.

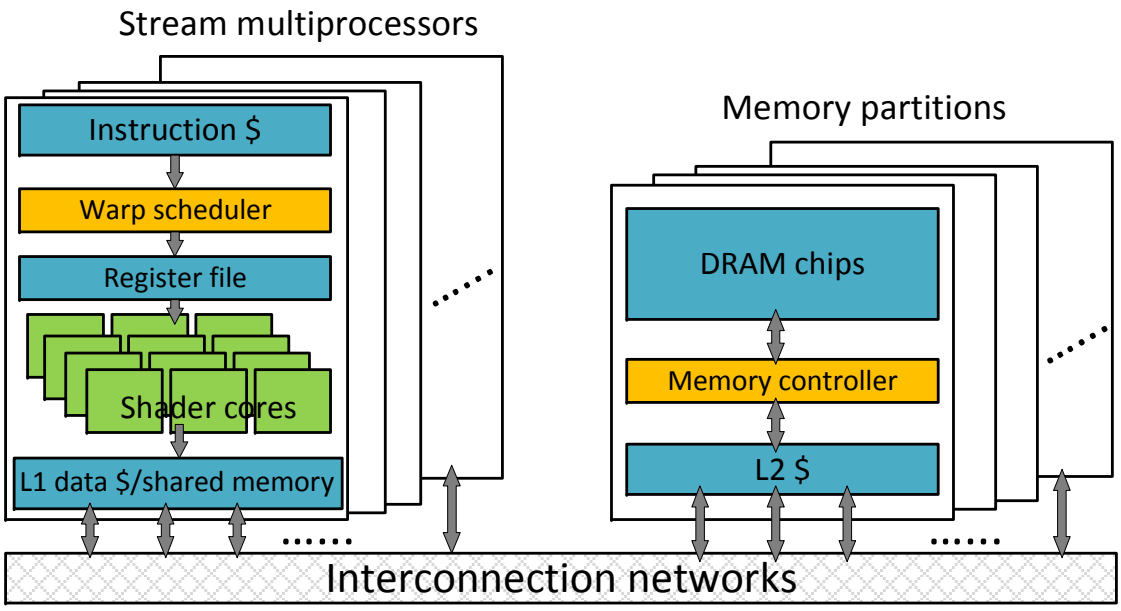

Figure 2: The GPU architecture.

3.0 POWER-EFFICIENT GPU REGISTER FILE

In this chapter we first introduce the GPU pipeline stages involved RF and the basics of RM. Then we present the array-level design of RM based RF, followed by the details of architecture optimization. We depict our experimental setup in Section 3.4 and show the experimental results and the relevant analysis in Section 3.5. At the end of this chapter we will summarize our works on GPU RF.

3.1 BACKGROUND

3.1.1 The GPU Pipeline Stages for RF Accessing

Figure 3 shows two stages – *issue* and *operand collection*, in a Fermi SM pipeline. At issue stage, warps are issued by a warp scheduler: if the selected warp has valid instruction in its *instruction buffer*, and confirms with scoreboard that it does not depend on previous instruction (notified by the r field), then it will be issued; otherwise, the scheduler will select another warp based on the scheduling algorithm. The issued instruction is attached with a mask by the SIMT-stack [38] that indicates the active threads within a warp.

Figure 3 (b) shows details of the data path involved in operand collection stage into which the issued instruction enter. The instruction is buffered in the collector unit, and the register requests from the instruction are sent to the arbitrator associated with each RF bank. The arbitrator solves the bank conflict of the pending requests. The GPU RF is highly banked to virtually support multi-port feature. In this work, we assume there are total 64 entries in a bank [2]; each entry is 128 bytes, containing 32 32-bit operands, namely, *warp register*[1]; and a single request can read/write one register.

[1]We use warp register and register interchangeably.

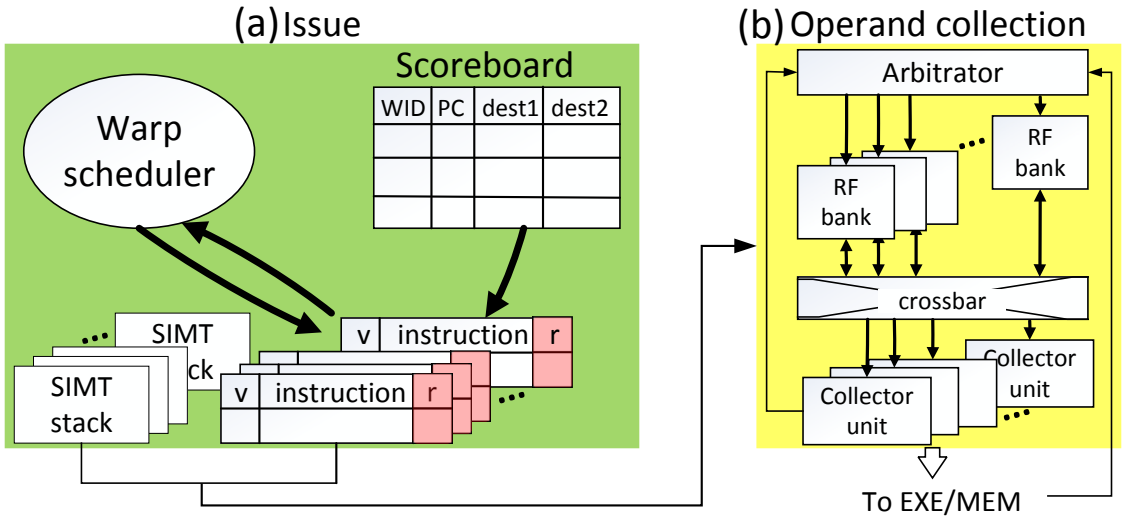

Figure 3: The (a) issue and (b) operand collection stage in GPU pipeline.

3.1.2 Domain-wall-shift-write based Racetrack Memory

Figure 4 (a) shows the schematic of one RM cell based on *domain-wall-shift-write* [39, 40]. As indicated in the name, write operations of such kind of cell rely on magnetic domain wall motion. The cell consists of a ferromagnetic wire, an MTJ and two access transistors. Different from conventional *1T1J* STT-RAM cell [41], the MTJ in the RM cell contains two fixed pinning regions whose magnetization orientations are opposite to each other. These two fixed regions are separated by a free region. In the free region, the magnetization direction can be changed by injecting current from the two adjacent pinning regions. In write operations, the WWL (Write Word-Line) transistor is turned on; the cell will be written to '1'/'0' by setting BL (Bit-Line) to high/low and SL (Source-Line) to low/high. During read operations, the RWL (Read Word-Line) transistor is turned on, then the sensing current will go through from the BL to the SL. The decoupled read and write paths can substantially reduce the read disturbance probability and improve the operation reliability of the RM cell.

A RM comprises of an array of magnetic strips, namely, racetracks. Figure 4 (b) shows the structure of a racetrack including an access port implemented by the RM cell [33]. The

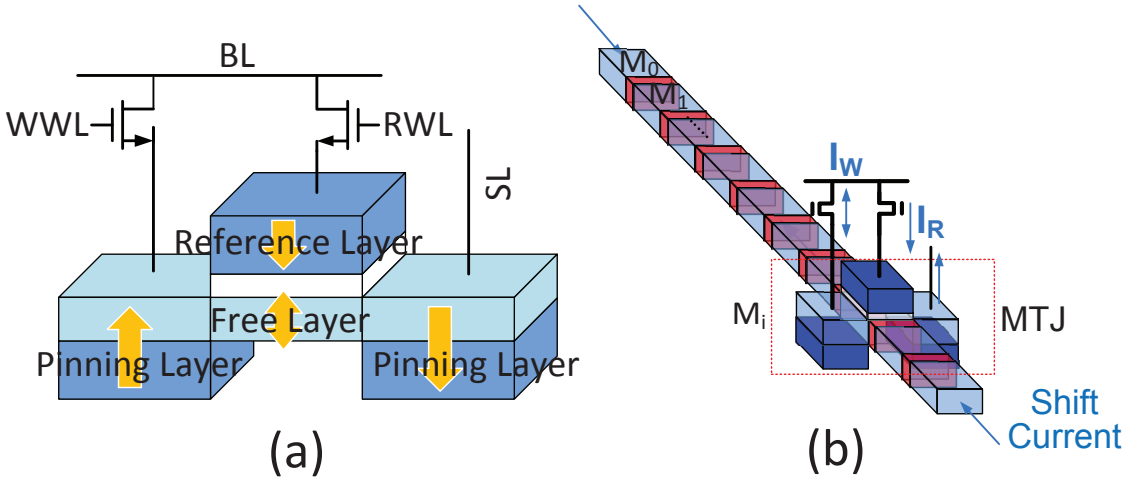

Figure 4: (a) The DWSW cell; (b) A track of RM.

racetrack is partitioned into consecutive magnetic domains which are separated by magnetic domain walls [42]. Each magnetic domain is considered as a RM cell where '0' and '1' are stored as the different magnetization directions, i.e., up and down, respectively. An important feature of RM is that the RM cell can move along the racetrack in either direction when injecting the current from different ends of the racetrack. Hence, multiple RM cells can share one access port: the RM cell to be accessed, say, M_i, must move to the slot under the access port. The ferromagnetic layer with fixed magnetization direction and the metal-oxide layer underneath, together with the M_i, constitute a magnetic tunneling junction (MTJ) structure. Depending on the relative magnetization directions in the fixed ferromagnetic layer and the M_i, i.e., parallel or anti-parallel, the whole MTJ structure demonstrates low or high resistance state, respectively.

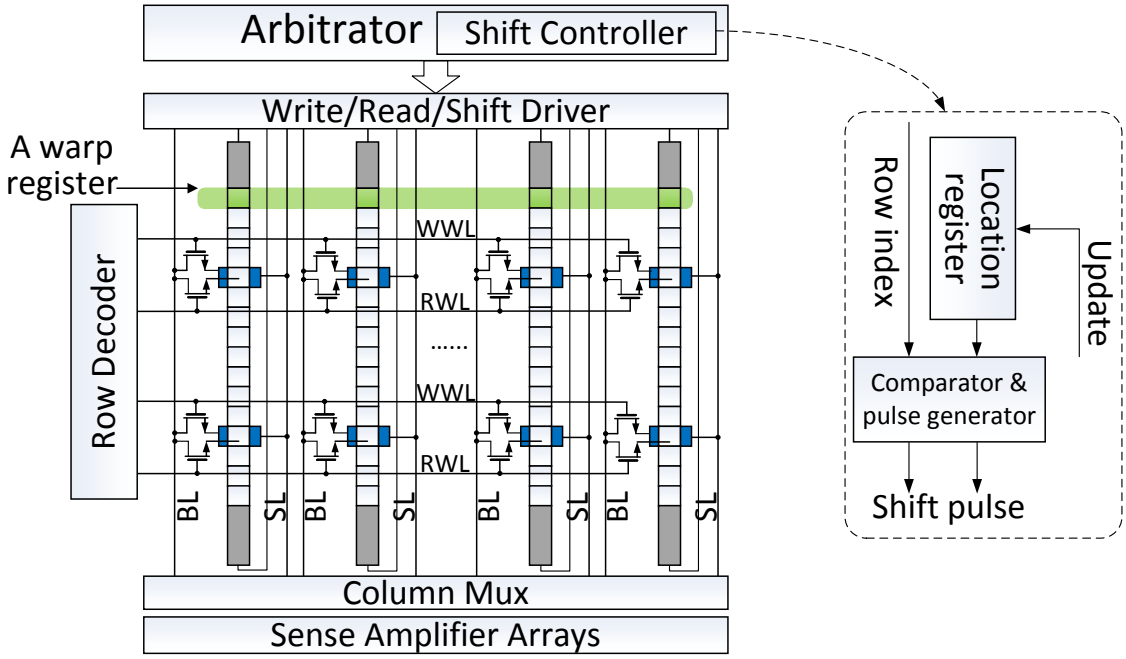

Figure 5: The schematic of a RF bank using RM. Shadow area at both ends of a track prevents domain walls from shifting out.

3.2 RF BANK DESIGN BASED ON RACETRACK MEMORY

Figure 5 shows the conceptual bank array design using RM[2]. Multiple rows share the same WL (word-line) connected to the access port, significantly reducing the number of WLs. Consequently, the design of row decoder becomes much simpler and the area is smaller than the traditional one. Note that for each access port, there are two WLs which are responsible for read and write, respectively. A shift driver supplies shift current to the head or tail of the track, depending on the location of the accessed bit on the track. Due to design complexity concern, all the tracks in an array share one shift driver and are controlled by the same command, or say, the magnetic domains on all tracks in an array move simultaneously. The magnetic domains on the same row represent a warp register.

The arbitrator associated with each bank is augmented with a shift controller that generates the shift pulse based on the incoming requests. The right of Figure 5 illustrates the inside of a shift controller. The row index generated from the row decoder is fed into the shift controller. The current location of the access port is stored in the location register. By comparing the row index and location register we can measure the distance between target register and the access port. Accordingly, the proper shift pulses are emitted from the pulse generator and drive the shifting on track.

Figure 6 indicates that the deployment of access ports is a tradeoff between area, power and read/write performance: increasing the number of access ports on a track can boost the read/write performance of the RM, however, also requires a more complex peripheral control circuit. As an extreme case, each array can have as few as only two WLs across all the tracks though the read/write performance will be dramatically degraded by the shapely increased shift delay.

Figure 6 shows the array-level design exploration of RM based RF of one Fermi SM. The parameters of each configurations are extracted from modified NVSim [43] which is enhanced with RM model. We restrict the write pulse within 1ns, and adopt the device-level parameters from [39, 33] by carefully scaling them down to 32nm technology node. Compared

[2]For simplicity, we skip the details of inner bank array organization and assume that one bank consists of one array.

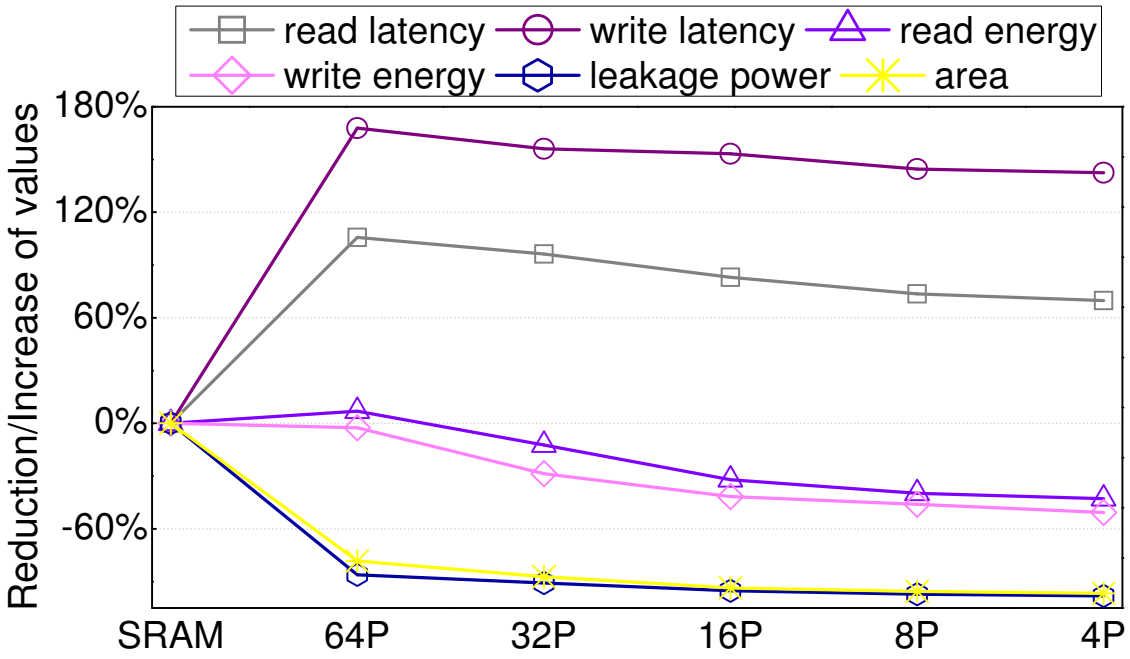

Figure 6: The design parameters of RM based GPU RF.

to SRAM design, the RF using 4-port (4P) RM reduces the area and leakage power by ∼90% and ∼93%, respectively, while the read/write energy is also saved by ∼42%/∼50% due to the smaller area, the shorter routing distance and the simpler decoding logic. However, the read and write latency are increased by 70% and 140%, respectively, due to the long sensing delay and write pulse width. Here the read and write latency is measure from the reading and writing operations on the access port, which means that shift delay is not considered. Note that the read/write latency of 4P configuration is 0.59/1.26ns that can fit into two SM cycles[3]. Therefore, the prolonged read/write latency generates very minimum impact on system performance. The result also shows that continuing to increase the number of access port on each track beyond 16 only result in marginal read/write energy saving as the power consumption of sense amplifiers and write drivers start to dominate the total power. Further reduction of read latency/energy can be achieved by using thicker oxide that can improve the TMR of the MTJ structure [40], which is beyond the scope of this dissertation. Given

[3]The SM frequency of most real GPUs is lower than 1GHz

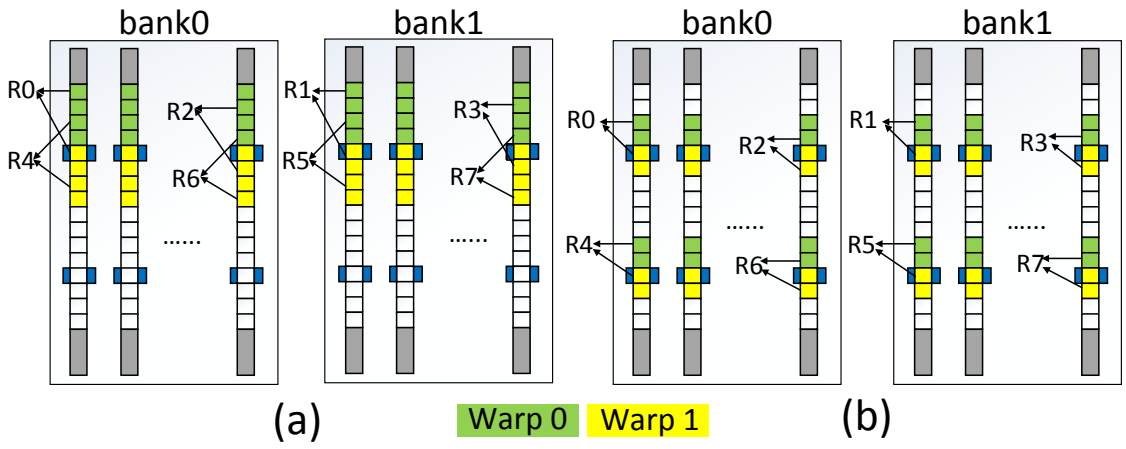

Figure 7: The (a) original register mapping and (b) register remapping.

the longest track – 64-bit – used in this work, the maximal deployment of access ports is 64. Such design omits any shifting since every magnetic domain/bit is associated with a private access port, just like RAM.

3.3 ARCHITECTURE OPTIMIZATIONS FOR RM BASED GPU REGISTER FILE

3.3.1 Register Remapping

We observe that the usage ratio of RF during GPU execution is generally low, and can be leveraged to optimize the read/write performance of RM. As we will show in Section 3.4, on average, only 62.15% RF are allocated for all simulated benchmarks because of the limited availability of the shared memory and max thread blocks supported by one SM usually make the RF under full utilization. As a result, only about 40 out of 64 entries are averagely used in a bank. Figure 7 (a) shows the original register mapping in two banks [44]. Here we assume there are 2 warps (W0&W1) running on a SM and each warp requires 8 registers (R0∼R7) interleaving in two banks. Each track has 16-bit and 2 access ports. Within a bank,

the original mapping allocates all 8 registers together of which the maximum shift distance is 8 (supposing the access sequence is *register 2 of warp 1* (W1R2)→W1R6→W0R0 and the access port on the top is selected).

Figure 7 (b) shows our proposed register remapping scheme. This scheme packs the registers around each access port to reduce the maximum shift distance down to 4. As a warp instruction often reads more than one operand, the shift delay may be reduced by accessing the registers distributed around different access ports simultaneously or consecutively. In present-day GPU design, the decoder in a RF bank has already designed to be reconfigurable [44] to fulfill the different register mapping requirements of different applications/CUDA kernels. Hence, the implementation cost incurred by register remapping is negligible. There are two preconditions in the implementations of register remapping: ① the hardware knows the register file allocation information and ② the mapping of the registers of each warp to the banks is fixed. ① can be satisfied with the help of compilation – compiler can deliver the register allocation information to the SM during execution (indeed, this scheme has already been adopted by CUDA and NVIDIA GPU series [17]). ② is promised by the thread block launching scheme: each *warp ID*, which is used to index the RF, is reusable for future warps. So once a CUDA kernel is launched, its warp register mapping is determined and such mapping will never change until the next kernel is launched.

3.3.2 RM Aware Warp Scheduling

As discussed in Section 3.2, although RW can help to reduce the RF area and leakage power, the read/write energy may not be necessarily saved if a long shift delay is required by data accesses (e.g., due to limited access port number). Hence, if we can arrange the RF accesses to be "sequential" then the shift distance between two adjacent RF accesses can be minimized and the overall shift overhead can be reduced.

Given the facts that thread switching in GPU happens very frequently, and there are normally dozens of warps pending for scheduling in a SM, we propose to re-arrange the issue order of the warps based on the distance between the access ports and the registers requested by corresponding warp instructions to generate sequential accesses to the RM based RF. We name this technique as *RM aware warp scheduling* (RMWS).

In RMWS, we define scheduling score as the maximum distance between the requested register location and the access port for the instruction of a warp. In a scheduling cycle, RMWS examines all running warps and selects the one with the lowest scheduling score to issue. There are three pieces of information are needed to obtain the scheduling score: ① the registers that the warp instruction accesses; ② the locations of the accessed registers; and ③ the current locations of the access ports.

We use Figure 7 (b) as an example to illustrate how the scheduler calculates the scheduling score of a warp instruction: assuming that an instruction of W1 accesses R2 and R4 (①). Scheduler can get ① from the instruction buffer associated with the warp. Here the location of R2/R4 (②) is 1/0 bit away from the nearest access port (③). The highest score – 1 bit, is set as the scheduling score of W1. Because we evenly distribute the access ports on each track, every access port covers the same number of bits (e.g., 8-bit in Figure 7 (b)), called a *port segment*. Note that all banks of a RF have the same port segment. We use the relative location w.r.t. port segment, namely, *bit-map location* (BML), to represent ②. A BML can be calculated by using the *register ID* and the warp ID attached to a RF request as:

$$num\_reg_{warp,bank} = \lceil num\_reg_{warp} / num\_bank \rceil, \tag{3.1}$$

$$num\_reg_{port\_segt} = \lceil num\_reg_{warp,bank} / num\_port_{track} \rceil, \tag{3.2}$$

$$BML = \lceil num\_reg_{port\_segt} \times warpID \rceil + \lfloor RegID / num\_bank \rfloor \% num\_reg_{port\_segt} + offset, \tag{3.3}$$

where num_reg_{warp} is the number of warp registers allocated to a warp; num_bank is the total register banks of a RF; num_port_{track} is the number of access ports on a track; $offset$ is the distance between the first empty bit on the top and the first valid bit in a port segment. For example, the BML of W0R0 in Figure 7 (b) is 2. The red values in Equation (3.1) – (3.3) can be pre-determined when a CUDA kernal is launched. Furthermore, all the arithmetic calculation are fix-point with 6-bit resolution[4]; Hence, the logic to generate BML has negligible hardware cost and can be easily installed in the scheduler.

[4]In Fermi a RF bank has at most 64 registers.

As aforementioned, all tracks in a RF bank move simultaneously. Hence, the BMLs of all the registers at every access port are the same. We can define the BML of a bank (i.e., bank BML) as the BML of any register at an access port. Here the bank BML represents the relative location of the access port in their assigned port segment (③). The scheduler tracks the bank BML of each RF bank: once an instruction is issued, the generated RF requests are pushed into the pending queue of an arbitrator associated with the RF bank. For a warp instruction that is scheduled to issue, we can compare the BML of the registers to be accessed by this instruction with the one of the newest pending request of the bank to obtain the scheduling score of the instruction. Like our former work [34], we do not introduce any automatic back or forth access port adjustment [45] after issuing a RF request, because we found that such adjustment has little performance or energy merit but complicates the shift controller design.

The major cost of RMWS is searching for the warp instruction which has the smallest scheduling score. This is a typical min/max search problem and the cost increases linearly when the number of concurrent warps in an SM increases [46]. Here we introduce a high efficient scheduler design which only looks for the minimum scheduling score and avoids many expensive calculations.

Figure 8 depicts an example about how the proposed scheduler works. Similar to Figure 7 (b), there are two warps to be scheduled: W0 has an instruction accessing R0 and R5, while W1 has an instruction accessing R2 and R1. At the first step of RMWS, the BML of each register access is preprocessed by:

$$BML_{pp} = 0\text{x}1 \ll (port_segt - 1) \gg BML. \tag{3.4}$$

Here both shifts are arithmetic shift. For example, the BML of W0R0 is 0x2. Through the preprocessing in step ①, we obtain the BML_{pp} of W0R0 which equals 0xE0. At step ②, the BML_{pp} of W0R0 is XORed with the BML_{pp} of *bank0* corresponding the newest pending request. At step ③, the result of step ② – 0x18, is then logically right shifted until its LSB becomes non-zero. The number of 1's in the obtained result – 0x03, called *normalized bit distance*, represents the distance between W0R0 and its assigned access port (in this case, normalized bit distance = 2). The normalized bit distances of the other register accesses

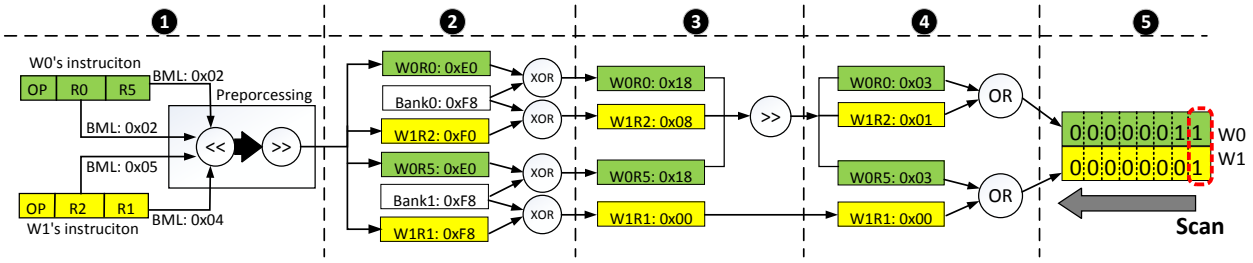

Figure 8: The scheduling procedure of RMWS.

can be calculated similarly, except for W1R1: in this case, the output of step ② is 0x0. The normalized bit distance of the W1R1 is directly set to 0x0 and nothing needs to be done at step ③. The normalized bit distances of the registers to be accessed by the same instruction are ORed together at step ④ to produce the scheduling score of the instruction. At step ⑤, the scheduling scores of all ready instructions form an array. We scan every column of the array from right to left and perform bit-wise AND of all the bits on the same column. If the result of the bit-wise AND of a column is zero, or say, there is at least one bit in that column is zero, the row whose bit is zero will have the minimum scheduling score and the corresponding warp instruction will be scheduled for issue. If the warp cannot be issued due to other constraints (we will discuss this case in Section 3.3.2.1), the scan will continue from the current column until the warp with the next minimum scheduling score is found.

The above RMWS algorithm can be easily implemented in current GPGPU design. In fact, the widely used scheduling algorithms, e.g., *loose round robin* (LRR) or *greedy-then-oldest* (GTO) [1], also adopt a scan-liked mechanism for warp selection: both of them select a warp from the warp pool and then check with the corresponding instruction buffer. If there is no ready instruction for the selected warp, they will select the next warp until a warp instruction can be issued. Hence, we expect RMWS to have a comparable timing overhead comparable to advanced GTO.

We note that RMWS overlooks one possibility that the registers accessed by a warp instruction locate in the same bank but are separated by the assigned access port at different sides. However, such a case rarely happens because in GPGPU execution, most of the warp

instructions are managed to collect their operands from different RF banks to minimize the bank conflict of register accesses. Although we are able to elaborate our RMWS algorithm to consider this scenario, e.g., using circular shift, we decide to tolerate such an inaccuracy simply because the occurrence probability of this scenario is very low: our profiling results shows that among all the executed instructions, only 1% of them have more than two registers located in the same RF bank, not even talking about if they are separated by the same access port at different sides. As we shall show in Section 3.5.1.1, such low probability of miscalculating the scheduling score does not visibly affect the efficacy of the RMWS.

3.3.2.1 Write Buffer RMWS design assumes that the service order of the pending RF requests is deterministic because of the FCFS (first-come-first-serve) policy adopted by the arbitrator. Write requests, however, may invalidate this assumption: a write request is generated from writeback stage of the pipeline but not issue stage. Hence, inserting a write request to the deterministic read request sequence will introduce uncertainty to the calculation of scheduling score. In order to remove this uncertainty, we introduce a write buffer to store the incoming write request and use *piggyback-write* policy to dispatch the write requests from write buffer to RF: the write operation is performed when the target register is moved to an access port by the previous read or write request. By doing so, RMWS only needs to consider the read requests during the calculation of scheduling score. When the write buffer is overflowed, a writeback will be performed regardless the current location of the registers being accessed in the RF bank. The overflow of the write buffer may harm the efficacy of RMWS. But as we shall show in Section 3.5.1.1, the probability of write buffer overflow is considerably low that the its impact is marginal.

In general, the RF access frequency in GPU is very high so that the lifetime of a GPU register value is short [4]. This characteristic has two positive side effects: first, the data stored in the write buffer is very likely to be accessed by a RF read request before it is written back to the RF (RAW dependency); second, a write request residing in the write buffer may be erased by a later issued instruction which also writes to the same warp register (WAW dependency). By leveraging these two dependencies, the introduction of write buffer can filter the read/write requests to the RF and further reduces the RF access energy consumption by

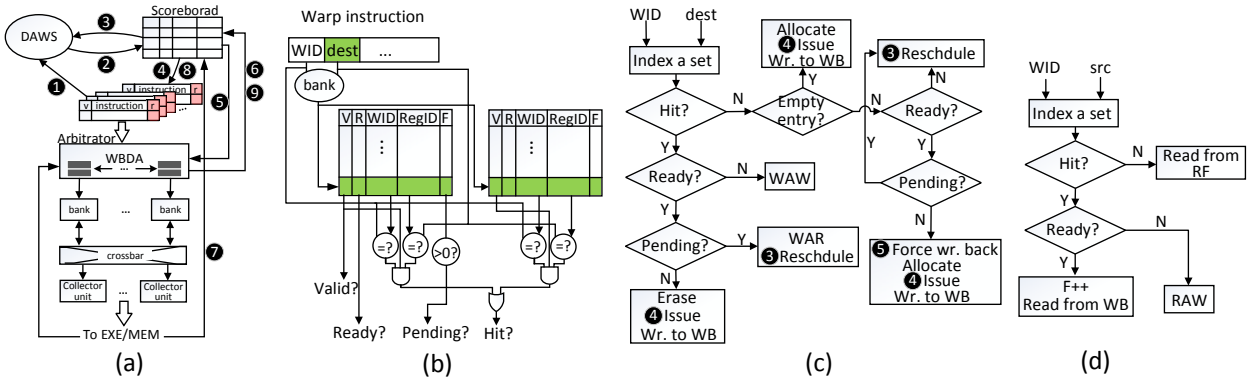

Figure 9: (a) The overview of the modified pipeline stages; (b) The WBIT by assuming each RF bank is associated with 2 WBDA entries, only the logic for logging write request is shown here; (c) The flow of logging write request; (d) The flow of logging read request.

leveraging the low access cost of the write buffer. To perform the data dependency detection between the write buffer and the issued instructions, the write buffer is divided into two parts – *write buffer data array* (WBDA) and *write buffer info table* (WBIT), which are deployed in arbitrator and scoreboard, respectively.

Figure 9 (a) shows GPU pipeline augmented with RMWS and write buffer, including the modifications on the following stages: As shown in Figure 9 (a), RMWS scheduler reads the instruction buffer (①) and then generates the scheduling score array (see Figure 8). The scheduler then picks up a ready warp instruction and updates the WBIT in the scoreboard (②). WBIT is a set-associated table tracking the utilization of WBDA; the ways of a set in WBIT equals the number of WBDA entries associated with a register bank, as shown in Figure 9 (b). Each WBIT entry has four fields – V (1 bit) indicates whether the associated WBDA entry is valid; R (1 bit) indicates whether the associated WBDA entry has received the data; *warp ID* (6-bit) and *register ID* (6-bit) record the write request; and F (up to 4-bit) is a counter recording how many in-flight read requests will read the corresponding WBDA entry so as to avoid the WAR hazards and help the scoreboard to label the relevant warp instructions as *ready* in the instruction buffer.

Figure 9 (c) and (d) depict how scoreboard processes an incoming warp instruction. The warp ID and register ID fields of an instruction are used to log the write/read request information into a WBIT entry. A warp instruction cannot be issued if: ① write buffer is full, ② WAW hazard or ③ RAW hazard happens. ① happens only when all WBDA entries are valid ($V = 1$) but none of them is ready for writing back to RF. It may be because the corresponding instructions have not produced the output data yet ($R = 0$) or the read requests to the particular WBDA entries are still in-flight ($F > 0$). ② and ③ never happen because the scoreboard can detect both hazards, preventing the unready warp instruction from being selected by scheduler. The scoreboard allows a warp instruction to issue if none of the above conditions exists (④); otherwise, the scheduler will select another warp instruction (③). Whenever a read request retrieves the data from a WBDA entry, the arbitrator notifies the scoreboard to decrease the F field of the WBIT entry associated with that WBDA entry (⑥). The write request writes the WBDA entry assigned by the scoreboard at issue stage, and notifies the scoreboard to update the R field of the corresponding WBIT entry (⑦). The scoreboard then checks the instruction buffer and labels the warp instructions as *ready* if no hazard is detected (⑧). Once the data has been written back from WBDA to RF, the arbitrator notifies the scoreboard to recycle the corresponding entry in WBIT (⑨).

If we assume there are two WBDA entries for each bank and total 16 banks in a SM, the WBIT costs 72-byte and the WBDA costs 4KB (128 bytes per entry). The incurred performance and energy overheads are very low and are considered in our evaluations.

3.3.3 Support for Multiple Warp Schedulers

Using multiple independent warp schedulers in each SM [46, 13] allows the GPU to achieve near-to-peak hardware performance as well as the reduction in the design complexity of each scheduler. However, with the introduction of RM based RF and RMWS, increasing the number of schedulers in a SM may potentially degrade the scheduling efficiency because the local optimization at each RMWS scheduler does not necessarily generate a global optimal outcome. We refer to such inefficiency caused by multiple schedulers as schedule hazard. Figure 10 gives an example illustrating the schedule hazard between two RMWS schedulers

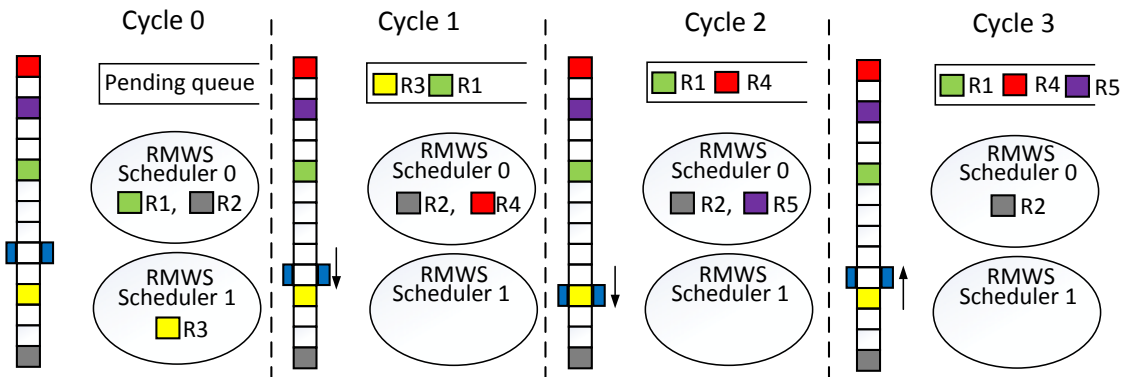

Figure 10: An example of schedule hazard between two RMWS schedulers. Register R1∼R4 are used to represent the scheduling scores/instructions of four warps. R4 is selected in cycle 2 because the bank BML is represented by the BML of R1 at that time.

of a RF bank: scheduler 0 should dispatch R2 at cycle 1 to achieve a global optimal solution by considering the track movement triggered by scheduler 1. However, it falsely picks up R1 which offers a local optimal solution for scheduler 0 only.

Schedule hazard comes from the inconsistency of the track movement directions requested by multiple simultaneously scheduled warp instructions. In order to eliminate the schedule hazard and keep a concise scheduler design, we introduce *warp-register remapping* which is shown in Figure 11. Compared to the original register remapping in Figure 7 (b), all registers of a single warp are mapped to the same bank by warp-register remapping. Each scheduler can work independently without interfering to each other.

To implement warp-register remapping, we divide the RF banks into equal-sized, non-overlap subsets, each of which is bound to a warp schedule. The warps, which are allocated to a scheduler, access all their operands from the subset of RF banks associated with that scheduler. As such each scheduler can exclusively emit RF requests to its *private-owned* RF banks and leverages RMWS to produce optimal scheduling without any interference of schedule hazard. Even the registers accessed by a warp span a reduced stretch now, the bank conflict incurred by warp-register remapping is expected as mild, since for each bank, the number of warps it serves is also decreased.

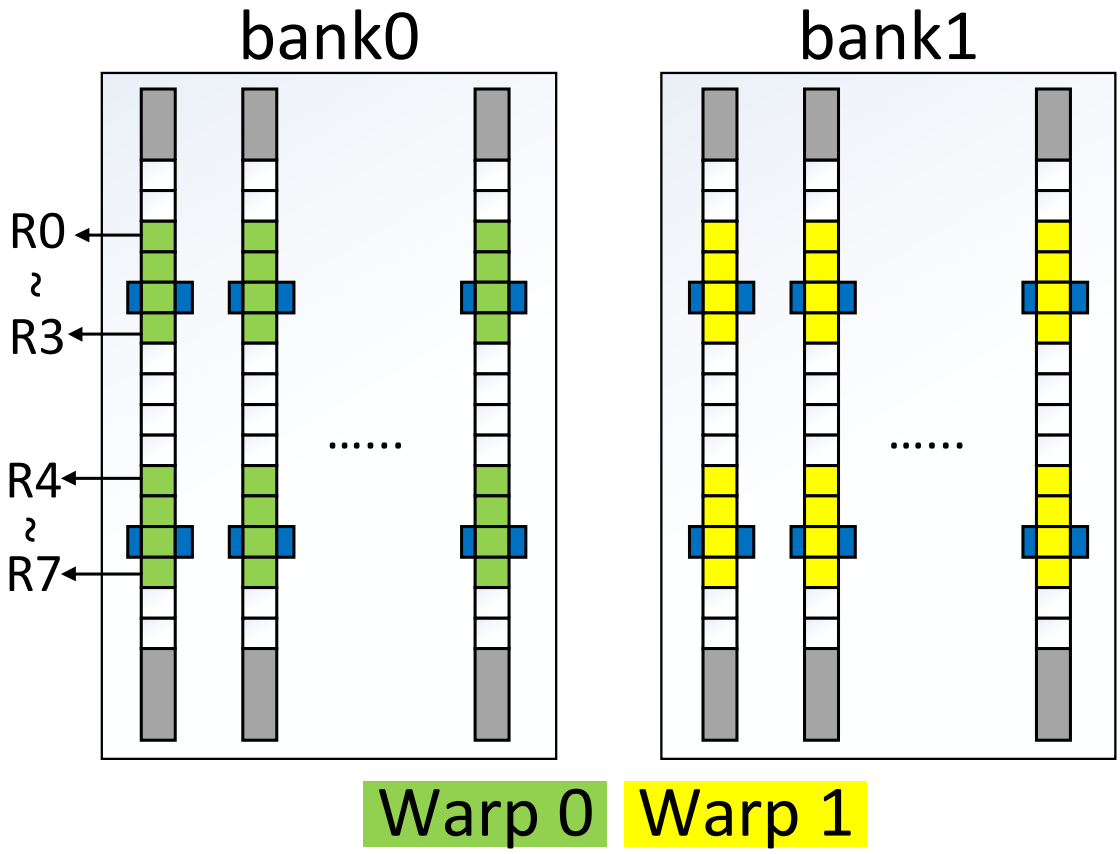

Figure 11: Warp-register remapping by assuming dual-scheduler, two warps and two banks.

3.4 EXPERIMENT METHODOLOGY

3.4.1 Applications

We construct a diverse application set from [1, 9, 10, 11] to evaluate our proposed GPGPU RF architecture. All applications are fully simulated except for `KM`, `FWT` and `RD`, which are simulated for the only first 2 billion instructions. The detailed characteristics of the applications used in this paper is summarized in Table 1.

3.4.2 Simulation platform

We use GPGPU-Sim [1] as our simulation platform, which has been modified with all the proposed architectural optimizations. Fermi-liked SM [13] is adopted and simulated by GPGPU-Sim. The simulator configuration is depicted in Table 2. To accurately simulate register remapping, we configure the GPGPU-Sim to run PTXPlus codes which exactly exposes the register allocation to hardware. We extract the performance statistics from GPGPU-Sim and also generate the detailed access statistic of RM based RF for energy consumption measurement.

We choose the widely-used GTO as the basic scheduler in our GPGPU configuration. Although there are many other scheduling algorithms [4, 47, 15, 19, 48, 49], most of them focus on improving the performance of some particular applications, e.g. memory intensive applications. Instead, RMWS is designed for more generic scenarios and can be easily incorporated into other schedulers. For example, for two-level scheduler [4], we can replace the LRR at each level with RMWS. In this work, we did not take into account the integrated solution combing RMWS and other schedulers and leave it to our future research.

The design parameters of RF implemented with different memories are depicted in Table 3; all parameters are generated by modified NVSim [43] at 32nm technology node. For the 4P RM, we adopt the device parameters from [39, 33] and the memory cell area data from [34]. The SM frequency is set to 700 MHz, which implies that the RF is running at 1400 MHz [1]. The read latency of 4P RM can fit into one cycle while the write latency is two cycles. We conservatively assume the delay of shifting one bit on a track as one cycle.

Table 1: The characteristics of GPU applications*: the number of registers allocated for each thread(# Regs/thread); the supported max warps/SM(Max warps); the usage of RF(RF usage); and the used entries/bank(Entries/bank).

Application	Abbr.	Class	# Regs/thread	Max warps	RF usage	Entries/bank
Coulombic Potential	CP	C1	15	32	49.6%	30
StoreGPU	STO	C1	48	12	56.3%	36
Matrix Q Computation	MRI-Q	C1	12	48	56.2%	36
Hotspot	HOT	C1	34	24	79.7%	52
Image Diffusion	SRAD	C1	14	48	65.6%	42
BlackScholes	BLK	C1	25	32	78.1%	50
Parallel Radix Sort	RST	C1	16	53	82.8%	53
Scalar Products	SCP	C1	20	48	93.7%	60
Sorting Networks	SN	C1	16	48	75.0%	48
Laplace Discretization	LPS	C2	17	32	53.1%	34
Neural Network	NN	C2	21	8	16.4%	13
Lattice-Boltzmann Method	LBM	C2	36	27	94.9%	62
Breadth First Search	BFS	C2	12	48	56.2%	36
Kmeans Clustering	KM	C2	10	48	46.9%	30
Path Finder	PF	C2	13	48	60.9%	39
Streamcluster	SC	C2	12	48	56.3%	36
Discrete Cosine Transform	DCT	C2	32	16	50.0%	32
Histogram	HIST	C2	16	48	75.0%	48
Vector Add	VD	C2	10	48	46.9%	30
Weather Predication	WP	C3	63	16	98.4%	63
Back Propagation	BP	C3	11	48	51.5%	33
Binomial Options	BINO	C3	20	48	93.7%	60
Fast Walsh Transform	FWT	C3	14	48	65.6%	42
Reduction	RD	C3	10	48	46.9%	30
SobolQRNG	SQG	C3	14	16	21.9%	16
Matrix Transpose	TSP	C3	10	48	46.9%	36

*For the applications containing multiple kernels, we sort the kernels by their RF usage and select the median one to represent the application.

Table 2: Simulation configuration.

Number of SMs	15
SM(core)/Interconnect/L2/DRAM clock	700/1400/700/924MHz
L1/shared memory/Constant/Texture/L2	16/48/8/2/786KB
RF size/SM	128KB
banks/RF	16
SP/SFU/MEM collectors	6/8/2
Warp size	32
Number of threads/SM	1536
Max CTAs/SM	8
scheduler	LRR, GTO, DAWS

The actual shift current density is decided by the target shift velocity [50] and the length of one magnetic domain. We use the parameters from [50] to estimate the shift energy of 4P RM while also taking the shift driver overhead into account. We also add one extra cycle to write back stage when performing the consistency in write buffer. All the above timing and energy overheads have been included in our simulations.

A register bank in a Fermi SM consists of 2048 32-bit registers; there are 64 entries (each entry has 32 32-bit registers) within a bank. In 4P RM RF design, a bank includes 1024 tracks, each of which has 64 bits. As there are 4 ports on a track, the maximum shift distance is 15-bit. The write buffer size is set to 32 and each two entries of the write buffer are associated with a bank.

3.5 RESULT

In this section, we will first evaluate the performance and energy of the GPU with one scheduler per SM, followed by the exploration on different design parameters. At last, we will present the evaluation of multi-scheduler design.

Table 3: Parameters of RF and write buffer.

Structure		Register file		Write buffer
Cell		SRAM	4P DWSW-RM	SRAM
R/W energy		103.88/102.37pJ	59.34/50.49pJ	8.34/8.09pJ
R/W latency		0.34/0.33ns	0.57/1.26ns	0.09/0.09ns
Leak. power		14.41mW	0.27mW	0.94mW
shift energy/bit		-	1.8pJ	-

3.5.1 Results of Single Scheduler Design

3.5.1.1 Performance Figure 12 shows the performances of different design choices. The baseline RF design is built with SRAM; GTO is used as the default warp scheduler. Directly deploying 4P RM based RF with GTO (4P+GTO) degrades the overall performance by 4.8% w.r.t. the GTO baseline. Register remapping reduces the average shift distance, such a performance degradation is limited within 3.4%. Combined with register remapping, the performance of RMWS is within 0.3% of the baseline. We also evaluate the performance of *register file cache* (RFC) [51]. As indicated by its name, RFC introduces a small cache for the RF, reducing the accesses to RFC. The function of write buffer in RMWS is a subset of the function of RFC; consequently, the implementation overhead of write buffer is less

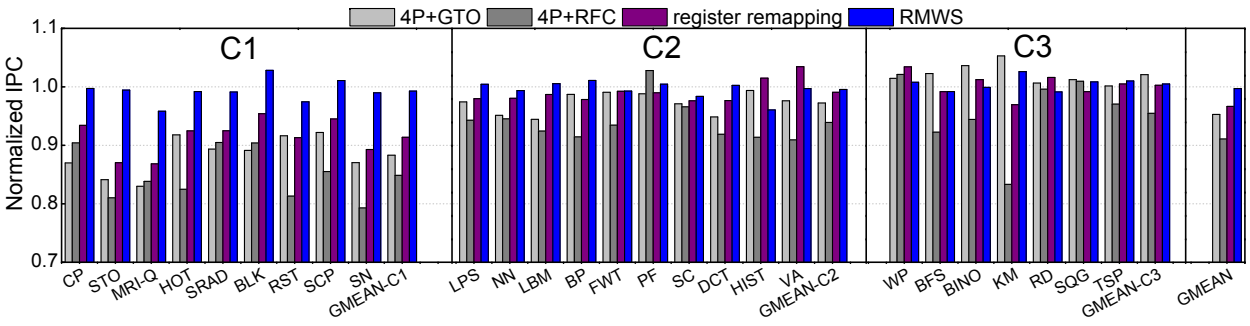

Figure 12: The performance of 4P+GTO, 4P+RFC, register remapping, RMWS.

than that of RFC. We observe 8.9% performance degradation of RFC. RFC uses *two-level scheduler* [51] which perform inferior compared to GTO, even RFC avoids a considerable amount of RF accesses.

To analyze the GPU performance in details, we divide the applications into three categories based on the sensitivity of their performance to RF access delay when GTO is employed:

- *C1* is sensitive to a prolonged RF access delay. With 4P+GTO, the *geometric mean* (GMEAN) of performance is reduced by ∼11.7%. Most applications in $C1$ are compute-intensive and exhibit extremely high data parallelism.
- *C2* is mildly sensitive to RF access delay as the included applications' performance degradation are restricted within 2.8%. Some applications in $C2$ are memory-intensive applications and their performance is limited by the available memory bandwidth; the performance of the other applications, however, is constrained by the limited innate parallelism. As a result, the performance impact of RF access delay is limited.
- The performance of the applications belonging to $C3$ is even improved slightly by 2.1% after employing RM based RF. This improvement mainly comes from some side effects introduced in the runtime behaviors such as: the RF access delay positively affects the pipeline execution, e.g., less bubble stalls in the pipeline due to less structural/data hazards; and the cache performance may be also be improved due to the different warp execution order [15, 19].

Figure 13 shows the accumulated percentage of scheduling scores of issued warp instructions. Nearly 39% of warp instructions in $C1$ are issued with a scheduling score of 3 or above; for $C2$ and $C3$, this percentage decreases down to 27%. This again explains that in 4P+GTO, why the performance variances of $C2$ and $C3$ are lower than that of $C1$. After applying RMWS, the number of the issued instructions with a scheduling score of 3 or above reduces significantly in all three categories, especially in C1. As a result, we observe a dramatic performance improvement of C1 in RMWS. For all applications with RMWS, the number of warp instructions issued with a scheduling score of 3 or above is averagely reduced by ∼13% w.r.t. 4P+GTO, demonstrating the effectiveness and applicability of RMWS.

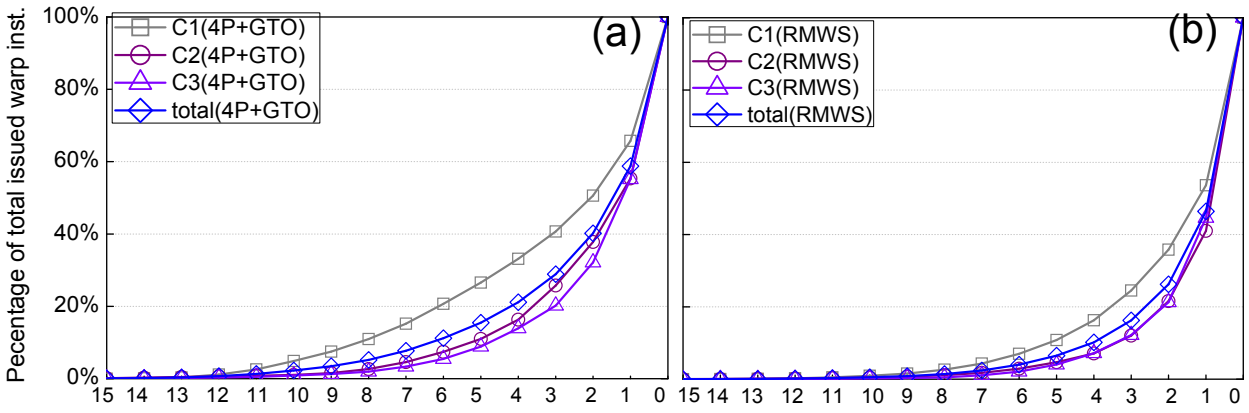

Figure 13: The accumulated percentage of scheduling score of (a) 4P+GTO and (b) RMWS.

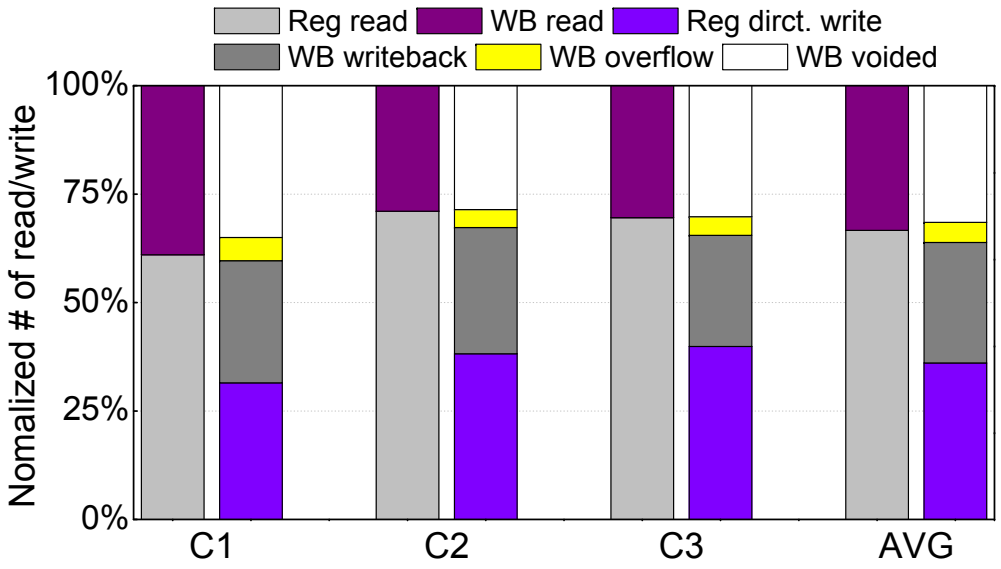

Figure 14: The RF access distribution of RMWS.

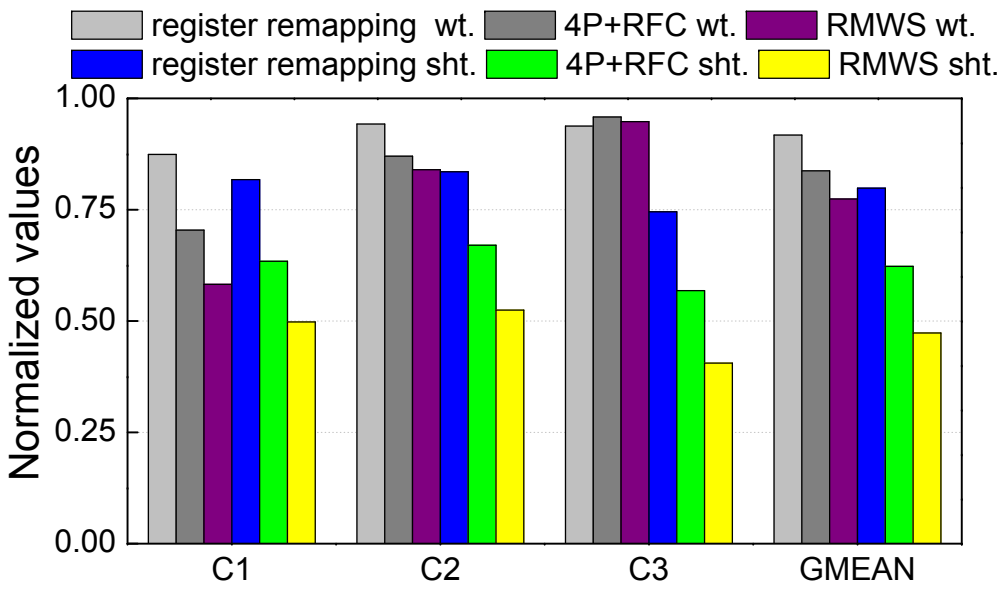

Figure 15: The waiting-cycle and shifts performed by RF requests.

Figure 14 depicts the efficacy of write buffer. By resolving the RAW and WAW dependencies, write buffer avoids ∼33.3% RF reads (WB read) and ∼31.5% writes (WB voided) in RMWS. Thanks to piggyback-write, 63.8% of total write requests to the RF are performed without any shifting: 36.1% of them can directly write to RF (Reg dirct. write) as the target register is just at the access port; the rest 27.7% are stored in write buffer and activated only when their target registers move to the access ports (WB writeback). The percentage of the RF writes triggered by the overflow of write buffer (WB overflow) is lower than 5%.

Some other factors to measure the efficacy of RMWS are the reduction in the waiting-cycle of RF requests and the average shift distance associated with RF requests, as depicted in Figure 15. Compared to 4P+GTO, the waiting-cycle of RF requests in register remapping/4P+GTO/RMWS are reduced by 8.3%/16.3%/22.6%, respectively. The reduction in waiting-cycle indicates the alleviation of shift-delay-induced RF access conflict, leading to substantial performance improvement in C1. C2 and C3, however, achieve marginal speed up because they are insensitive to the RF access delay.

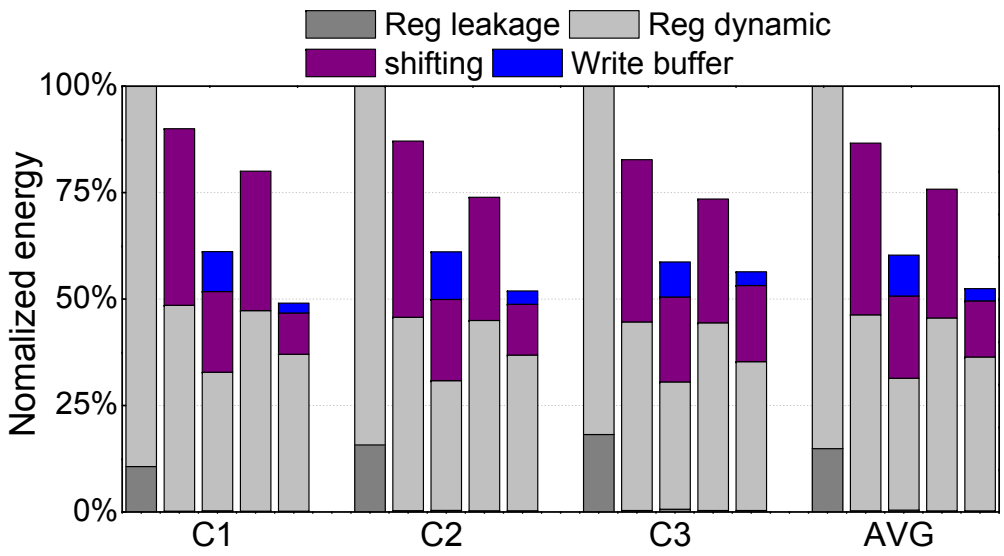

Figure 16: The RF energy consumption of GTO (1st bar), 4P+GTO (2nd bar), 4P+RFC (3rd bar), register remapping (4th bar) and RMWS (5th bar), all of which are normalized to the value of GTO.

3.5.1.2 Energy As shown in Figure 16, the introduction of RM achieves significant RF energy saving. The application of RM (4P+GTO) eliminates almost the entire leakage energy consumption. The dynamic energy directly consumed by the read and write operations of RM cells is also averagely reduced by 39.1%. The shifting on track, however, introduces 40.3% energy overhead. Total 13.4% energy is saved by 4P+GTO compared to baseline. Register remapping further saves the energy by 11.8% due to the reduction in shift energy. 4P+RFC reduces 39.7% of the RF energy consumption. The considerable energy saving of 4P+RFC is due to the RFC filtering a large amount of RF accesses. RMWS achieves 13.4% more (48.5% total) average energy saving on top of register remapping. The energy saving of RMWS mainly comes from the filtering of unnecessary RF requests by the write buffer as well as the further reduction in shift energy. As shown in Figure 15, the number of shifts performed in RMWS decreases by 52.7% compared to 4P+GTO. The energy consumption of write buffer is only about 3% of the total RF energy consumption.

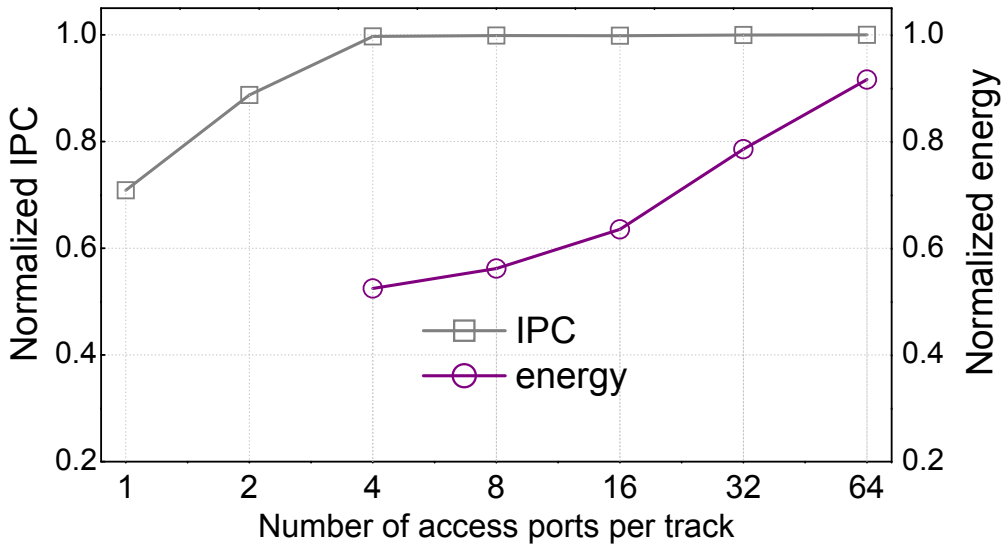

Figure 17: The performance/energy of RMWS with different access ports on a track, normalized to the performance/energy of GTO with SRAM based RF.

3.5.2 Exploration of Access Port Placement

Figure 17 shows the performance and energy exploration results of RMWS by varying the number of access ports from 1 to 64 on a track. As discussed in Section 3.2, the energy dissipation raises along with the increasing of access port followed by the decrease in shift overhead. The overall performance continues being improved as the increasing of the number of access ports. It is because the schedule decision made by RMWS gradually approaches the one of GTO when the number of access ports increases. The energy consumption sharply climbs up when the number of access ports on a track increases because of: ① The dynamic energy increases due to a more complicated design of the peripheral circuitry and the increase of the interconnect length over a larger RF area. ② More writes to the RF bank due to the improved availability of the access ports. ③ The degraded capability of write buffer to solve the RAW/WAW dependency when piggyback write is applied. It directly results in the increases of RF writes and reads. And ④ the additional leakage energy introduced by the extra access ports. Here we could not obtain the energy consumption result from NVSim

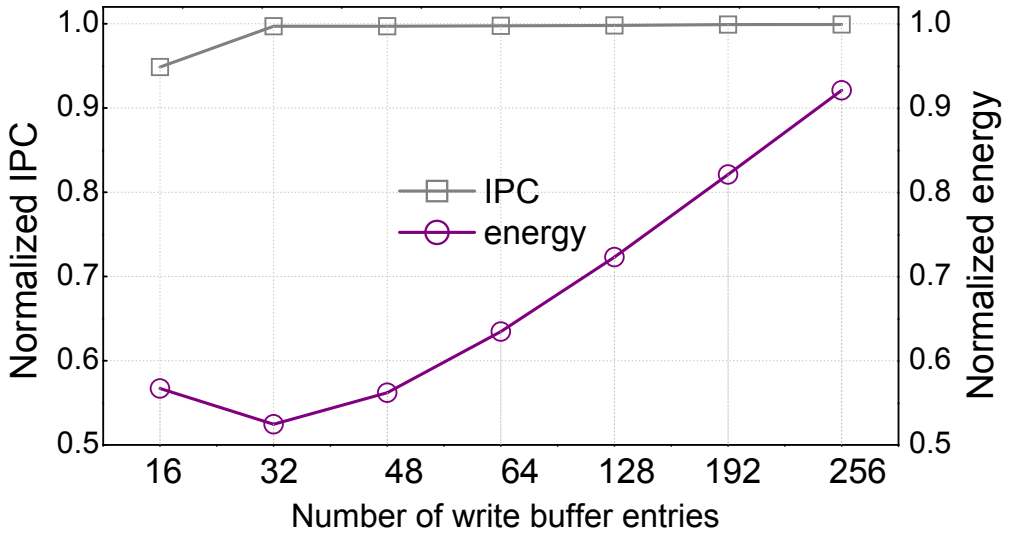

Figure 18: The impact of write buffer size on RMWS performance/energy, normalized to the performance/energy of GTO with SRAM based RF.

of the 1P and 2P designs because no practical layout design can be found for such irregular array structures. Figure 17 shows that 4P RM design achieves the highest performance as well as has the lowest energy consumption among all the options.

3.5.3 Exploration of Write Buffer Size

Figure 18 depicts the variations of the normalized performance and energy of RMWS over different write buffer sizes. A write buffer with mere 16 entries, i.e., one WBDA entry per bank, can achieve 95% of the baseline performance. Increasing the number of entries in write buffer to 32 can approach over 99% of baseline. Continuing to increase the number of entries in the write buffer, however, only gives us very marginal performance enhancement. As expected, increasing the write buffer size generates some energy consumption overhead. For example, the energy consumption of RF designs doubles when the number of entries in the write buffer increases from 32 to 256. We note that the energy consumption of 32-entry write buffer design is lower than that of 16 entry design because the 32-energy design filters more RF access requests and encounters fewer write buffer overflows. Hence, the 32-entry write buffer design achieves the best trade-off between performance and energy consumption.

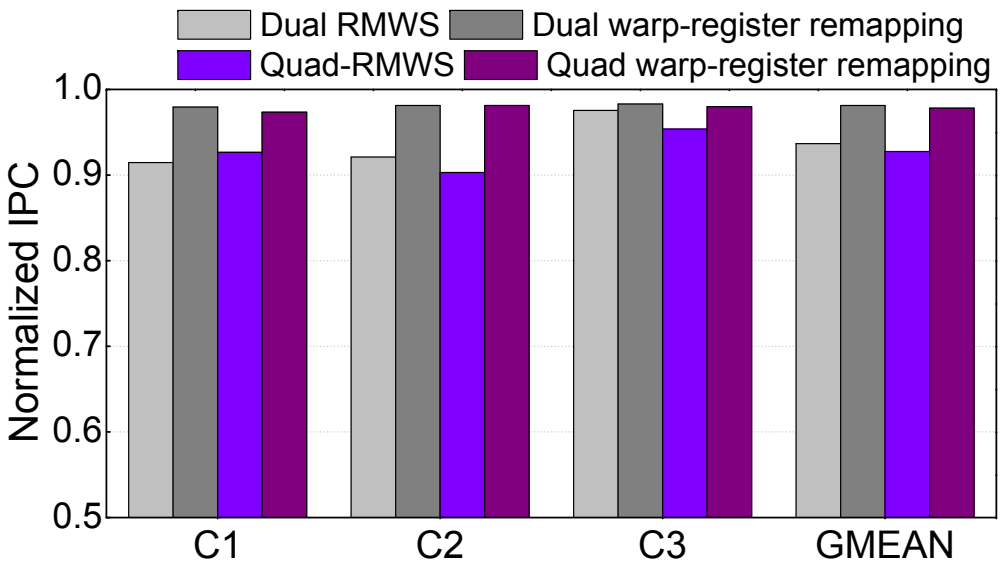

Figure 19: The performance of different multi-scheduler designs.

3.5.4 Results of Multi-scheduler

3.5.4.1 Performance Figure 19 shows the performance of different multi-scheduler designs. Dual-RMWS design decreases the performance of 6.4% compared to Dual-GTO across three types of applications; Dual-GTO is with SRAM based RF. As previously discussed, the schedule hazard among multiple RMWS schedulers harms the potential performance. After combining Dual-RMWS and warp-register remapping, the GPGPU performance is improved by 4.7% over Dual-RMWS, within 1.9% of the performance of Dual-GTO. It implies that warp-register remapping effectively suppresses the schedule hazard and reduces the number of shifts performed by RF requests. Intuitively, as more RMWS schedules are introduced in a SM, the possibility of scheduler hazard happening claims up. The Quad-RMWS further downgrades the performance by 7.2% w.r.t. Quad-GTO. Again, warp-register remapping salvages 5.1% performance on top of Quad-RMWS, within 97.8% of the performance of Quad-GTO.

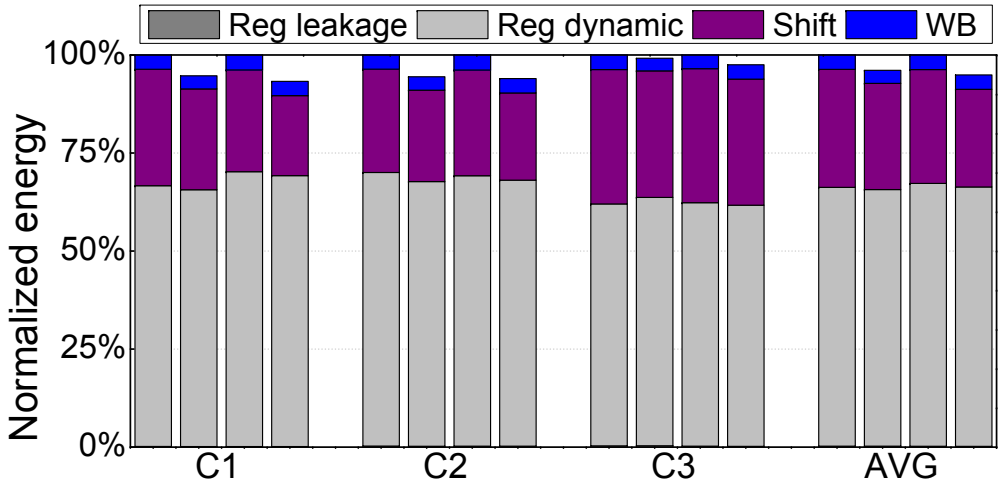

Figure 20: The energy consumption of dual-RMWS (1^{st} bar), warp-register remapping with dual-RMWS (2^{nd} bar), quad-RMWS (3^{rd} bar), and warp-register remapping with quad-RMWS(4^{th} bar).

3.5.4.2 Energy Figure 20 shows the energy consumption of different multi-scheduler designs. Compared to dual-RMWS, the energy consumption of RF with warp-register remapping is saved by 3.9% mainly due to the saving of shifting energy. warp-register remapping allows global optimized schedule decision made by individual RMWSs during every schedule cycle, suppressing the shift operations of RM based RF. For the quad-schedule case, warp-register remapping reduces the energy consumption by 5.1% compared to quad-RMWS.

3.6 SUMMARY

In this section we propose a RM based GPU RF design which can achieve significantly improve energy and area efficiency of GPU RF compared to the conventional SRAM design. In order to overcome the negative performance impact induced by the inherent sequential access of RM, we also propose to dynamically recognize the register mapping to reduce the shift delay in RM accesses. An efficient RM aware warp scheduling (RMWS) scheme,

including a newly introduced write buffer is designed to hide the long RF access latency. A warp-register reorganization scheme is developed to eliminate the schedule hazard among multiple schedulers. After combining all the proposed technologies, we can achieve more than 48% RF energy saving w.r.t. the SRAM based RF design while keeping the performance stale even with the introduction of long RF access delay.

4.0 VERSATILE WARP SCHEDULER

In this chapter we introduce VWS which can explore diverse L1 data cache localities of GPU application. First we analyze the various cache localities in GPU application. Then we present the detailed design of VWS, followed by the our experimental setup and the experimental results. At the end of this chapter we will summarize VWS.

4.1 ANALYSIS OF CACHE LOCALITY IN GPU

In GPU architecture, there are various causes for cache misses. The competition for cache among the threads inside a warp could cause cache misses, which are known as intra-warp misses. Different warps inside a thread block or *cooperative thread array* (CTA) competing for cache usage induce intra-CTA misses. Different CTAs dispatched to the same SM may also cause inter-CTA cache misses. In this subsection, we will analyze the contribution of each type of misses to the overall L1 data cache misses.

We conducted experiments with 14 representative GPU applications. Figure 21 shows the breakdown of L1 data cache accesses for these benchmarks. The experiments showed that different applications have different sensitiveness to cache misses. Based on the observation, we categorize the applications into two types by *access per kilo instructions* (APKI) of L1 data cache: 1) C1 is highly cache-sensitive with a APKI above 10 (Figure 21(a)); 2) *C2* is mildly cache-sensitive or cache-insensitive with a APKI below 10 (Figure 21(b)).

Our results show that intra-warp and inter-CTA misses dominate the cache misses in C1. Intra-warp misses dominate 4 out of 7 applications in C1. Inter-CTA misses contribute the most cache misses in `MM`, `BFS` and `SS`, successfully harvesting inter-CTA locality in those

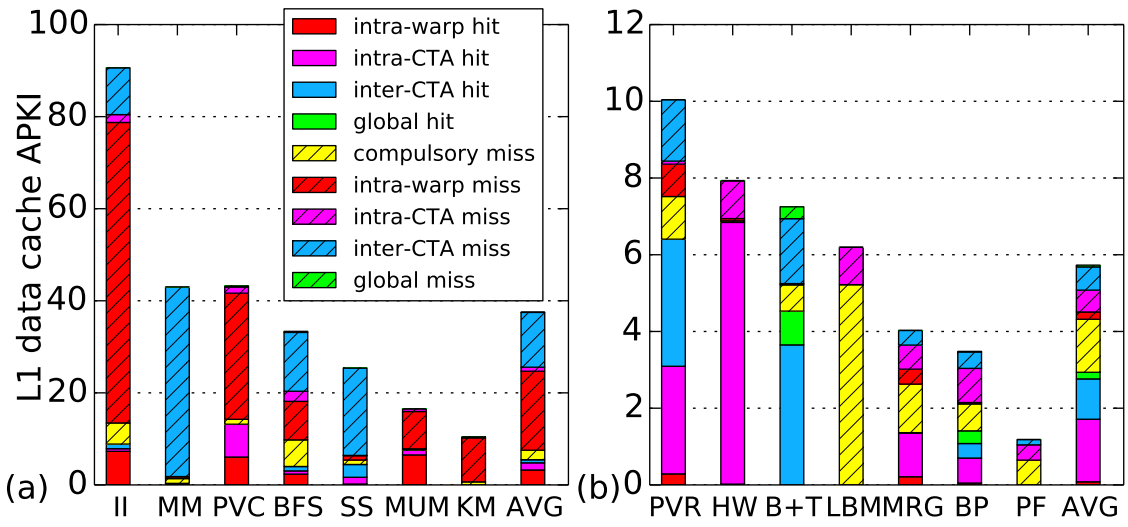

Figure 21: The L1 data cache APKI of (a) C1 and (b) C2 applications. The details of applications and the simulation platform are given in Section 4.3.

applications is critical for the system throughput. Intra-CTA access, which reflects the inter-warp data sharing within a CTA, generates considerable cache accesses in `PVC`. Intra-CTA locality also commonly exists in C2. Preserving such type of locality in the warp scheduler becomes necessary. Although the performance of the applications in C2 is not expected to be highly sensitive to the usage of L1 data cache, some applications are mildly cache-sensitive (e.g., `PVR`, `HW` and `B+T`).

From Figure 21, one can clearly see that in both C1 and C2, there are always more than one type of cache misses contributing to a significant portions of the total misses.

4.2 VERSATILE WARP SCHEDULING

VWS is enable to explore diverse cache localities of GPU applications by locality-aware CTA dispatching and warp scheduling. VWS consists of three main components: IWL, IntraCL, and InterCL. When a kernel is offloaded for execution, **InterCL** dispatches CTAs into a

SMs by considering the inter-CTA locality. In a SM, VWS samples the warp execution and estimates the per-warp working set. According to the estimated result, **IntraCL** try to maximize the inter- and intra-CTA locality when it decides which warps to be kept active. In the meantime, by keeping a target number of active warps, **IWL** preserves the intra-warp locality in each SM.

The combination of these three components leads to a complete VSW scheme that explores the intra-warp, intra-CTA and inter-CTA cache locality. The details of them will be presented in Section 4.2.1, 4.2.2 and 4.2.3, respectively.

4.2.1 Intra-Warp Locality Aware Scheduling

Intra-warp locality aware scheduling (IWL) is designed to find a number of warps the total working set of which can be fitted into the L1 data cache of a SM.

GPU programming model guarantees that the execution of each warp in a kernel proceeds with almost the same phases, implying that the per-warp working set of the kernel can be predicted by sampling a small subset of all the warps. Thus, we divide the kernel execution into two periods: *sampling* and *post-sampling*. Once a kernel launches, it first enters the sampling stage where IWL keeps the active warps as many as possible to estimate the per-warp working set. According to the estimation result, IWL starts throttling warps at the beginning of the post-sampling stage. During the post-sampling stage, IWL continues estimating the per-warp working set by following the same way in sampling stage. Every time the execution of a warp completes, IWL calculates and updates the estimation of the per-warp working set. The estimated per-warp working set is updated in each SM throughout the lifetime of a kernel as:

$$WS_i = \frac{1}{2}WS_{i-1} + \frac{1}{2}WS_{new}. \tag{4.1}$$

Here WS_{i-1} is the estimated working set based on the first $i-1$ warps; WS_{new} is the estimated working set of the i^{th} warp that just finishes. Note that here Equation (4.1) gives higher weight to the contribution of the recent finished warp to reflect the temporal trend of kernel execution. The number of warps monitored in the sampling stage in a SM (N_{pred}) is empirically set to

$$N_{pred} = \lceil \alpha \frac{N_{CTA} * Size_{CTA}}{Size_{warp} * N_{SM}} \rceil, 0 \leq \alpha \leq 1. \tag{4.2}$$

Here N_{CTA} is the number of CTAs in a kernel[1]. $Size_{CTA}$ and $Size_{warp}$ are the numbers of threads in a CTA and a warp, respectively. N_{SM} is the number of SM in a GPU. α is a constant that determines the length of the sampling stage, i.e., the sampling stage ends when α of total warps finish their executions in a SM.

In order to obtain WS_{new}, we need to record the number of cache blocks that a warp touches during its execution. Instead of directly recording the cache block accesses, in IWL implementation, we propose to use a counter to monitor the L1 data cache misses generated by a warp to approximately derive the WS_{new}: If a warp triggers a miss on L1 data cache, its miss counter will increase by 1. Multiple misses on the same cache block, however, can inflate the real number of cache blocks that a warp touches. For example, a warp misses on a cache block that gets evicted later; a second miss on that cache block can happen if it is revisited by the same warp or other warps after eviction. To avoid the duplicated counting of cache misses, each SM is augmented with a binary table – *address miss table* (AMT), each entry of which indicates whether there is already a cache miss happens to the associated cache block. Once a miss on L1 data cache occurs, the miss address is used to index the AMT. If the AMT entry is unset previously, the warp indexing the AMT increases one on its miss counter and set the AMT entry; otherwise the warp does nothing for the miss counter or AMT entry. By the end of the warp execution, WS_{new} is set to the value of the miss counter and used to calculate WS_i. Based on the estimated per-warp working set, the best number of warps kept active in a SM can be calculated by:

$$N_{act} = \lceil \frac{Size_{L1D}}{WS_i} \rceil, \tag{4.3}$$

where $Size_{L1D}$ is the number of total cache blocks in the L1 data cache.

Equation (4.3) roughly takes the distinct cache blocks demanded by a warp as the per-warp working set. In some kernels, the level of cache block re-reference can be very low. One extreme case is when a warp streamlines its large working set that can occupy the whole L1 data cache without any re-reference of cache block. So IWL possibly keeps only

[1] N_{CTA} is actually not available until a kernel finishes its execution because CTAs are dynamically dispatched to SMs. Here we estimate it before the execution by using total number of CTAs of a grid divided by the number of SMs.

one warp active, which sacrifices the thread-level parallelism without any payback of locality enhancement. To solve this problem, we introduce *cache re-reference degree* of a warp as:

$$RRDegr_i = \frac{1}{2}RRDegr_{i-1} + \frac{1}{2}RRDegr_{new}, \tag{4.4}$$

where

$$RRDegr_{new} = \frac{N_{L1D\_acc}}{WS_{new}}. \tag{4.5}$$

Here N_{L1D_acc} is the number of L1 data cache accesses generated by a warp during its execution. Similar to Equation (4.1), Equation (4.4) gives higher weight to the contribution of the newest cache re-reference degree. Equation (4.3) is then modified by considering the cache re-reference degree as:

$$N_{act} = \lceil \beta \frac{Size_{L1D}}{WS_i * RRDegr_i} \rceil, \beta > 1, \tag{4.6}$$

where β is an empirical coefficient that controls the aggressiveness of IWL in making the throttling decision on warps.

Implementation Overhead: The major overhead of IWL is the calculations of WS_i and $RRDegr_i$. Each running warp is associated with two 10-bit counters – one to monitor the number of cache misses and the other to count total accesses to the L1 data cache. These counters consumes negligible hardware overhead since only dozens of warps (*e.g.*, 48 in Fermi [13]) can concurrently run on a SM. We found that an AMT of 8192 entries, or total 1KB is sufficient to identify the overlap of the cache misses on the same cache block.

As a comparison, *cache conscious wavefront scheduler* (CCWS) requires another tag cache that implements all the functions of a normal L1 data cache. Furthermore, CCWS needs to sort-then-sum all the running warps based on their lost locality scores in every scheduling cycle. IWL brings in substantially smaller hardware and operation costs than CCWS.

4.2.2 Intra-CTA Locality Aware Scheduling

IWL only decides *how many* warps that need to be activated in a SM. To preserve intra-CTA locality, we also need to properly identify *which* warps that shall be activated in a SM.

An illustrative example is given in Figure 22, which has two CTAs – CTA0 and CTA1 – running on a SM. Each CTA contains two warps – W0 and W1. We assume both warps in a CTA share two memory segments (M0 and M1) with the same capacity and L1 data cache can simultaneously hold at most two memory segments. Following the algorithm in IWL, no more than two warps will be activated in a SM. Figure 22(a) shows two warps activated from different CTAs periodically compete the L1 data cache. Consequently, the memory segments of two warps are frequently evicted and re-visited, thrashing the L1 data cache. Because of unawareness of intra-CTA locality, the intra-warp locality is destroyed and the effort paid by IWL is offset.

To simultaneously preserve the intra-warp locality and maximize the intra-CTA locality, a simple yet effective scheme, *intra-CTA locality aware scheduling* (IntraCL) is proposed. Once IWL determines the number of the warps to be activated (i.e.,N_{act}), IntraCL will select N_{act} warps from as few CTAs as possible. In GPUs, every SM maintains an array of warp slots. A CTA is assigned to a set of continuous warp slots when it is dispatched to a SM. Therefore, InterCL can select the warps residing in the continuous warp slots to minimize the total active CTAs. Figure 22(b) shows the effect after applying IntraCL on top of IWL: two warps from the same CTA are activated so that the total working set of both warps can be fitted into the L1 data cache; the thrashing on L1 data cache is eliminated.

4.2.3 Inter-CTA Locality Aware Scheduling

Inter-CTA locality is another important type of locality that needs to be considered in our scheme. Unlike the global locality defined as the data sharing among all the CTAs of a grid, here inter-CTA locality denotes the data sharing among a subset CTAs of a grid. We observe two types of inter-CTA data sharing by running application profiling: cache blocks are shared by 1) multiple consecutive CTAs or 2) multiple CTAs with random sequence.

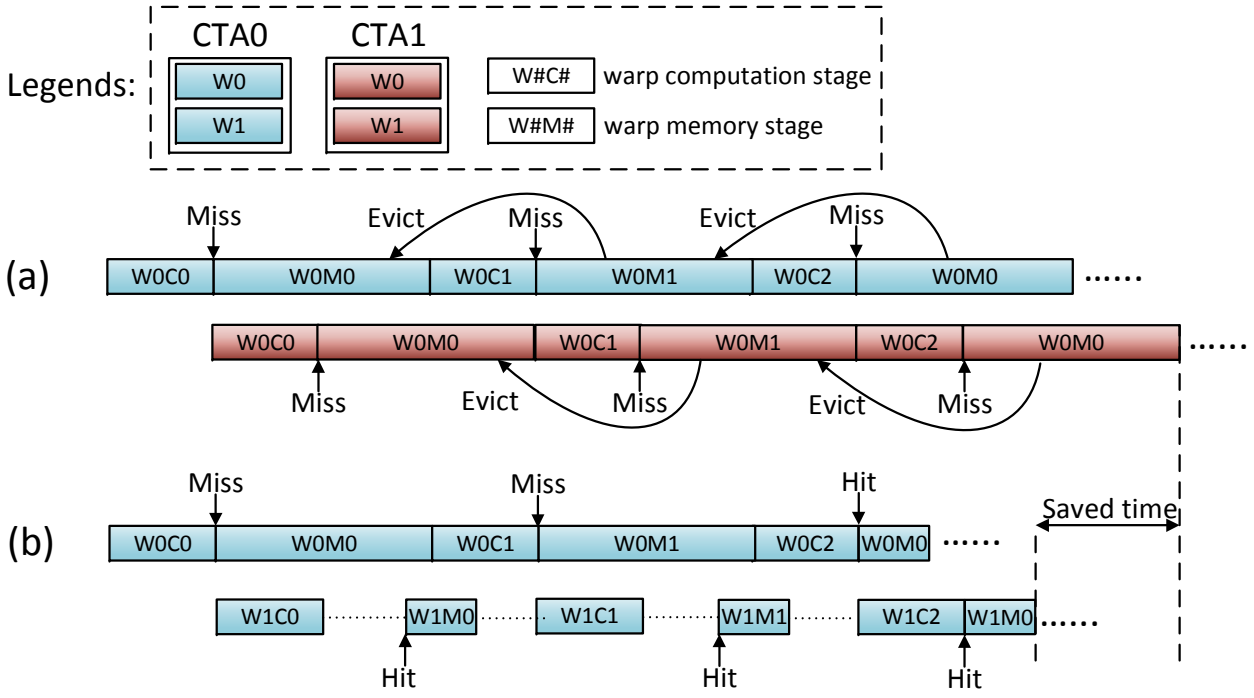

Figure 22: The warp execution (a) before and (b) after applying IntraCL.

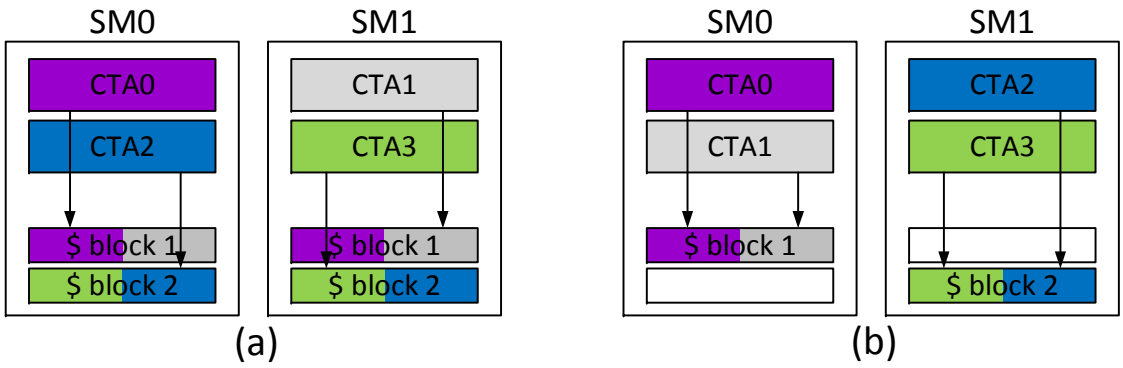

Figure 23: The execution (a) before and (b) after applying InterCL.

The first type of inter-CTA data sharing can cause inefficient usage of L1 data cache under traditional CTA dispatching mechanism in GPU. Figure 23(a) shows four CTAs are interleaved between two SMs, and each two consecutive CTAs share one cache block. From the figure we can see that the contents of the L1 data cache in SM1 is a complete mirror of that in SM0 and half of the L1 data cache capacity is wasted. Moveover, such a CTA dispatching mechanism also increases the compulsory misses and reduces the inter-CTA hits on the two L1 data caches. If the CTAs are serially dispatched to the SMs, four compulsory misses in Figure 22(a) become two compulsory misses and two inter-CTA hits in Figure 22(b). The required cache capacity is accordingly reduced by half, as shown in Figure 23(b).

The second type of inter-CTA data sharing in contrast is purely random and the cache blocks shared by the CTAs are mainly determined by the application inputs. For example, during GPU executions, every CTA is assigned with an independent query in *B+ tree* (B+T). All the threads in a CTA simultaneously search for the key value associated with the query [52]. Since the key value is randomly assigned to each CTA, the cache blocks shared by the CTAs are random. So it is very difficult to capture such inter-CTA locality.

Based on the above observations, we propose *inter-CTA locality aware scheduling* (InterCL) to explore the first type of inter-CTA data sharing. InterCL consists of two parts. The first part is to ensure consecutive CTAs are dispatched into every SM. Each SM is augmented with a dispatch queue to hold the pending CTAs. When a kernel is launched,

all the CTAs are serially assigned to the dispatch queues of the SMs. The second part of InterCL is a two-level scheduler [4]. The first level includes certain number of warps which are ready for scheduling. These warps may come from one or multiple consecutive CTAs. The second level includes the warps from the rest CTAs. When a warp from the first level is suspended due to pipeline stall, it will be demoted to the second level. At the same time, InterCL promotes a warp from the second level to the first level by following such a priority order (from high to low): the CTA that the warp belongs to is 1) currently retained in the first level (e.g., the warp is originally demoted from the first level); 2) the precursor or 3) the successor of the CTA(s) whose warps are in the first level. This policy ensures the warps in the first level always belong to the consecutive CTAs so as to increase the hit rate of the inter-CTA shared cache blocks.

The dispatch queue of each SM can be implemented *virtually* in runtime which is responsible to generate the thread blocks. Thus, the hardware cost of InterCL is neglectable. The combination of IWL, IntraCL and InterCL forms a complete VWS scheme.

4.3 EXPERIMENT METHODOLOGY

GPU simulation is also performed on a modified GPGPU-Sim [1] augmented with the specific warps schedulers. Table 4 summarizes the configuration of the simulator. Fermi-liked architecture [13] is adopted and simulated by GPGPU-Sim.

A combined GPU application set from [10, 19, 9] is used in our evaluations. Most of the applications are fully simulated except the applications from MapReduce [53], of which only the first two billion instructions are simulated. Table 5 lists all the applications evaluated in this work.

The configuration with *loose round-robin* (LRR) is used as the baseline. Besides VWS, we also evaluated CCWS for comparison purpose. The configuration of CCWS used in the experiments is the same as [15] except the maximum number of warps supported by a SM is increased to 48. The configuration details of VWS include: 1) The default value of α in Equation (4.2) is set to 0.1; 2) AMT is periodically cleared after every N_{pred} (Equation (4.2))

Table 4: Simulation configuration.

Number of SMs	15
SM(core) Clock	700MHz
SIMD width	16
L1 Data Cache	32KB, 128B line, 4-way, LRU
Warp Size	32
Max Number of Threads	1536/SM
Max Thread Blocks	8/SM
Scheduler	LRR, CCWS, VWS
# of Memory Channels	6
L2 Cache	128B line, 8-way associated, 128KB/channel, total 768KB
Memory Controller (MC)	6 MCs, FR-FCFS, open-page, 16-entry request queue/MC
Interconnection	2D Mesh
DRAM	GDDR5, 924MHz, 177.4GB/s

Table 5: GPU applications.

Cache-sensitive (C1)			
Description	Abbr.	Description	Abbr.
Inverted Index	II	Matrix Multiply	MM
Page View Count	PVC	Breath-First Search	BFS
Similarity Score	SS	MUMmerGPU	MUM
Kmeans Clustering	KM		
Mild or not cache-sensitive (C2)			
PageViewRank	PVR	Heartwall	HW
B+ Tree	B+T	Lattice-Boltzmann Method	LBM
MRI-Gridding	MRIG	Back Propagation	BP
Pathfinder	PF		

warps finished the execution; 3) The default value of β in Equation (4.6) is set to 4; 4) At most 16 warps are kept in the first level of InterCL and scheduled in greedy-than-oldest [15] fashion.

4.4 PERFORMANCE EVALUATION

Figure 24 compares the IPC of the GPU for different benchmarks under different warp schedulers. All IPCs are normalized to LRR. Figure 25 depicts the detailed analysis of the L1 data cache access.

For the cache-sensitive applications in C1, CCWS achieves 53.9% geometric mean performance improvement compared to LRR, which is consistent with the results in [15]. From Figure 25 we can see that, the performance speedup of C1 achieved by CCWS mainly comes from the improved intra-warp locality: the intra-warp misses of the L1 data cache are significantly reduced by 66% compared to that in LRR. On the other hand, the performance

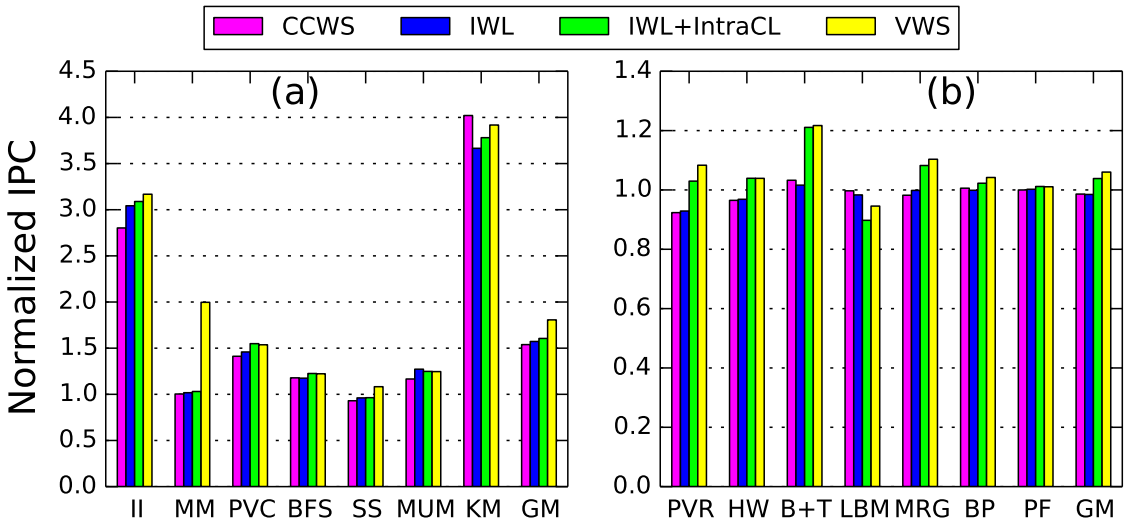

Figure 24: The performance of (a) C1 and (b) C2 with CCWS, IWL, IWL+IntraCL and VWS, normalized to LRR's performance.

of C2 slightly degrades by 1.4% under CCWS. This is because CCWS does not consider the intra-CTA locality during its warps scheduling. The intra-CTA misses increase noticeably in `PVR`, `HW` and `MRIG` when CCWS is applied, causing the performance degradation in C2.

Like CCWS, IWL alone also boosts the system performance of C1 by 57.3% (3.4% higher than CCWS). IWL estimates the per-warp working set with higher accuracy as well as much less hardware overhead, efficiently preserving intra-warp locality. However, IWL does not help on preserving the intra-CTA locality, resulting in an on average 1.6% performance degradation in C2.

The combination of IWL and IntraCL (IWL+IntraCL) improves the performance of C1 by 60.5% w.r.t. the LRR baseline. Compared to only IWL, the addtional 3.2% performance improvement comes from the exploration of intra-CTA locality. More important, the successful exploration of intra-CTA locality in IWL+IntraCL achieves 3.8% average performance improvement in C2.

The integration of IWL, IntraCL and InterCL, i.e., VWS, results in 80.7% and 6.0% system performance improvements in C1 and C2, respectively. Compared to LRR, VWS

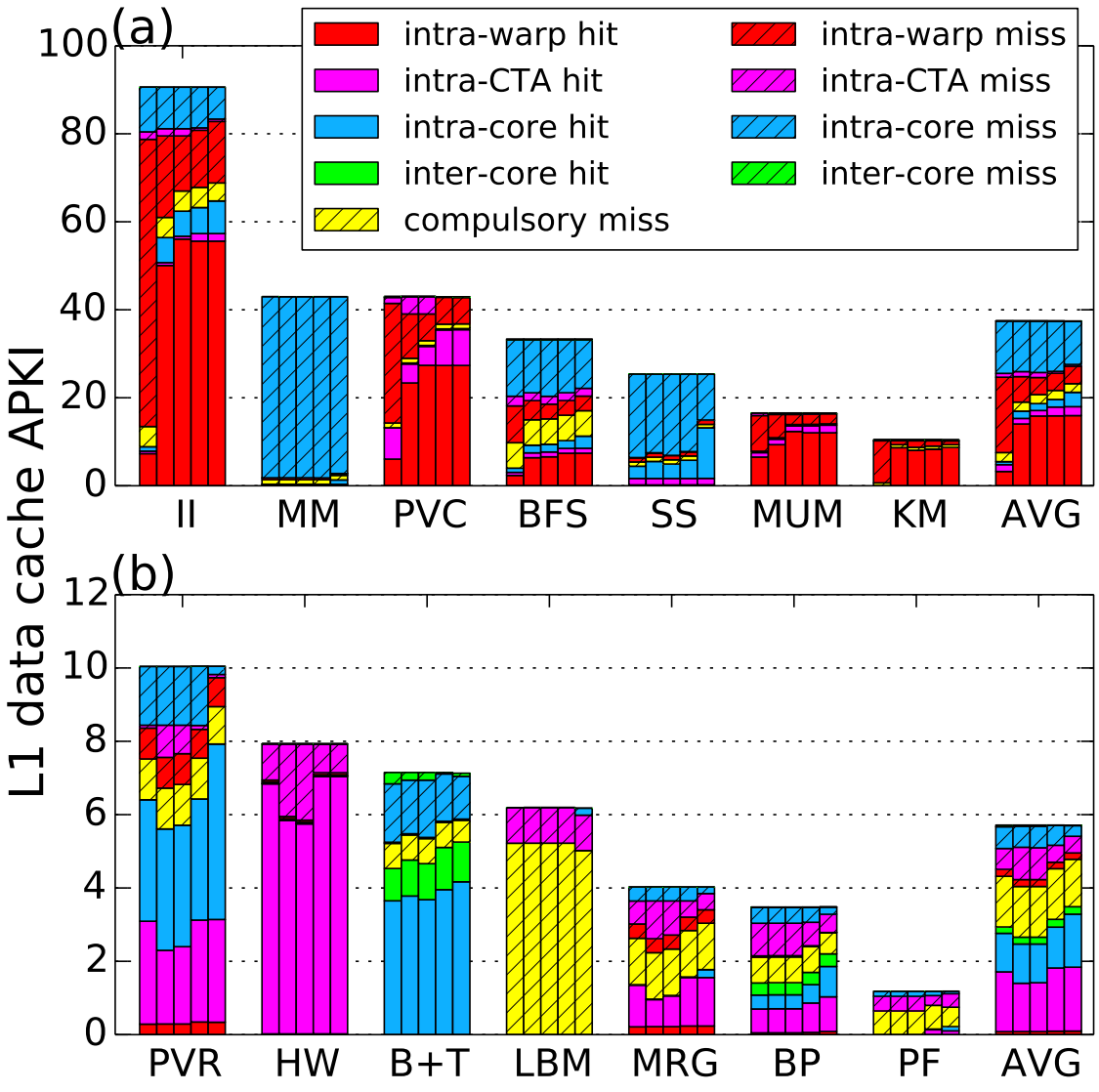

Figure 25: The L1 data cache AKPI of (a) C1 and (b) C2 with LRR (1^{st} bar), CCWS (2^{nd} bar), IWL (3^{rd} bar), IWL+IntraCL (4^{th} bar) and VWS (5^{th} bar).

improves the GPU performance by 38.4% over total 14 applications and outperforms CCWS by 9.3%. Because of InterCL, the CTAs that share cache blocks are dispatched to the same SM so that the inter-CTA misses across the SM boundaries are reduced substantially. Our detailed analysis found that most of inter-CTA shared cache blocks in `MM`, `SS` and `PVR` are shared by consecutive CTAs while those of `II`, `BFS` and `B+T` are shared by random CTAs. Thus, InterCL is anticipated to perform better in the former three applications than the latter three.

There are two more interesting abstractions that need to be pointed out here:

- `MM` enjoys an astonishing speedup (2×) when InterCL is employed without obvious reduction in inter-CTA misses on L1 data cache. As shown in Figure 26, after InterCL is applied, the majority of the inter-CTA locality in `MM` is preserved by L2 cache that is shared among SMs: Here the inter-CTA locality is composed of two types of L2 cache access localities – inter-CTA locality within a SM (intra-core) and inter-CTA locality across SMs (inter-core). Both intra- and inter-core hits on the L2 cache of `MM` dramatically rise after InterCL is applied. It is possibly because in a SM, InterCL prioritizes only a subset of warps, of which the working set can be fit into L2 cache and hit by other subsets of warps in remote SMs.
- The performance of `II` in C2 is considerably improved by InterCL i.e., 7.9% w.r.t. IWL+IntraCL. We find that the CTAs that sharing cache blocks in `II` are often in close proximity. These CTAs are very likely assigned to the same SMs by the serial CTA dispatching mechanism in InterCL, resulting in partially enhanced inter-CTA locality in a single SM.

The APKI of L2 cache reduces along with the decrease in *misses per kilo instructions* (MPKI) of L1 data cache[2]. The congestion on the interconnection network between SMs and memory partitions are also alleviated accordingly due to the less traffic between SMs and L2 caches. Figure 27 shows the reserved APKI of L1 data cache where the reserved accesses include all the accesses that fail to generate a complete list of transactions when accessing L1 data cache, e.g., an access misses on L1 data cache, but fails to deploy a L2

[2]The APKI of L2 is not necessarily equal to the MPKI of L1, because L1's MHSR can coalesce considerable amount of L2 accesses.

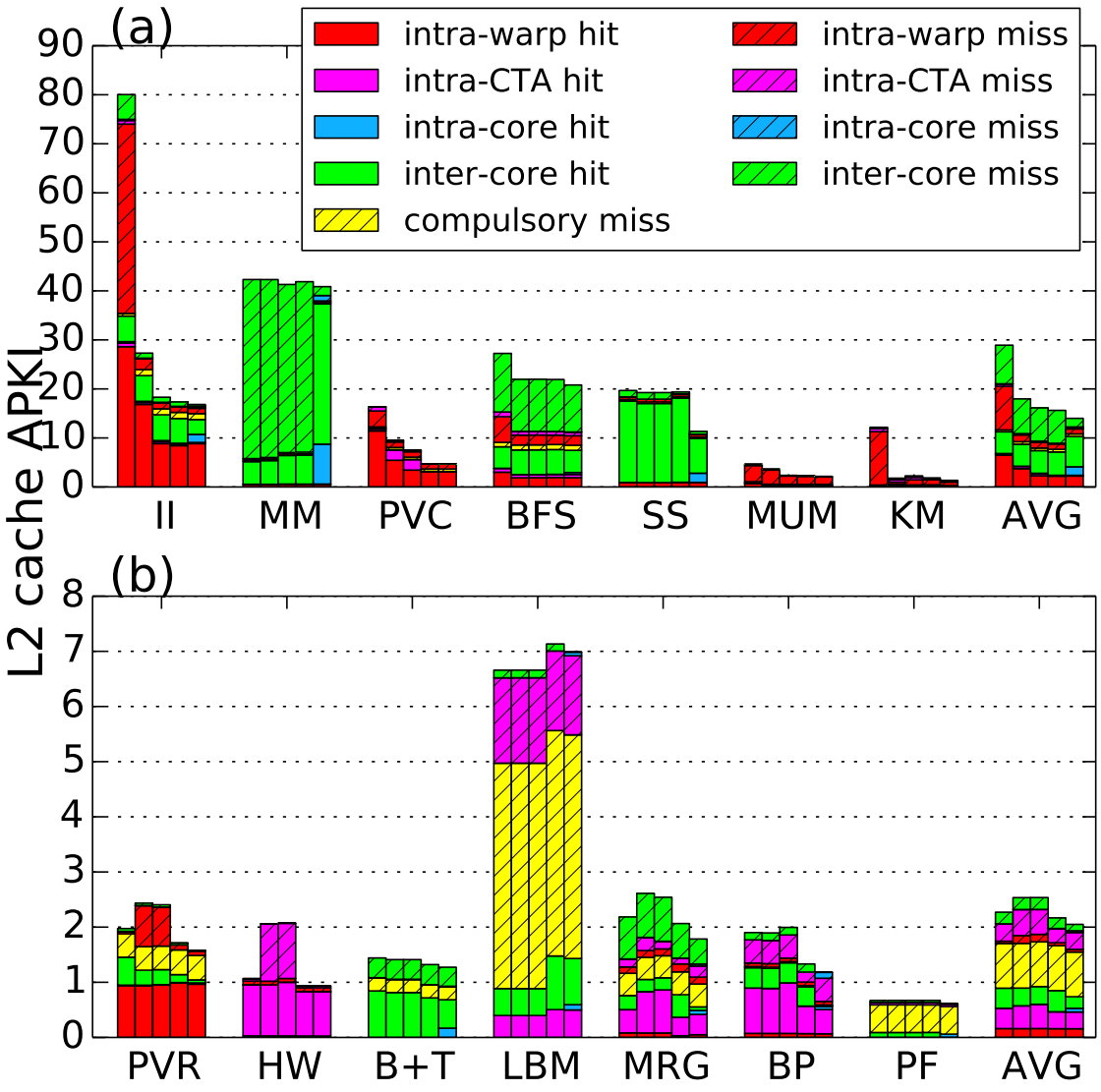

Figure 26: The L2 cache AKPI of (a) C1 and (b) C2 with LRR (1^{st} bar), CCWS (2^{nd} bar), IWL (3^{rd} bar), IWL+IntraCL (4^{th} bar) and VWS (5^{th} bar).

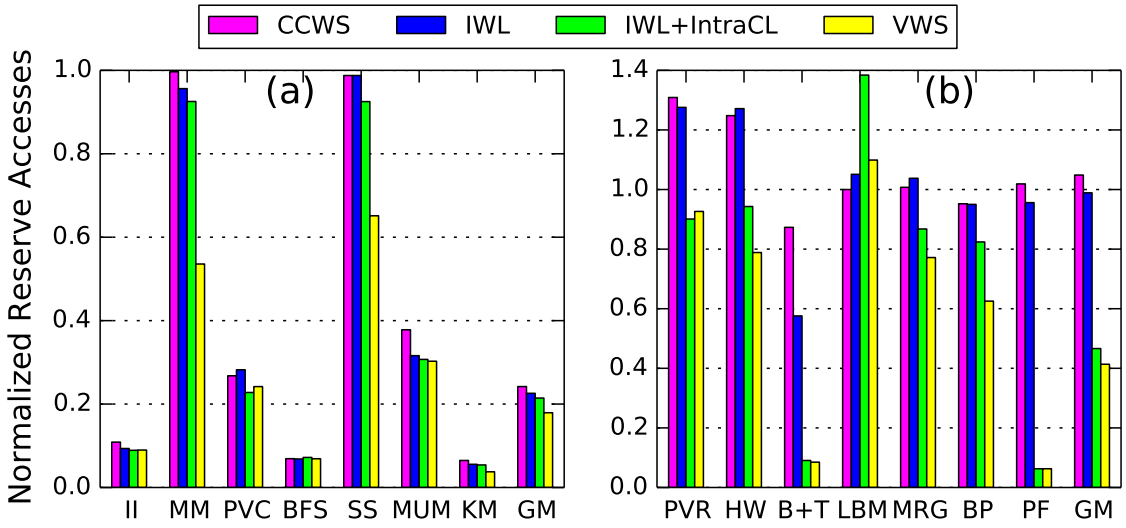

Figure 27: The reserved APKI of L1 data cache in (a) C1 and (b) C2, normalized to LRR.

access request on interconnection network due to network congestion. Results show that the reserved accesses can be greatly reduced in all benchmarks. The reduction of reserved APKI leads to less contentions on the shared resources and consequently, a better performance.

4.5 DESIGN EXPLORATION

Figure 28(a) depicts the impacts of changing the sampling stage period on the GPU performance. The peak performance is obtained when α is set to 0.1 in Equation (4.2).Note that $\alpha = 1$ means IWL is excluded and only IntraCL and InterCL take effect. In such a case, the system performance is significantly harmed. Figure 28(b) depicts the system performance at various warp throttling aggressiveness of IWL in VWS. On one hand, if the β in Equation (4.6) is too small, the thread-level parallelism will be hurt as IWL may overestimate the per-work working set. One the other hand, a larger β allows more warps to be active; it will lose the intra-warp locality and harm the performance speedup. The best selection of $\beta = 4$. From Figure 28 we can see that when $\beta = 4$ the performance of VWS is best.

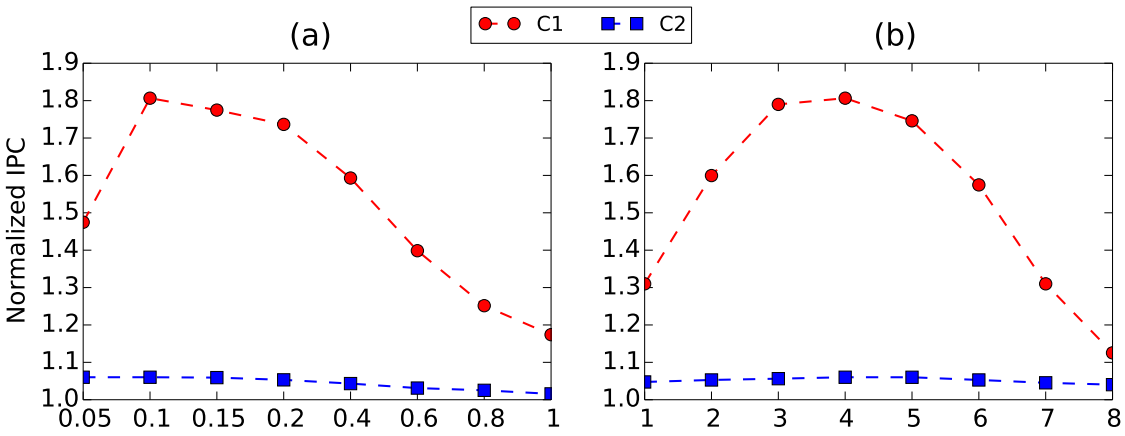

Figure 28: Performance of VWS at various IWL settings: (a) α in Equation (4.2) and (b) β in Equation (4.6), both of which normalized to the performance of LRR.

4.6 SUMMARY

In this chapter, we proposed VWS to explore multiple levels of cache locality in GPU executions. VWS consists of three components, each of which aims at capturing a particular type of cache locality. Firstly, IWL preserves the intra-warp locality by throttling the number of warps in a SM with dynamic per-warp working set estimation. Secondly, IntraCL keeps the intra-CTA locality by activating the warps from as few CTAs as possible. Finally, InterCL enhances the inter-CTA locality by introducing a serial CTA dispatching mechanism and a CTA-aware two-level schedule policy. The experimental results show that VWS can reduce more than half of L1 data cache misses. Across seven cache-sensitive GPGPU applications, VWS improves the average GPU performance by 80.7% and 17.4%, compared to the classic LRR scheme and the state-of-the-art CCWS scheme, respectively.

5.0 GPU DEVICE MEMORY OPTIMIZATION

In this chapter we introduce TEMP and TBAS to improve both the performance and energy-efficient of GPU. We first provide the background of the heterogeneous CPU-GPU integrated system and DRAM based GPU device memory. Then we detail the design of TEMP and TBAS, followed by the experiment methodology and results. At last we will summary our proposed techniques.

5.1 BACKGROUND

5.1.1 Heterogeneous CC-NUMA

The heterogeneous CPU-GPU integrated systems are evolving towards unified memory address space [54]. As the discrepant bandwidth requirements, most likely GPU is still physically attached with bandwidth-optimized DRAM while CPU is attached with capacity- and cost-optimized DRAM, and both DRAMs form a unified memory address space [55]. In such heterogeneous *cache coherent non-uniform memory access* (CC-NUMA) system, a computing unit has different access delays to local and remote memories even it sees a unified address space. Figure 29 shows a heterogeneous CC-NUMA system including CPUs and a GPU. The system interconnection networks bridge two memories and maintain the coherence between the caches of CPUs and GPU.

Heterogeneous CC-NUMA allows better programmability and fine-grained memory management of GPU. OS can allocate the GPU pages in all memories. In this work, we use the default NUMA page placement policy in Linux, *i.e.*, *local*, which places as many pages as

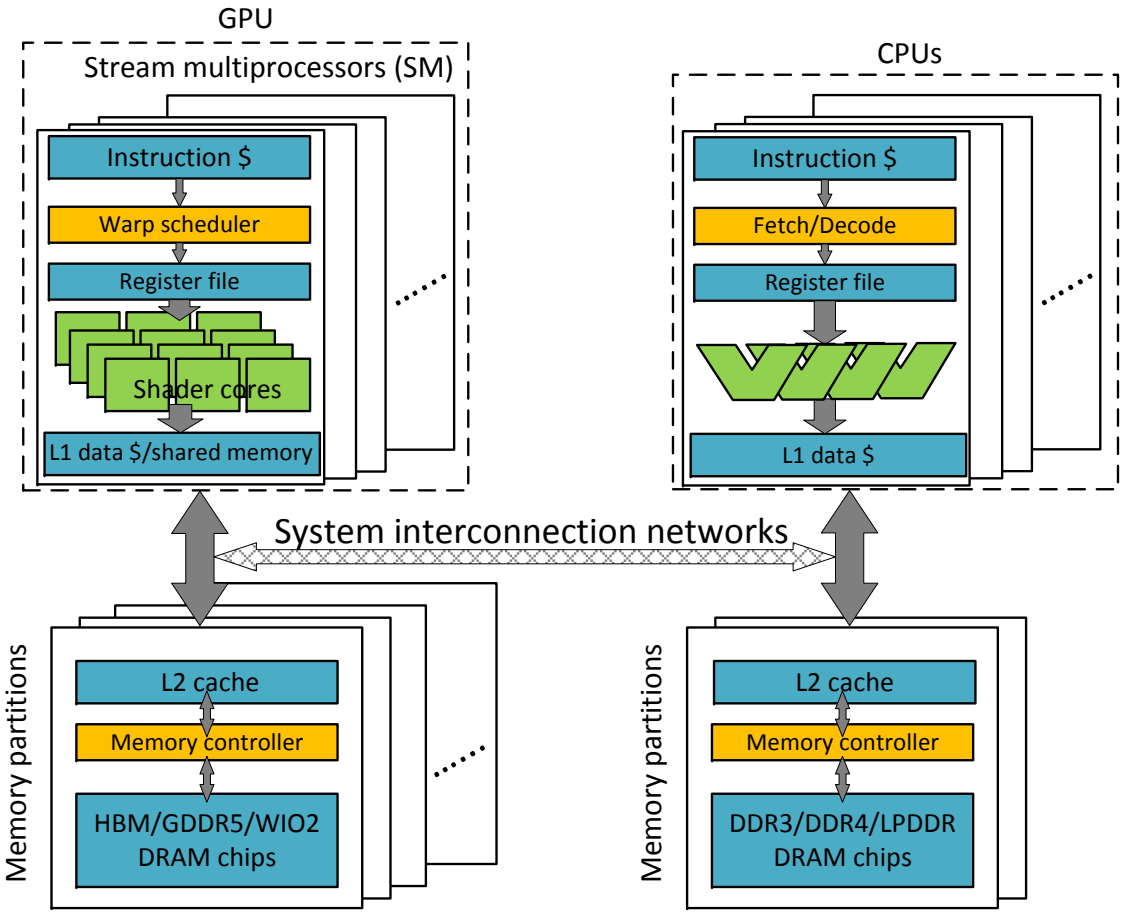

Figure 29: Organization of a heterogeneous CC-NUMA system.

possible in the local memory. By using `local` policy we can avoid most bandwidth contentions between GPUs and CPUs in heterogeneous CC-NUMA. The L2 caches of CPUs and GPU are separated and placed in memory partitions, each of which has its own memory channel. The on-chip caches, including L1 data and instruction caches in CPUs and GPU, are connected to the L2 caches via an mesh network. In such design GPU can take page faults and is no longer restricted to page locked memory [56] or non-pageable memory [13].

5.1.2 DRAM Basics

A modern JEDEC compliant DDRx DRAM system consists of one or more channels, each of which has its own buses for data, command, and address transferring. Figure 30(a) depicts the basic organization of a DRAM channel, which is affiliated with a *memory controller* (MC) to control the operations on the channel. A channel may be comprised of multiple DIMMs. Within each DIMM, there are several ranks each of which consists of multiple DRAM devices. In DDR3, a DRAM device contains 8 banks. The data of each bank are always pre-loaded to its private row buffer before being accessed.

DRAM address mapping complies with the DRAM organization. The address mapping in [1] is used as the baseline DRAM address mapping scheme in the heterogeneous architecture adopted in our work, as shown in Figure 30(b). We use the address mapping shown in Figure 30(c) for the page coloring mechanism utilized in our work. If the number of page offset bits is no greater than the sum of column and byte offset bits, by using page coloring, a GPGPU page can be mapped to any arbitrary channel, rank, bank or row in a bank.

DRAM usage efficiency is mainly determined by *bank-level parallelism* (BLP) [25] and row locality measured by *row buffer hit rate* (RBHR). All the banks in a rank can be accessed concurrently as each bank has its own address decoder and sensing logic. However, only one bank can put/receive the data on/from the shared bus at a time and all memory accesses (read and write) need to perform through the row buffer. Memory access latency and energy can be reduced when the access hits on the row buffer as no row activation is needed. In multi-core systems, many memory schedulers [24, 25] have been proposed to improve the BLP and row locality as well as maximize the access fairness. However, these designs are

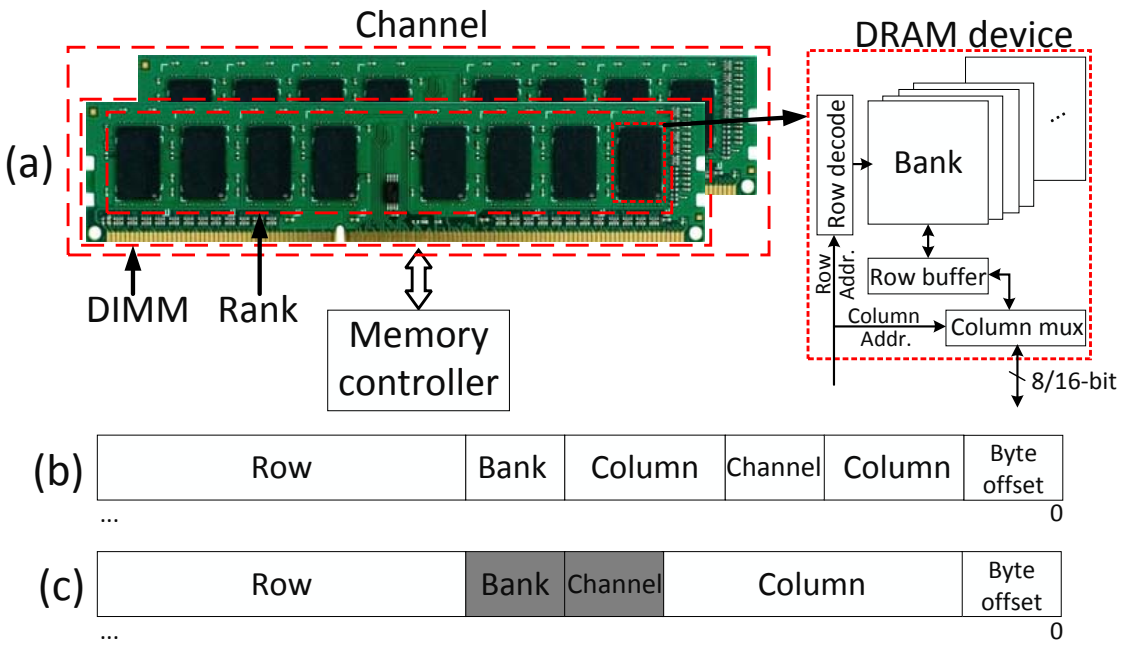

Figure 30: (a) The organization of a DRAM channel; (b) The default DRAM address mapping used in this work; (c) The modified DRAM address mapping used for page coloring. Shadow segment is the color bits.

generally insufficient to handle the massive parallel applications of GPU [27]. In this work, we propose TEMP and TBAS that can improve the DRAM efficiency in GPU by minimizing the inter-SM interference of memory accesses, which is the root reason of low BLP and low row locality of DRAM accesses [29].

5.2 THREAD BATCH ENABLED MEMORY PARTITIONING

A naive GPU memory partitioning may bind each SM to one or more banks. All the pages touched by a thread block can be placed to the banks bound to the SM where the thread block is executed. Ideally, if no page is shared by different thread blocks, the banks can be exclusively accessed by the associated SM. Unfortunately, page sharing between thread blocks commonly exists in GPU kernels. Simple page placement mentioned above is not able to separate the memory access streams raised from different SMs. To address the issue, we propose TEMP which *identifies and forms* the thread blocks sharing pages (Section 5.2.1) and *dispatches* them to the same SM (Section 5.2.2) so as to minimize the inter-SM interference of memory accesses. We call the group of these thread blocks sharing pages as a *thread batch*. The rest of this section will detail the design and implementation of TEMP.

5.2.1 Thread Batch Formation

By profiling the prevalent GPU benchmark suites, we find two major types of thread-data mappings with some page sharing patterns in thread blocks[1] The first major type of thread-data mappings is: the data accessed by each thread block is clustered over a sequential address space. Figure 31 shows the skeleton of the `Mapper` kernel in MapReduce engine of Mars [53]. This kernel employs fixed 1D thread blocks and scatters them to 1D or 2D grid. Generally, consecutive thread blocks sequentially access the 1D vector `inputKeys`, and each thread block accesses a linear address space ranging from `recordBase` to `terminate` within `inputKeys`.

[1] In this work, we only consider the kernels constructed with 1D or 2D thread block/grid because none of the profiled benchmarks employs 3D thread block/grid (see Table ??).

```
__global__ void Mapper(char* inputKeys, char* inputVals, int4*
              inputOffsetSizes, int recordsPerThread ...) {
       // calculate the global block and thread ID
       int bid = gridDim.y * blockIdx.x + blockIdx.y;
       int tid = threadIdx.x;
       int index = bid * blockDim.x + threadIdx.x;
       // a linear, contiguous data piece is assigned to this thread block
       int recordBase = bid * recordsPerThread * blockDim.x;
       int terminate = (bid + 1) * recordsPerThread * blockDim.x;
       // each thread retrieves its workloads from the data piece
       for (int i = recordBase + tid; i < terminate; i+=blockDim.x) {
              char *key = inputKeys + /*const offset*/;
              char *val = inputVals + /*const offset*/;
              map(key, val ...);
       }
}
```

Figure 31: Annotated code snippet of `Mapper` kernel in Mars library.

Figure 32 simplifies and visualizes the first type of thread-data mapping. In this example we assume the grid of the kernel contains four thread blocks, each of which consists of four threads. The 1D thread blocks are arranged in a 2D grid. Their accessed data matrix is shown in Figure 32(b). In this example, the first row of the data matrix is accessed by thread block *(0,0,0)*, the second row is accessed by thread block *(1,0,0)*, and so on. If the row address of the matrix aligns to page, SM-level page coloring can perfectly places the pages accessed by a SM to its bound banks, as depicted in Figure 32(c). Here a page is equal to a matrix row. However, if a page is comprised of multiple matrix rows, say, two matrix rows, conventional thread block dispatching which interleaves thread blocks across SMs will generate interweaved memory accesses, as shown in Figure 32(d). In order to address the situation, we can pack those thread blocks accessing the same set of pages into a *thread batch* and then dispatch the thread batch as a whole to a SM. For the example shown in Figure 32(d), we can group the 4 thread blocks into 2 thread batches, each of which goes to a SM. The memory accesses to banks 0 and 1 are successfully separated, as illustrated in Figure 32(e).

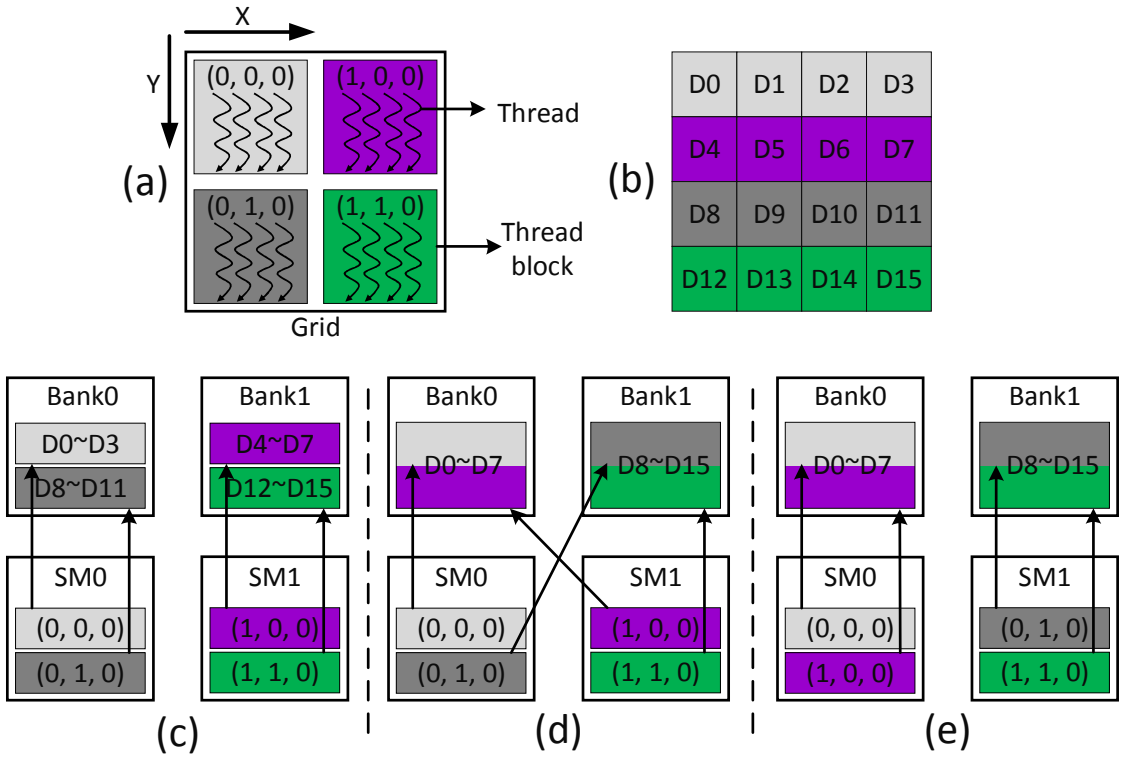

Figure 32: (a) A 2D grid – 1D thread blocks; (b) The accessed data matrix; The memory access footprint when the page size is (c) a matrix row, two matrix rows (d) without or (e) with thread batching.

```
__global__ void cenergy(int numatoms, float * energygrid ...) {
       // assign a point from the energygrid to this thread
       // PointsPerThread == 2, so each thread works on two points
       int xindex  = blockIdx.x *  blockDim.x * PointsPerThread + threadIdx.x;
       int yindex  = blockIdx.y * blockDim.y + threadIdx.y;
       int outaddr = gridDim.x * blockDim.x * PointsPerThread * yindex + xindex;
       // calculate the energy of two points based on xindex and yindex
       for (atomid=0; atomid<numatoms; atomid++) {
              energyvalx1 += /*simple math on xindex and yindex*/;
              energyvalx2 += /*simple math on xindex and yindex*/;
       }
       // Update two points on the energygrid
       energygrid[outaddr]   += energyvalx1;
       energygrid[outaddr + blockDim.x] += energyvalx2;
}
```

Figure 33: Annotated code snippet of `cenergy` kernel in `CUTCP`.

The second major type of thread-data mappings is that the data accessed by consecutive thread blocks are interleaved over a linear address space. Figure 33 shows the code snippet of the `cenergy` kernel in the `CUTCP` benchmark [9]. `CUTCP` computes the coulombic potential at a molecular grid `energygrid`. A point in `energygrid` is indexed by `xindex` and `yindex` generated from a thread's indies. All threads form a 2D grid which is further tiled with 2D thread blocks. Figure 34 demonstrates a simplified thread-data mapping in this 2D grid. The thread organization and accessed data matrix can be found in Figure 34(a) and (b), respectively. Here we again assume one grid with four thread blocks and each thread block has four threads. In this example, every thread block has two active dimensions (x-axis and y-axis). Each matrix row is accessed by two thread blocks while each thread block accesses two rows. In such a situation, it is very likely that the consecutive thread blocks access the same set of pages. Similarly, we can pack those thread blocks sharing the same set of pages into a thread batch. Figure 34(c) gives an thread batching example where every matrix row in Figure 34(b) exactly forms one page. Thread block *(0,0,0)* and *(1,0,0)* share pages 0 and 1 while thread block *(0,1,0)* and *(1,1,0)* share pages 2 and 3. Consequently, we can group thread block *(0,0,0)* and *(1,0,0)* into thread batch 0 and group thread block *(0,1,0)* and *(1,1,0)* into thread batch 1. By allocating pages 0 and 1 into bank 0 and pages 2 and 3 into bank 1, the memory accesses from SM 0 to bank 0 and from SM 1 to bank 1 are separated.

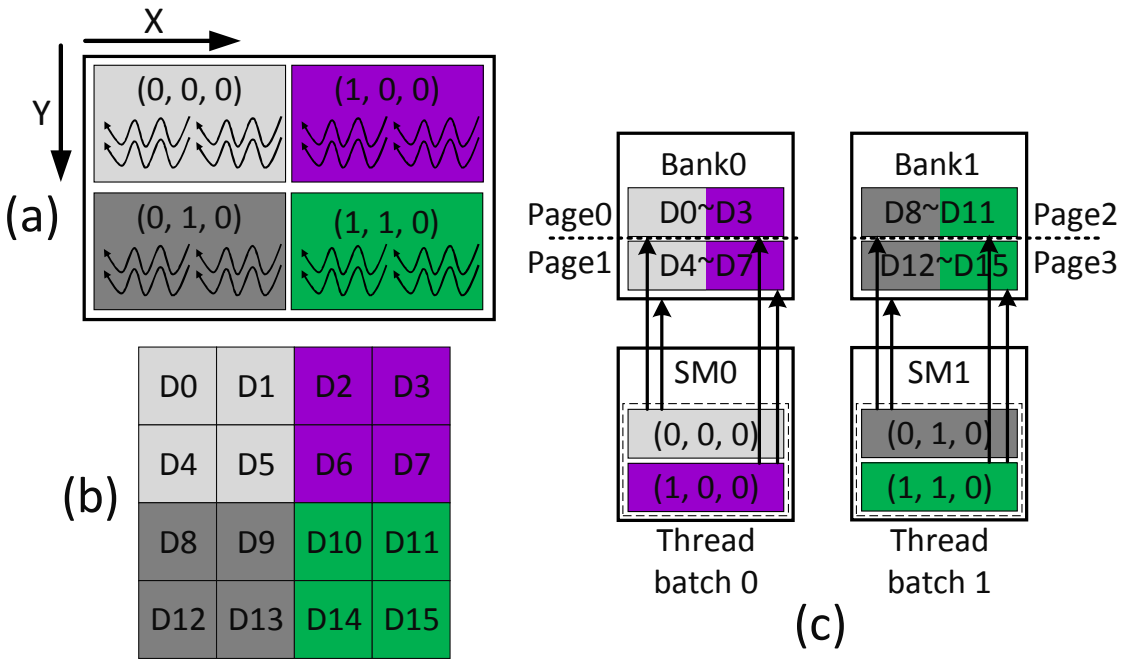

Figure 34: (a) A 2D grid – 2D thread blocks; (b) The accessed data matrix; (c) The memory access footprint with thread batching.

The two major thread-data mappings indicate consecutive thread blocks probably share pages. Accordingly, we introduce *thread block stride* to indicate the number of the consecutive thread blocks that belong to the same thread batch. In the examples in Figure 32(c) and 34(c), thread block stride is 1 and 2, respectively. To find the thread block stride of a GPU kernel, one approach is to profile a kernel given a page size. This profiling can be done during the compile time since the programmer already determined the thread hierarchy and how the threads access the data matrices. In profiling stage, the beginning addresses of the data matrices are set to zero. Therefore, during dynamic memory allocation, the starting memory address of a data matrix shall align to the beginning of pages to guarantee the thread block stride found in compile time. Figure 35 shows the optimal thread block stride of some GPU applications. Optimal thread block stride is the one suppressing most cross-batch page sharing. Here the page size is set to 4KB supported by most machines. 89% of kernels achieve the minimum inter-thread batch page sharing through a batch formation with a fixed thread block stride. There are also 6% of kernels where the batch formation can be realized using modulation. Some kernels in `MUM` and `LBM` cannot fitted with a formula for the batch formation.

The static compile time profile, however, may be sub-optimal since it cannot proactively remove the cross-batch page sharing. In the next section we will introduce a simple dynamic hardware approach which can better support thread batching without paying heavy effort on static profile.

5.2.2 Serial Thread Block Dispatching

Thread batching cannot completely eliminate the cross-batch page sharing in some GPU kernels. For example, the last thread block in a thread batch may share a page with the first thread block in its following thread batch, if those thread batches are formed with fixed thread block stride. We further analyze some GPU applications which form thread batch with fixed thread block stride. The accumulated percentage of the pages shared by different sizes of consecutive thread batches is shown in Figure 36. Horizontal axis shows the maximal distance of the thread batches sharing pages. Among all the accessed pages,

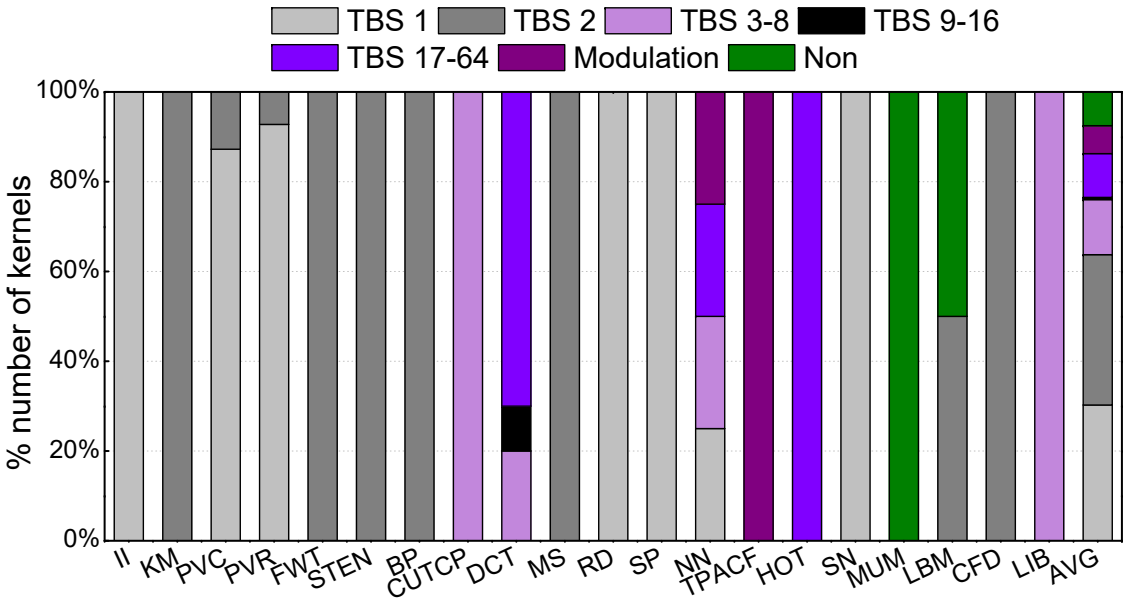

Figure 35: The distribution of thread block stride.

nearly 75% on average is exclusively accessed by a single thread batch and 22% is accessed by two consecutive thread batches. These two cases dominate the page access patterns of thread batches ($> 97\%$). There are more than 2% of pages are globally shared by almost all thread batches in a kernel, such as program text pages.

Given the stride thread batching and cross-batch page sharing dominating the GPU applications, we propose *serial thread block dispatching*. The consecutive thread blocks, which are very likely enclosed by the consecutive thread batches, are emitted to a SM. As such most thread batches are formed implicitly by the serial thread block dispatching, and most cross-batch page sharing are constrained within a SM. Now the cross-batch page sharing only happens when some thread blocks of a thread batch is distributed to more than one SMs. This would happen in the first or last thread batch in a SM.

Traditional interleaved thread block dispatching, *e.g.*, *GigaThread engine* in NVIDIA GPU [13], generates and dispatches a new thread block to a SM once a SM has an idle slot. Typically, the dispatching unit only passes the id of the new thread block to the SM, and the SM will construct the whole thread block according to the received thread block id. The

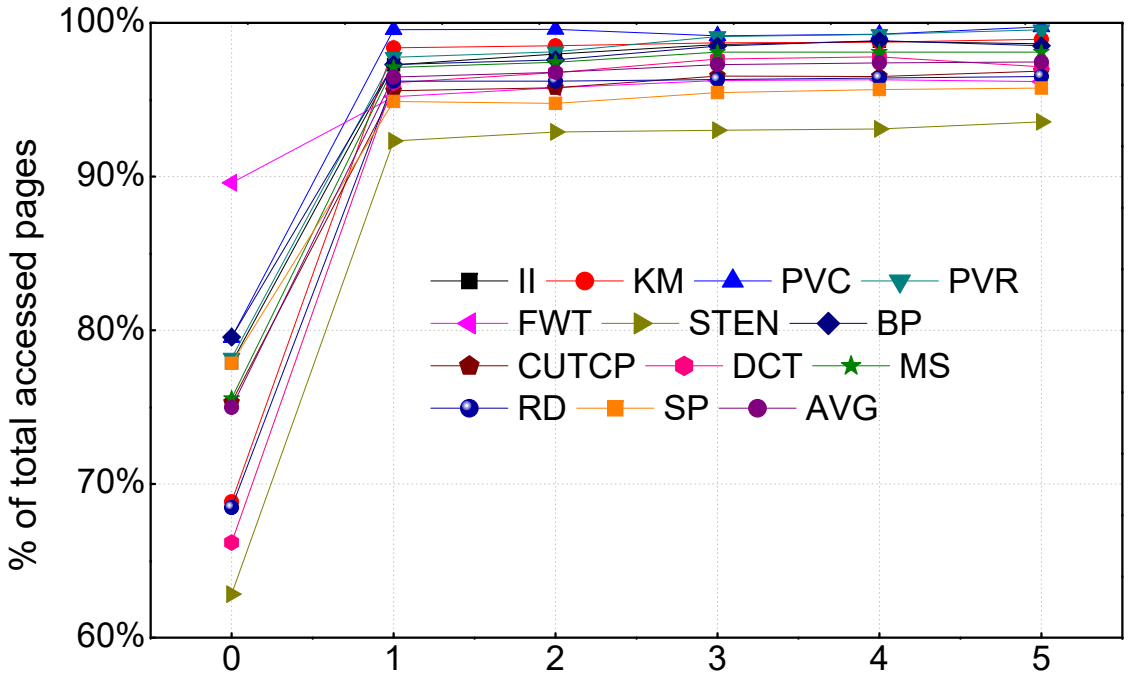

Figure 36: The accumulated percentage of page sharing between thread batches.

dispatching unit generates the thread block ids sequentially and the thread block ids are dispatched to SMs randomly. To implement deterministic, serial thread block dispatching, we introduce a dispatch queue in every SM. The content, *i.e.*, the thread block ids, in the dispatch queue are pre-inserted before launching a kernel. Each SM receives similar amount of thread block ids for workload balance, which can be determined at the compile time. During the kernel execution, thread block ids are popped from the dispatch queue and pushed to the associated SM.

Compared to traditional thread block dispatching, serial thread block dispatching does not delay the launch of thread blocks, because a SM can always pop a thread block id from its dispatch queue when it has an idle slot. The implementation of the dispatch queue can be highly efficient since only two extra registers are required in each SM, to record the head and the tail of thread ids, respectively. The head register is increased by one once a new thread block id (the head register itself) is popped. The dispatching of thread block ends when the head register meets the tail of thread block id. Thus, the serial thread block dispatching incurs marginal timing and hardware overheads.

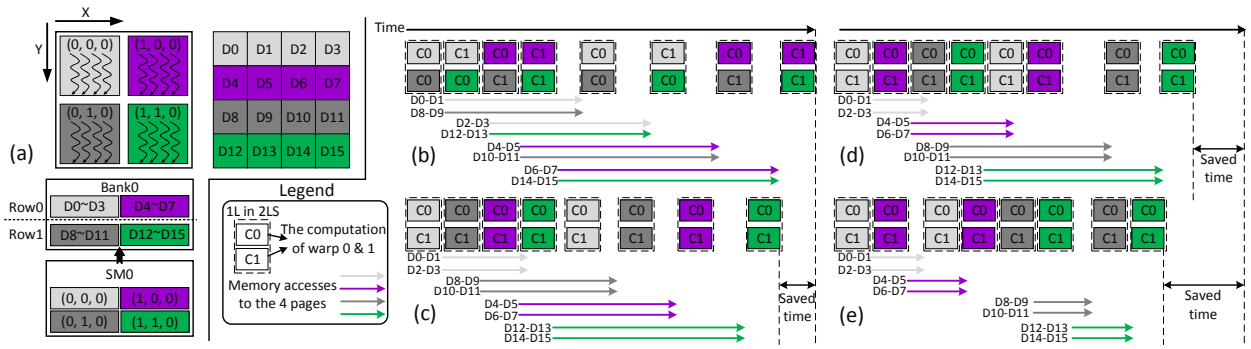

Figure 37: (a) The thread organization, data matrix, thread batches and memory access footprint of a kernel running on SM0; (b) The execution sequence of warps generated by CCWS; (c)∼(e) The progressive improvement of TBAS.

5.3 THREAD BATCH-AWARE SCHEDULING

TEMP constrains the memory accesses from a SM within the associated memory banks, offering an opportunity of scheduling the execution of the threads for intra-bank/row locality enhancement. Our proposed TBAS can be explained by an example in Figure 37: Figure 37(a) shows the thread organization and data matrix in the example. In a GPU, there is only one SM (i.e., SM0) associated with its own DRAM bank. Four thread batches, each of which consists of only one thread block, are formed and dispatched to SM0. Every thread batch exclusively accesses its own page while the page layout of SM0's bank is also shown in Figure 37(a). We assume two pages are included in one row in the bank[2]. Every two threads in a thread block forms a warp. Since there are 4 threads in one thread block, each thread block has 2 warps and total 8 warps (or 4 thread blocks) are running on SM0.

Figure 37(b) shows the execution of SM0 with CCWS [15]. Typically, CCWS only keeps a subset of warps running and throttles the rest of warps pending in SM, if a GPU application suffering from cache thrashing is detected. Once a warp in the running set triggers a long operation and encounters a stall, it will be demoted to pending set. Simultaneously, another warp in pending set will be promoted to running set. Here we assume the size of running

[2]Generally the row size of a DRAM is multiple times greater than the smallest page size that the OS can support.

set is two warps. It is very likely that the two warps in running set come from different thread batches. Hence, they may compete for different rows in the bank and degrade the row locality.

We can propose a better CCWS design to improve the row locality, as shown in Figure 37(c): the running set always tries to gather active warps from the same thread batch as they normally access the same page (obviously also the same row). If the thread batch in running set does not have sufficient active warps, all the warps of this thread batch are demoted to pending set and a new thread batch that has sufficient active warps will be promoted to running set.

In such a design, promoting warps may still harm the row locality if the row accessed by the previously active warps in running set is not the same as the one accessed by the newly promoted ones. It happens only when the thread batch which the previously active warps belong to is different from that of the promoted ones. Hence, as shown in Figure 37(d), a better promotion scheme can be promoting a thread batch that is the successor of the demoted thread batch, e.g., promoting *(1,0,0)* (or *(1,1,0)*) after demoting *(0,0,0)* (or *(0,1,0)*). As the result of page allocation, it is very likely that the adjacent thread batch will access the same row in the bank.

The above sequential thread batch switching often results in round-robin execution sequence, potentially incurring the burst of memory accesses in a short time window. As illustrated in Figure 37(d), all memory accesses are evoked in the first four scheduling cycles. Consequently, there are two situations that may deteriorate the efficiency: 1) A thread batch demoted from pending set due to the long operation could access the same page again in the near future. However, it may not be scheduled again in time to detain such locality. 2) When generated by the thread batches which are continuously promoted to running set, the burst of memory accesses is coupled with the lost locality. The prolonged queuing delay in memory controller may overwhelm the reply network connecting memory controllers and SMs [57].

To overcome the above drawbacks, we assign higher promotion priority to older thread batches in the pending set. Assume the priority of the thread batches in Figure 37(a) descends from left to right and then from top to bottom. Figure 37(e) shows the scheduling

sequence of the thread batches considering our proposed promotion priority. The improvement of row locality, especially the decreasing of memory access burst, leads to significant reduction in average memroy access latency. We named the scheduling method corresponding to the example presented in Figure 37(e) as TBAS.

Besides the maintenance of intra-/inter-thread batch row locality and alleviation on congestion of reply network, TBAS also reduces the stretch of memory access footprint by limiting the active thread batches in a particular time window. Such limitation on thread-level parallelism can bring in an implicit positive side effect on the cache locality [15] as we shall explain in Section 5.5.1.

The hardware overhead of TBAS is similar to that of CCWS except for the promotion priority arbitrator. Fortunately, the number of concurrent thread batches in a SM is usually small: A SM of Fermi GPU, for example, supports only 8 concurrent thread blocks (or at most 8 thread batches). Therefore, the implementation overhead of the arbitrator is negligible.

5.3.1 Comparison between TBAS and VWS

TBAS targets the DRAM locality and does not explicitly perceive cache locality, while VWS only captures the cache locality. However, we find TBAS also implicitly preserves the cache locality which was not included in its original design goal. In fact, as shown in Figure 37(e), the thread-level parallelism is throttled since only a subset of thread batches are active in a particular time span, keeping the working set with limited size in a capacity-bounded L1 data cache. Such implicit throttling is similar to the IWL component in VWS. In Figure 37(c), the warps from the same batch are prioritized beyond the warps from the other batch. Because a thread batch usually contains a list of continuous thread blocks, TBAS partially implements intraCL when a thread batch only contains single thread block. TBAS also explore inter-CTA cache locality, as shown in Figure 37(d). We examined the execution statistic when running TBAS with TEMP and found that it is comparable to CCWS in terms of the L1 cache misses. Synergistic combination of VWS and TBAS may potentially provide a comprehensive solution covering both L1 data cache and DRAM locality. We will further discuss such possibility in Chapter 7.

Table 6: The characteristics of GPU applications: the category an application belongs to; the used thread organizations labeled as (grid dimension, block dimension).

Application	Abbreviate	Category	Thread organizations	MPKI
InvertedIndex	II	C1	(1D, 1D)	45.23
Kmeans Clustering	KM	C1	(1D/2D, 1D)	11.91
PageViewCount	PVC	C1	(1D, 1D)	5.52
PageViewRank	PVR	C1	(1D, 1D)	2.10
Fast Walsh Transform	FWT	C1	(1D, 1D)	1.01
Seven Point Stencil	STEN	C1	(2D, 2D)	0.85
Back Propagation	BP	C1	(1D, 2D)	0.75
Merge Sort	MS	C1	(1D, 1D)	0.53
Discrete Cosine Transform	DCT	C1	(2D, 2D)	0.37
Scalar Products	SP	C1	(1D, 1D)	0.17
Reduction	RD	C1	(1D, 1D)	0.04
Coulombic Potential	CUTCP	C1	(2D, 2D)	0.03
Neural Network	NN	C2	(1D/2D, 1D)	1.21
Hotspot	HOT	C2	(2D, 2D)	0.75
Sorting Networks	SN	C2	(1D, 1D)	0.27
Angular Correlation	TPACF	C3	(1D, 1D)	0.89
MUMmerGPU	MUM	C3	(1D, 1D)	5.85
Lattice-Boltzmann Method	LBM	C3	(2D, 1D)	5.82
CFD Solver	CFD	C3	(1D, 1D)	0.85
LIBOR Monte Carlo	LIB	C3	(1D, 1D)	0.62

Table 7: The characteristics of CPU applications.

Application	MPKI	Application	MPKI
mcf	52.28	bzip2	4.14
omnetpp	34.67	h264ref	1.95
xalancbmk	27.52	gcc	0.58
lbm	20.23	perlbench	0.21

5.4 EXPERIMENT METHODOLOGY

5.4.1 Application Selection

We adopt a diverse GPU application set from [11, 1, 10, 19, 9] in our evaluations. Most of the applications are fully simulated except for the applications from [19] of which only the first two billion instructions are simulated. The detailed characteristics of each application are summarized in Table 6. All GPU applications are profiled to generate the optimal thread batches before execution.

To evaluate the effectiveness of the integrated TEMP and TBAS (TEMP+TBAS) for heterogeneous workloads, we choose eight CPU applications from SPEC CPU 2006, as shown in Table 7. PinPoint [58] is used to extract the execution phases for all CPU applictions. The CPU applications are divided into two types: memory intensive where L2 cache *misses per kilo instructions* (MPKI) is higher than 20; and memory non-intensive where L2 cache MPKI is lower than 20. The GPU applications can be also divided into two types based on L2 cache MPKI – memory intensive (MPKI>2) and non-intensive (MPKI<2). Although the L2 cache MPKI of most GPU applications are lower than that of CPU applications, within an arbitrary time window, GPU applications possibly generate L2 cache misses two orders of magnitude greater than that of CPU applications due to their high instruction throughput (i.e., IPC).

We permute the combination of different types of CPU and GPU applications, coming up twelve heterogeneous workloads. Each workload consists of two CPU applications and one GPU application, as shown in Table 8. The performance of C1 is highly sensitive to TEMP+TBAS (shall be explained in Section 5.5.1), we construct ten workloads (WL0∼WL9 in Table 8) where the GPU applications are picked up from C1. Half of GPU applications in WL0∼WL9 are memory intensive while the rest are memory non-intensive. For the CPU workloads in WL0∼WL9, we can have three combination types (i.e., NN, IN, and II) of the dual-application. The generated ten heterogeneous workloads cover most cases where EMU may act variably. We also construct two extra workloads, each of which includes an GPU application from C2 and C3, respectively.

5.4.2 Simulation Platform

Since the CPU-GPU CC-NUMA has not been shipped by any industry vendors, we simulate a GPU system attached with heterogeneous GDDR5-DDR3 DRAM subsystem. System simulation is performed on gem5-gpu [61], and its simulation parameters are listed in Table 9. Table 9 summarizes the configuration of the simulator.

The GPU subsystem includes 8 SMs, each of which has the similar computational capability as the SM in Fermi, except we lower the SM frequency to 600MHz. The memory bandwidth per *shared-core-clock* is comparable and even higher than that of real high-end heterogeneous processors integrating similar GPU unit [62]. Such setting ensures our simulation platform resemble real product and rules out the concern that our simulation configuration favors the proposed techniques.

The page size is set to 4KB, a typical size adopted widely. To avoid the bottleneck of GPGPU TLB and expose the limitations of DRAM bandwidth in heterogeneous shared memory systems, we also optimize the GPU TLB design in our heterogeneous system including per-SM TLB, highly-threaded PTW and shared L2 TLB [63]. We choose the configuration with CCWS in [15] as our baseline.

We estimate the GDDR5 DRAM energy consumption through a modified MICRON DRAM power calculator [64] based on the datasheet [22]; the DDR3 DRAM energy consumption is directly obtained from MICRON DRAM power calculator by feeding the runtime statistic generated from gem5-gpu.

Table 8: The characteristics of heterogeneous workloads.

Workloads	Type*	Applications
WL0	NN-N	perlbench, bzip2, FWT
WL1	NN-N	gcc, h264ref, BP
WL2	NN-I	gcc, bzip2, II
WL3	NN-I	perlbench, h264ref, KM
WL4	IN-N	omnetpp, gcc, STEN
WL5	IN-I	xalancbmk, h264ref, PVC
WL6	II-N	mcf, lbm, DCT
WL7	II-N	omnetpp, xalancbmk, FWT
WL8	II-I	mcf, xalancbmk, PVR
WL9	II-I	omnetpp, lbm, II
WL10	IN-N	lbm, bzip2, NN
WL11	IN-I	mcf, perlbench, MUM

*Each workload type is denoted by the types of combined applications, e.g., NN-I means two memory non-intensive CPU applications run with a memory intensive GPU application.

Table 9: Simulation configuration.

CPU	
Number of Cores	2
Execution	3 GHz, OOO 4 issues, 256-entry ROB
L1 Data Cache	32KB 4-way, 2-cycle hit, write back, 64B
L2 Cache	2MB 8-way, write back, 64B
GPU	
Number of SMs	8
SM Clock	600MHz
SIMD width	16
L1 Data Cache	32KB 4-way, write through, 128B
Warp Size	32
Max Number of Threads	1536/SM
Max Thread Blocks	8/SM
Scheduler	CCWS, OWL, TBAS
TLB	64-entry L1, 8KB page walk cache, 512-entry shared L2
L2 Cache	128B line, 8-way associated, 2 banks, 512KB/bank, total 1MB
Shared resources	
# of memory channels	2 for GDDR5, 1 for DDR3
Memory controller	FR-FCFS [59], open-page, 64-entry request queue/MC
Interconnection	2D Mesh
GDDR5 Banks	16
Timing	from [22]
DDR3 Banks	8
Timing	from [60]

5.5 RESULT

5.5.1 Evaluation for Pure GPU Workloads

5.5.1.1 Performance We first analyze the performance of different configurations employing various schedulers and/or combinations of schedules and TEMP, only pure GPU workloads are evaluated. The results are normalized to CCWS that does not have any optimization on DRAM system, as shown in Figure 38. Applying TEMP on top of CCWS introduces 5.7% GM speedup, while replacing CCWS with TBAS can further raise the speedup to 10.3%. We also compare to OWL [19]. OWL targets on cache performance through intelligent warp scheduling. It also tries to improve the BLP of memory accesses by prioritizing different-numbered thread blocks in consecutive SMs. From our evaluation, OWL is 93.6% within the performance of CCWS across the application set. We observe that CCWS has higher cache hit rate compared to OWL, and the BLP improvement of OWL is limited because only a small subset of thread blocks which share pages are considered. As a result, TEMP on top of CCWS is 12.9% better than OWL. Figure 39 shows the local access ratio to each memory bank. Here *local access* denotes the memory access from the SM associated to the banks while *remote access* denotes the access from other SMs.

To better understand the respective efficacy of TEMP and TBAS, we partition all the GPU applications into three categories, by considering both the local access ratio and the speedup under TEMP:

- **C1**: These applications have very high local access ratio (on average >99%) and obtain substantial performance improvement, in all the configurations employing TEMP.
- **C2**: Similar to C1, the applications in C2 also demonstrate high local access rate (>93%). Compared to CCWS, however, introducing TEMP to these applications results in slight performance degradation (∼1%) except TEMP+TBAS can still effectively boost the GM speedup to 2.4%.
- **C3**: Even applying the best thread batch formation, the applications in C3 cannot obtain very high local access ratio due to intrinsic thread-data mapping and memory access pattern. Compared to CCWS, we cannot see any performance speedup but an overall degradation.

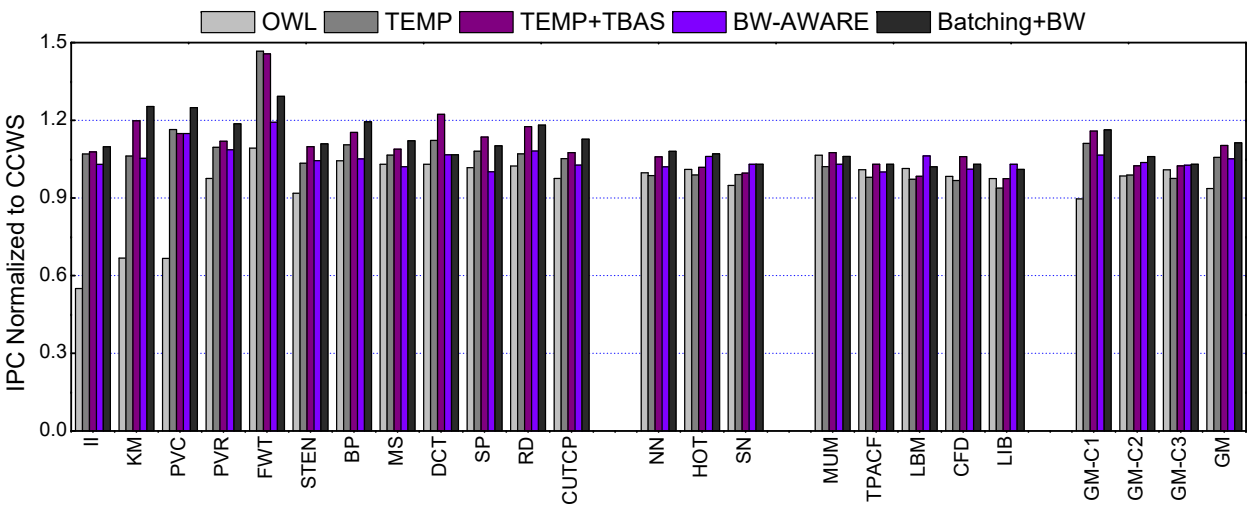

Figure 38: The performance of different schemes, all of which are normalized to that of CCWS.

We summarize the DRAM usage statistic as well as the *stall on reply network* from memory controllers to SMs in Figure 40. All the data are normalized to those in CCWS. The BLP of TEMP is significantly improved by 58.3% in C1, while the RBHR is increased by 17.8%. As expected, by suppressing the inter-SM interference of memory accesses, TEMP unveils the intrinsic locality and access parallelism of thread batches. On the other hand, OWL obtains 16.3% BLP and 8.6% RBHR improvement. OWL adopts opportunistic prefetching to burst the RBHR while TEMP does not introduce any prefetching. We also investigate the network congestion between the SMs and GDDR5 DRAM partitions. The network congestion of OWL is 33.6% more than that of CCWS. This value quantitatively demonstrates that CCWS has higher L1 cache hit rate, less L2 accesses, most likely less DRAM accesses compared to OWL. All the above factors together lead to 17.3% reduction in DRAM access delay with TEMP in C1. Consequently, TEMP bursts 11.1% performance, 24.0% higher than OWL in C1. For TEMP+TBAS, C1's BLP is 9.1% smaller than that of TEMP. This is because the number of active thread batches is intentionally limited for row locality enhancement. On the other hand, C1's RBHR in TEMP+TBAS is further raised by 33.1% and its DRAM access delay is drastically reduced by 29.9%. More importantly, a considerable reduction in congestion of reply network (18.7%) is observed. As a result, more than 15% performance speedup is achieved in C1 with TEMP+TBAS.

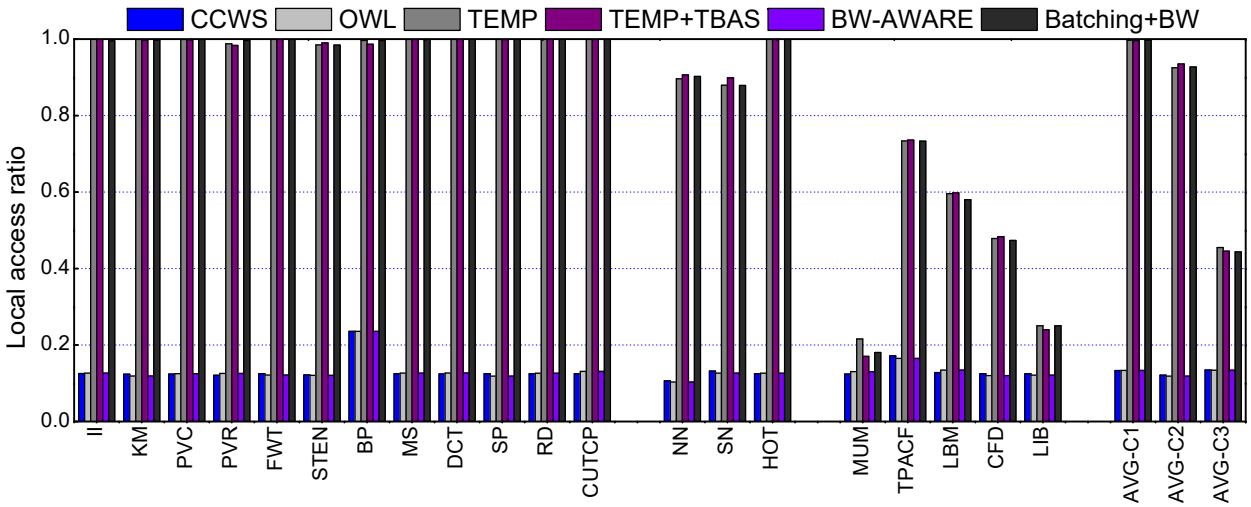

Figure 39: The local access ratio of memory accesses of different schemes.

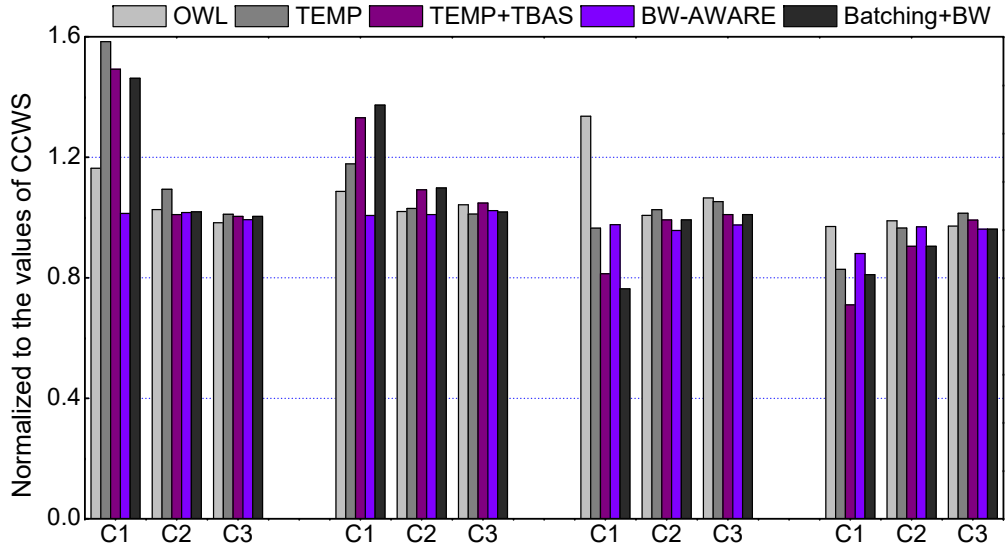

Figure 40: The statistic of DRAM system and congestion on reply network.

With TEMP, C2 achieves a high local access rate. However, it is hard to increase its BLP since it already approaches the theoretical upper bound. For instance, some kernels in `NN` have only a few thread blocks: the number is even lower than the bank count. Applying TEMP on those kernels possibly limit the BLP. Fortunately, TBAS takes effect in terms of row locality enhancement and network congestion reduction, resulting in slight speedup ($\sim$2%). On average, the performance of C3 in TEMP/TEMP+TBAS is reduce/improvement by 2.5%/2.3%. Note that it is difficult to formalize the thread-data mapping of the applications in C3. Thus, applying TEMP aggravates DRAM access delay.

We also consider a synergistic page placement policy, *bandwidth-aware* (BW-AWARE) [55] which places the GPU pages across GDDR5-DDR3 DRAM. The memory footprint of each GPU application we evaluate completely fits within the GDDR5 DRAM, so *local* page placement policy places all the GPU pages in GDDR5 DRAM. BW-AWARE, on the other hand, keeps a page placement ratio the same as the bandwidth ratio between GDDR5 and DDR3, by which BW-AWARE can fully utilize the combined bandwidth of both memories. From Figure 38 we can see BW-AWARE gains 5.1% extra performance on top of CCWS. The performance gain is compliance to the value reported in [55] given the similar bandwidth ratio. TEMP and TBAS is orthogonal to BW-AWARE. To combine TEMP and BW-AWARE: 1) we assign a DDR3 DARM bank to a SM. 2) BW-AWARE decides which memory to place a new page. 3) TEMP then places the page in the bank bound to the SM from which this page is raised. 4) And TBAS scheduler the warps as if a homogeneous DRAM is employed. As a result, combing TEMP, TBAS, and BW-AWARE (Batching+BW) has better BLP and RBHR compared to BW-AWARE, and further improve the performance by 11.2% compared to CCWS.

5.5.1.2 Energy The normalized DRAM energy consumption of all configurations is shown in Figure 41. Generally, the DRAM energy savings come from two main sources: 1) the saving of activate energy that dominates DRAM energy consumption because of the increased RBHR and 2) the saving of the background energy which is proportional to the reduction of the execution time. Therefore, the DRAM energy reduction is determined by its access locality as well as the overall performance improvement. Our results show that

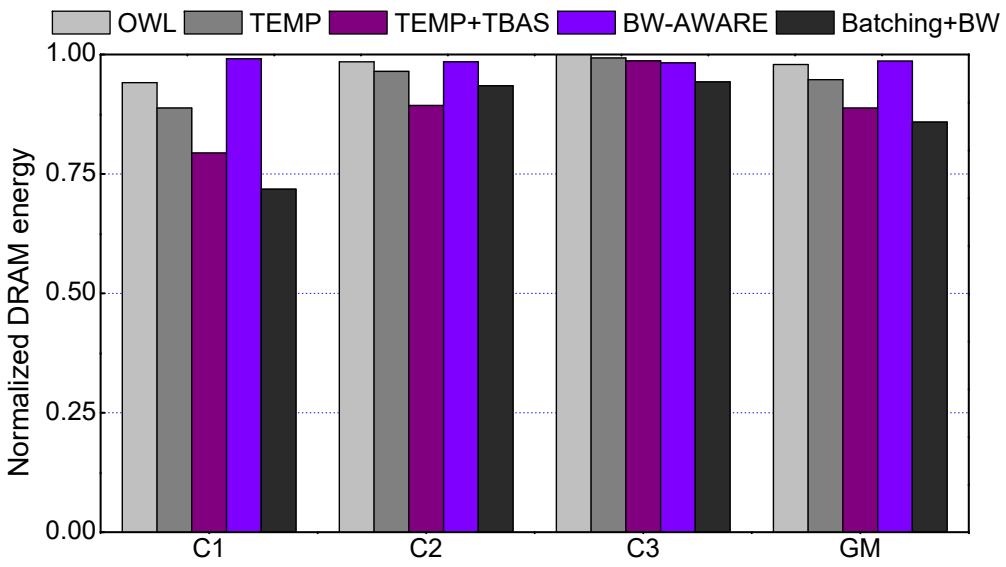

Figure 41: The normalized DRAM energy consumption of different schemes.

compared to CCWS, the DRAM energy savings of TEMP is 11.2%. TEMP+TBAS is 20.7% better than CCWS because the significant improvement of RBHR. OWL saves less DRAM energy, 5.9% because the higher row activation plus inferior performance. Batching+BW achieves the most energy saving by 14.2%.

5.5.2 Evaluation for Heterogeneous Workloads

Figure 42 shows the performance of CPU (WS-C) and GPU (IPC-G) applications in each heterogeneous workload when TEMP+TBAS is applied. The performance of CPU applications in a workload is measured by the weighted speedup [65], and normalized to the weighted speedup of the same CPU applications running exclusively on the heterogeneous system. The IPC of a GPU application is also normalized to the IPC obtained by running with TEMP+TBAS exclusively. The memory intensive applications, no matter CPU or GPU applications in the workloads, experience non-trivial performance slowdown due to the contention in shared resources, e.g., interconnection network and DRAM. On the contrary, the performance slowdown of memory non-intensive applications is much less. The weighted

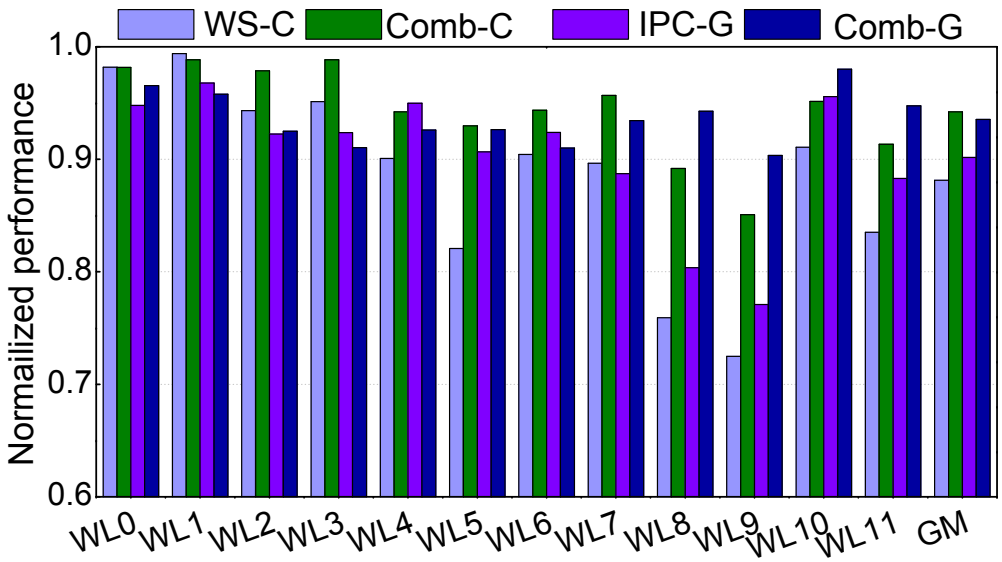

Figure 42: The performance of heterogeneous workloads.

speedup of CPU applications across twelve workloads is reduced by 11.9%; correspondingly, the IPC of GPU applications is 9.2% lower than that obtained by TEMP+TBAS running alone.

The effectiveness of TEMP or TBAS is constrained and the performance of CPU applications is degraded as both TEMP and TBAS bias the GPU applications: 1) TBAS expects consecutive thread blocks to access their physical pages in a limited span of rows. The physical addresses of the pages accessed by CPU applications, however, can mix with those of the pages accessed by GPU applications, breaking the expectation of TBAS and consequently, deteriorating the row locality of GPU applications. 2) On the other hand, even if TBAS successfully keeps the row locality of GPU applications, due to the high volume of memory accesses from GPU, the memory controller probably always prioritizes the memory accesses issued from GPU, suspending the memory accesses from CPU.

To address above problems, we can first divide each bank into two portions – one for CPU and the other for GPU. We reserve the rows with higher addresses in a bank for CPU and the ones with lower addresses for GPU. CPU or GPU are assigned with the new pages from corresponding address space. By doing so most pages of CPU and GPU can be physically

separated in a bank, allowing TBAS to keep the row locality of GPU applications when CPU applications are running simultaneously. Secondly, the memory controller is always set to promote the memory accesses from CPU, as proposed in [26]. CPU applications are usually delay-sensitive, unconditionally promoting the memory accesses from CPU against the ones from GPU can eliminate the risk of CPU-side memory access starvation. The combined two solutions reduce the performance loss by 6.1%/3.5% in CPU/GPU applications, denoted by Comb-C and Comb-G in Figure 42. And we can see that some workloads (e.g., `WL8` and `WL9`) including both CPU and GPU intensive applications benefit significantly from the integrated heterogeneous-aware thread batching.

The solution mentioned above are simple yet capable to keep the effectiveness of TeMP and TBAS for GPU applications while preventing considerable performance loss of CPU applications. We believe more sophisticated techniques can further well balance the CPU and GPU throughput [66], which however is beyond the interests of this paper and left in the future work.

5.6 SUMMARY

GPU suffers from the mismatching between thread-level parallelism and DRAM bandwidth. To improve the DRAM usage efficiency of GPU applications, we propose an integrated thread batching which is composed of TEMP and TBAS techniques: TEMP improves GPU memory access parallelism for massive multi-threaded applications by minimizing the memory access interweaving across SMs; and TBAS maximizes the row locality by elaborately prioritizing the execution of the threads. Heterogeneous-aware thread batching is also introduced to promise the effectiveness of thread batching when running heterogeneous workloads. Our results show that TEMP+TBAS can achieve up to 9.3% system performance improvement and 11.3% DRAM energy saving compared to the baseline employing CCWS. By using simple, existing solutions, heterogeneous-aware thread batching can still maintain 94.2% CPU and 93.5% GPU throughput, respectively, across twelve diverse heterogeneous workloads.

6.0 RELATED WORKS

6.1 POWER-EFFICIENT GPU REGISTER FILE

Improving power efficiency is one of the main focuses of the research and development of GPU in both industry and academia. For example, NVIDIA Kepler architecture triples the number of cores for lower shader frequency and includes the compilation support of energy-efficient warp scheduling for better performance per watt [46]. The traditional low-power techniques, e.g., DVFS [2] and power gating [2, 67], have been also utilized to reduce both dynamic and leakage power consumption of GPU. Special efforts [68, 69, 70] on hacking the microarchitecture or pipeline, *e.g.*, exploiting value structures during execution, are also proposed to address the power issue. Nonetheless, the systematic analysis [71, 2] on GPU power consumption showed that RF is one of the major factors affecting GPU power efficiency.

The power efficiency improvement on GPU RF can be also achieved through architecture optimizations. Gebhart *et al.* proposed RFC to minimize the RF accesses with two-level scheduling [4, 51] for both leakage and dynamic energy consumption reduction. The authors further introduced an unified on-chip memory combining L1 cache, shared memory and RF with significantly enhanced power-efficiency and performance [72]. Unlike RFC, our write buffer design piggybacks on the temporal locality within the pipeline execution (*i.e.*, WAW/RAW) and does not require any special scheduling policies.

Yu *et al.* first proposed using new memory technology, eDRAM to build GPU RF [73]. The authors designed a RF context aware scheduler to maintain the issue fairness of warps [73]. In our design, RMWS aims to minimize the shift delay of RM without putting special focus on the fairness. Very interesting, our simulations show that RMWS naturally carries

a good issue fairness so that its performance very close to GTO, a fair scheduler design. In [74] and [75], Jing *et al.* developed an opportunistic/compiler-assisted refreshing scheme to retain the data in volatile eDRAM cells. Due to the non-volatility of RM, such costly refreshing scheme can be safely removed in our proposed RF design to achieve significant standby power reduction.

Goswami *et al.* exploited the application of nonvolatile STT-RAM in GPU architecture as on-chip memory [76]. STT-RAM possesses near-zero leakage power as well as very high dynamic power, which is actually the main challenge in GPU RF designs. Hence, early write termination [77] is utilized to minimize the dynamic power consumption of STT-RAM based on-chip memory [76]. In our proposed RF design, the dynamic power is naturally reduced by the energy-efficient write mechanism of RM while the write buffer also filters the unnecessary RF accesses.

We note that traditional memory power management schemes can be also applied to reduce the RF power consumption. In [78], Abdel-Majeed *et al.* introduced drowsy RF design to save the leakage power consumed over a long time period between two successive accesses. The dynamic power of RF can be also minimized by masking the RF accesses from inactive threads within a warp. The architectural motivation of these solutions is orthogonal to our proposed techniques and can be incorporated together to further improve the energy efficiency of the GPU RF and its peripheral circuits.

6.2 DOMAIN WALL MOTION AND ITS APPLICATIONS

Domain wall motion is predicted by [79] and has drawn increasing attentions as a promising candidate for future storage [80] and logic devices technology [81]. The storage practice utilizing current driven domain wall motion, i.e., racetrack memory [82], has been fabricated with IBM 90nm technology [83] and widely studied as on-/off-chip memory components. Venkatesan *et al.* [45] first demonstrated using racetrack memory to build the last-level cache (LLC) on CPU. They also proposed several scheduling policies to process the cache requests. Sun *et al.* [34] proposed a very dense racetrack memory based LLC design where

multiple tracks are placed on top of each access port and the access ports are carefully placed to minimize the wasted area. Venkatesan *et al.* [33] further proposed domain-wall-shift-write based RM that has lower programming power; they then also introduced RM based cache hierarchy for GPU [84]. Following the similar design motivation, we use RM to build the GPU RF in our work. In a GPU, the generation of RF requests can be controlled by switching warps, leaving a natural optimization space of warp scheduling for shift overhead reduction in RMWS. Due to its low power and small area, RM has been continuously investigated in various platforms or application domains [85, 86, 87, 88, 89], demonstrating its potential being a rival replacement candidate of traditional memory technologies.

6.3 CACHE LOCALITY AWARE WARP SCHEDULING

There are many cache locality aware warp scheduling algorithms have been developed based on throttling warps or thread blocks/CTA in a SM [19, 20, 15, 21]. They usually keep a proper number of warps active to balance the thread-level parallelism and cache locality. Most of them, however, target only a single type of cache locality and lack a comprehensive consideration of diverse cache localities in GPUs.

For example, CCWS aims reducing the intra-warp L1 data cache misses. Generally, CCWS tries to keep a *proper* number of warps running without stressing the L1 data cache while fully using the capacity of it. It uses a *lost locality detector* to monitor the L1 data cache thrashing of each running warp. If a warp is detected as "losing locality", which means its cache blocks are frequently revisited and evicted by other warps, then the warp will be prioritized over all the other warps. The warps without losing locality will be suspended by the warp scheduler until the thrashing on L1 data cache is alleviated. Therefore, in CCWS, the thread-level parallelism is sacrificed for better intra-warp locality and higher system throughput.

CCWS requires non-trivial modifications of the caching system and introduces run-time computation cost, i.e., sorting all the running warps based on their lost locality scores in each schedule cycle. Hence, such types of locality-aware warp scheduling schemes are facing scalability challenges as the total number of the warps concurrently supported by a SM keeps ascending [13, 23].

Some other works [19, 20] have partially exploited inter-CTA locality in GPU applications, although the primary goal of them is to alleviate shared resources contention for better performance. As we have shown earlier, there are always more than one type of cache miss contributing to the total misses. Unfortunately, none of them, including CCWS, comprehensively takes into account the diverse cache localities in GPU application.

In this dissertation, we propose VWS that is capable of retaining intra-warp, intra-CTA, and inter-CTA localities to achieve a significantly improved system throughput. Moreover, in VWS, the decision making is much more computation and hardware efficient than existing works. As we already showed in Section 4.4, the lost cache locality is the main reason for the shared resources (e.g., interconnection network) contention. VWS can greatly mitigate the contention by enhancing diverse cache localities in GPU applications.

6.4 MEMORY PARTITIONING IN MULTI-CORE SYSTEMS

In multi-core systems, *memory bank partitioning* (MBP) binds a thread to one or more memory banks. Every thread accesses its own private banks to avoid the interference from other threads. Mi *et al.* [30] first proposed MBP and used modified bank permutation to compensate the degraded BLP. Jeong *et al.* [29] used sub-ranking to overcome the BLP degradation on single thread after applying MBP. Liu *et al.* [28] designed a purely software MBP based on OS page allocation. They also explored the utilization of MBP in a multi-threaded application but the result was not very promising because of the inter-thread data sharing. Xie *et al.* [31] pointed out that unbalanced memory requirements across the threads is the main reason of the BLP degradation and then proposed a dynamic bank partitioning approach to solve this problem. In TBMP, BLP is guaranteed by workload balancing across the SMs while the memory access fairness is guaranteed by the homogeneity of the GPU threads in a kernel. Thread batching in TEMP also alleviates the negative impact of inter-thread data sharing on system performance in multi-threaded applications.

6.5 DRAM EFFICIENCY IN GPU

Compiler-assisted data layout transformation [90, 91] proactively prevents unbalanced accesses to DRAM components by carefully reorganizing the array or thread block index. However, this method is not aware of any hardware implementation details, e.g., thread scheduling and DRAM address mapping, both of which may offset the efforts at compiler level. Some examples of hardware approaches for DRAM usage efficiency enhancement in GPU or CPU-GPU systems include:

Enhanced memory schedulers Jeong *et al.* [92] designed a QoS-aware memory scheduler for MPSoC with CPUs and GPUs. The DRAM bandwidth allocation between the CPUs and GPUs is dynamically adjusted to meet the frame rate requirement of GPUs and maximize the overall system throughput. Ausavarungnirun *et al.* [26] proposed a staged memory scheduling framework with affordable hardware cost for heterogeneous systems. We adopt the memory scheduling policy from [26] to customize our proposed heterogeneous-aware thread batching.

Enhanced thread scheduler Jog et al. [19] revealed that serial thread block data layout and sequential thread block dispatching can cause BLP degradation of GPU applications. A scheduler is then designed to improve the BLP by prioritizing different-numbered thread blocks in consecutive SMs. The authors also utilized prefetching to compensate the degradation of row locality. However, if the memory of GPU is pageable, the effect of prioritized thread scheduling will become uncertainty, because the pages of consecutive thread blocks can be nonconsecutive or not concentrated a DRAM row. In our scheme, TEMP relies on thread batching and page coloring to improve the BLP and TBAS enhances the row locality, targeting a heterogeneous system design supporting pageable GPU memory.

7.0 CONCLUSION

As the scaling of GPU towards to higher thread-level parallelism, the gap between the computational capability and memory subsystem will continuously enlarge. This dissertation focuses on the optimization on memory subsystem of GPU to alleviate the discrepancy between computation and memory. We investigate the inefficiency of the GPU memory subsystem vertically ranging from on-chip RF and L1 data cache to off-chip DRAM device memory, and come up with three solutions, each of which tackles one particular memory component. The components as well as the contributions of this dissertation are summarized as following:

- *A RM based RF is introduced to replace the SRAM based RF in GPU.* The traditional SRAM based RF digests a considerable amount of power budget of the whole GPU system because a large RF is dispensable for fast thread switching. RM is much better than SRAM in terms of power and area; however the location-related access mechanism of RM is the major impedance preventing the adoption of RM based RF. We first explain the bank-level design of RM and quantify its advantages compared to SRAM. We then propose two architecture-level optimizations, register remapping and RMWS, to address the drawback of prolonged access latency. We also consider the execution scenario where multiple warp schedulers are presented, for which we propose warp register remapping to ensure the applicability of RMWS. Experiment results show that, after employing RM based RF, the RF energy consumption is drastically reduced by 48.5% while the system performance is kept stable.
- *VWS is designed to explore diverse localities in GPU.* VWS is capable of preserving intra-warp, intra-CTA and inter-CTA L1 data cache locality. By limiting the number of

warp running on a SM, IWL tries to keep the aggregated working set in L1 data cache. IntraCL picks the warps from as less CTAs as possible to make sure the warps coming from the same CTA hit the intra-CTA shared data on cache. InterCL leverages serial thread block dispatching and sequential CTA switching to guarantee sequential CTA hit the inter-CTA shared data on cache. VWS improve the GPU performance by more than 9% w.r.t. the CCWS.

- *A architecture substrate – TEMP and TBAS – is proposed to reduce the performance and energy bottleneck raised from device DRAM memory.* The DRAM bandwidth is subject to the less rapid development of device manufacture compared to computing component, especially for GPU. Consequently, the DRAM subsystem becomes a key factor affecting the overall throughput of GPU. TEMP and TBAS can improve the efficient utilization of the pressure DRAM bandwidth by increasing the memory access parallelism and locality. Particularly, TEMP resolves the memory access inter-weaving between SMs and banks by batching thread blocks and page coloring; on top of TEMP, TBAS maximums the DRAM row buffer hit rate. The combined substrate improves the GPU performance by 10.3% and reduces the DRAM energy consumption by 11.3%.

To combine the above optimizations together for an integrated solution for high performance and energy-efficient GPU, one issue needs to be addressed is how to orchestra different goals which are optimizable via warp scheduling. Currently, each memory component optimization relies on a warp scheduler design which is tailored for a specific purpose: RMWS is designed for minimal RM shifting; VWS is for less L1 cache thrashing; and TBAS is for maximal DRAM locality. Fortunately, there do exist overlaps between different warp scheduler designs. For example, given the fact that TBAS also covers the DRAM locality, we may potentially combine both VWS and TBAS for both L1 data cache and DRAM locality enhancement: IWL in VWS provides the number of warps which must kept active in a SM, and TBAS operates on those warps accordingly. We leave the combination of the optimization techniques proposed here to the future works.

Another possible future work is automatic deployment of the GPU workload beyond a single GPU by further detailed investigation of thread-data mapping. Automatic GPU scaling is a problem that hinders GPU adoption in the real HPC domain, because a kernel

is assumed a uniform memory layout in a single device memory, arbitrarily distributing the workloads of a kernel to multiple GPUs, *i.e.*, multiple device memories may incur intolerable data synchronization. In this dissertation we already demonstrate that it is possible to separate the data sharing between thread blocks through the analysis of two major thread-data mappings in kernels. Therefore, we can use thread batch, but not thread block as the basic distributed element among the GPU pool. Since most data sharing is restricted within the thread batch, we may potentially restrict most of the data synchronization between GPUs. More detailed thread-data mapping analysis needs to be done and a profiler is wanted for automatic and accurate thread batch generation.

BIBLIOGRAPHY

[1] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software(ISPASS)*, 2009, pp. 163–174.

[2] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "Gpuwattch: Enabling energy optimizations in gpgpus," in *Proceedings of the 40th Annual International Symposium on Computer Architecture(ISCA)*, 2013, pp. 487–498.

[3] NVIDIA, "Geforece gtx 980 white paper," http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF/, 2015.

[4] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron, "Energy-efficient mechanisms for managing thread context in throughput processors," in *Proceedings of the 38th International Symposium on Computer Architecture(ISCA)*, 2011, pp. 235–246.

[5] G. Sun, X. Dong, Y. Xie, J. Li, and Y. Chen, "A novel architecture of the 3d stacked mram l2 cache for cmps," in *Proceedings of the 15th International Conference on High-Performance Computer Architecture(HPCA)*, 2009, pp. 239–249.

[6] X. Wu, J. Li, L. Zhang, E. Speight, R. Rajamony, and Y. Xie, "Hybrid cache architecture with disparate memory technologies," in *Proceedings of the 36th International Symposium on Computer Architecture(ISCA)*, 2009, pp. 34–45.

[7] M. Mao, H. H. Li, A. K. Jones, and Y. Chen, "Coordinating prefetching and stt-ram based last-level cache management for multicore systems," in *Proceedings of the 23rd ACM international conference on Great lakes symposium on VLSI*. ACM, 2013, pp. 55–60.

[8] M. Mao, G. Sun, Y. Li, A. K. Jones, and Y. Chen, "Prefetching techniques for stt-ram based last-level cache in cmp systems," in *Design Automation Conference (ASP-DAC), 2014 19th Asia and South Pacific*. IEEE, 2014, pp. 67–72.

[9] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, 2012.

[10] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the IEEE International Symposium on Workload Characterization(IISWC)*, 2009, pp. 44–54.

[11] NVIDIA, "Cuda sdk," https://developer.nvidia.com/cuda-downloads/.

[12] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP Laboratories*, pp. 22–31, 2009.

[13] NVIDIA, "Nvidia fermi architecture," http://www.nvidia.com/object/fermi-architecture.html/, 2009.

[14] W. Jia, K. A. Shaw, and M. Martonosi, "Characterizing and improving the use of demand-fetched caches in gpus," in *Proceedings of the 26th ACM international conference on Supercomputing (ICS)*, 2012, pp. 15–24.

[15] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-conscious wavefront scheduling," in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture(MICRO)*, 2012, pp. 72–83.

[16] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, 2010, pp. 60–71.

[17] NVIDIA, "Cuda," http://www.nvidia.com/object/cuda_home_new.html/.

[18] "Opencl." https://www.khronos.org/opencl/.

[19] A. Jog, O. Kayiran, N. Chidambaram Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Owl: cooperative thread array aware scheduling techniques for improving gpgpu performance," in *Proceedings of the 18th international conference on Architectural support for programming languages and operating systems(ASPLOS)*, 2013, pp. 395–406.

[20] M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu, "Improving gpgpu resource utilization through alternative thread block scheduling," in *HPCA*, 2014, pp. 260–271.

[21] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Divergence-aware warp scheduling," in *MICRO*, 2013, pp. 99–110.

[22] "Micron tn-ed-01: Gddr5 sgram introduction." http://www.micron.com/products/dram/gddr5/.

[23] "Nvidia maxwell architecture." https://developer.nvidia.com/maxwell-compute-architecture/.

[24] O. Mutlu and T. Moscibroda, "Stall-time fair memory access scheduling for chip multiprocessors," in *MICRO*, 2007, pp. 146–160.

[25] ——, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems," in *ISCA*, 2008, pp. 63–74.

[26] R. Ausavarungnirun, K.-W. Chang, L. Subramanian, G. Loh, and O. Mutlu, "Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems," in *ISCA*, 2012, pp. 416–427.

[27] G. L. Yuan, A. Bakhoda, and T. M. Aamodt, "Complexity effective memory access scheduling for many-core accelerator architectures," in *MICRO*, 2009, pp. 34–44.

[28] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu, "A software memory partition approach for eliminating bank-level interference in multicore systems," in *PACT*, 2012, pp. 367–376.

[29] M. K. Jeong, D. H. Yoon, D. Sunwoo, M. Sullivan, I. Lee, and M. Erez, "Balancing dram locality and parallelism in shared memory cmp systems," in *HPCA*, 2012, pp. 1–12.

[30] W. Mi, X. Feng, J. Xue, and Y. Jia, "Software-hardware cooperative dram bank partitioning for chip multiprocessors," in *NPC*, 2010, pp. 329–343.

[31] M. Xie, D. Tong, K. Huang, and X. Cheng, "Improving system throughput and fairness simultaneously in shared memory cmp systems via dynamic bank partitioning," in *HPCA*, 2014.

[32] M. Mao, W. Wen, Y. Zhang, Y. Chen, and H. H. Li, "Exploration of gpgpu register file architecture using domain-wall-shift-write based racetrack memory," in *DAC*, 2014, pp. 196:1–196:6.

[33] R. Venkatesan, M. Sharad, K. Roy, and A. Raghunathan, "Dwm-tapestri-an energy efficient all-spin cache using domain wall shift based writes," in *Proceedings of the Conference on Design, Automation and Test in Europe(DATE)*, 2013, pp. 1825–1830.

[34] Z. Sun, W. Wu, and H. H. Li, "Cross-layer racetrack memory design for ultra high density and low power consumption," in *Proceedings of the 50th Annual Design Automation Conference(DAC)*, 2013, p. 53.

[35] M. Mao, J. Hu, Y. Chen, and H. Li, "Vws: a versatile warp scheduler for exploring diverse cache localities of gpgpu applications," in *Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE*. IEEE, 2015, pp. 1–6.

[36] M. Mao, W. Wen, X. Liu, J. Hu, D. Wang, Y. Chen, and H. Li, "Temp: Thread batch enabled memory partitioning for gpu," *To appear in DAC*, 2016.

[37] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan, "Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems," in *HPCA*, 2008, pp. 367–378.

[38] W. W. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation and scheduling for efficient gpu control flow," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture(MICRO)*, 2007, pp. 407–420.

[39] S. Fukami, T. Suzuki, K. Nagahara, N. Ohshima, Y. Ozaki, S. Saito, R. Nebashi, N. Sakimura, H. Honjo, K. Mori *et al.*, "Low-current perpendicular domain wall motion cell for scalable high-speed mram," in *Proceedings of the 2009 Symposium on VLSI Technology*, 2009, pp. 230–231.

[40] M. Sharad, R. Venkatesan, A. Raghunathan, and K. Roy, "Multi-level magnetic ram using domain wall shift for energy-efficient, high-density caches," in *Proceedings of the International Symposium on Low Power Electronics and Design(ISLPED)*, 2013, pp. 64–69.

[41] M. Hosomi, H. Yamagishi, T. Yamamoto, K. Bessho, Y. Higo, K. Yamane, H. Yamada, M. Shoji, H. Hachino, C. Fukumoto *et al.*, "A novel nonvolatile memory with spin torque transfer magnetization switching: Spin-ram," in *Proceedings of the IEEE International Electron Devices Meeting(IEDM)*, 2005, pp. 459–462.

[42] S. S. Parkin, M. Hayashi, and L. Thomas, "Magnetic domain-wall racetrack memory," *Science*, vol. 320, no. 5873, pp. 190–194, 2008.

[43] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi, "Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 7, pp. 994–1007, 2012.

[44] S. Liu, J. E. Lindholm, M. Y. Siu, B. W. Coon, and S. F. Oberman, "Operand collector architecture," Nov. 16 2010, uS Patent 7,834,881.

[45] R. V. et al, "Tapecache: a high density, energy efficient cache based on domain wall memory," in *ISLPED*. ACM, 2012, pp. 185–190.

[46] NVIDIA, "Nvidia kepler architecture," http://www.nvidia.com/object/nvidia-kepler.html/, 2012.

[47] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving gpu performance via large warps and two-level warp scheduling," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture(MICRO)*, 2011, pp. 308–317.

[48] M. Abdel-Majeed, D. Wong, and M. Annavaram, "Warped gates: Gating aware scheduling and power gating for gpgpus," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture(MICRO)*, 2013.

[49] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Divergence-aware warp scheduling," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture(MICRO)*, 2013.

[50] S. Fukami, T. Suzuki, N. Ohshima, K. Nagahara, and N. Ishiwata, "Micromagnetic analysis of current driven domain wall motion in nanostrips with perpendicular magnetic anisotropy," *Journal of Applied Physics*, vol. 103, no. 7, pp. 07E718–07E718, 2008.

[51] M. Gebhart, S. W. Keckler, and W. J. Dally, "A compile-time managed multi-level register file hierarchy," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture(MICRO)*, 2011, pp. 465–476.

[52] J. Fix, A. Wilkes, and K. Skadron, "Accelerating braided b+ tree searches on a gpu with cuda," in *A4MMC*, 2011.

[53] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: a mapreduce framework on graphics processors," in *PACT*, 2008, pp. 260–269.

[54] P. ROGERS and C. FELLOW, "Amd heterogeneous uniform memory access," 2013.

[55] N. Agarwal, D. Nellans, M. Stephenson, M. O'Connor, and S. W. Keckler, "Page placement strategies for gpus within heterogeneous memory systems," in *ASPLOS*, 2015, pp. 607–618.

[56] A. Branover, D. Foley, and M. Steinman, "Amd fusion apu: Llano," *IEEE Micro*, vol. 32, no. 2, pp. 28–37, 2012.

[57] A. Bakhoda, J. Kim, and T. M. Aamodt, "Throughput-effective on-chip networks for manycore accelerators," in *MICRO*, 2010, pp. 421–432.

[58] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," pp. 190–200, 2005.

[59] J. D. Owens, W. J. Dally, S. Rixner, P. Mattson, and U. J. Kapasi, "Memory access scheduling," in *ISCA*, 2000, pp. 128–128.

[60] "Micron ddr3 sdram part mt41j512m4." Micron Technology Inc., 2006.

[61] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood, "gem5-gpu: A heterogeneous cpu-gpu simulator," *Computer Architecture Letters*, vol. 14, no. 1, pp. 34–36, 2015.

[62] AMD, "Amd quad-core a10-series apu for desktops," http://products.amd.com/en-us/DesktopAPUDetail.aspx?id=100/.

[63] J. Power, M. Hill, and D. Wood, "Supporting x86-64 address translation for 100s of gpu lanes," in *HPCA*, 2014.

[64] Micron, "Micron system power calculators." http://www.micron.com/products/support/power-calc/.

[65] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads," *IEEE micro*, vol. 28, no. 3, pp. 42–53, 2008.

[66] O. Kayıran, N. C. Nachiappan, A. Jog, R. Ausavarungnirun, M. T. Kandemir, G. H. Loh, O. Mutlu, and C. R. Das, "Managing gpu concurrency in heterogeneous architectures," 2014.

[67] P.-H. Wang, C.-L. Yang, Y.-M. Chen, and Y.-J. Cheng, "Power gating strategies on gpus," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 3, p. 13, 2011.

[68] S. Z. Gilani, N. S. Kim, and M. Schulte, "Exploiting gpu peak-power and performance tradeoffs through reduced effective pipeline latencies," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture(MICRO)*, 2013.

[69] S. Z. Gilani, N. S. Kim, and M. J. Schulte, "Power-efficient computing for compute-intensive gpgpu applications," in *Proceedings of the 19th IEEE International Symposium on High Performance Computer Architecture(HPCA)*, 2013, pp. 330–341.

[70] J. Kim, C. Torng, S. Srinath, D. Lockhart, and C. Batten, "Microarchitectural mechanisms to exploit value structure in simt architectures," in *Proceedings of the 40th Annual International Symposium on Computer Architecture(ISCA)*, 2013, pp. 130–141.

[71] S. Hong and H. Kim, "An integrated gpu power and performance model," in *Proceedings of the 37th International Symposium on Computer Architecture(ISCA)*, 2010, pp. 280–289.

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

[73] W.-K. S. Yu, R. C. Huang, S. Q. Xu, S.-E. Wang, E. Kan, and G. E. Suh, "Sram-dram hybrid memory with applications to efficient register files in fine-grained multi-threading," in *Proceedings of the 38th International Symposium on Computer Architecture(ISCA)*, 2011, pp. 247–258.

[74] N. Jing, Y. Shen, Y. Lu, S. Ganapathy, Z. Mao, M. Guo, R. Canal, and X. Liang, "An energy-efficient and scalable edram-based register file architecture for gpgpu," in *Proceedings of the 40th Annual International Symposium on Computer Architecture(ISCA)*, 2013, pp. 344–355.

[75] N. Jing, H. Liu, Y. Lu, and X. Liang, "Compiler assisted dynamic register file in gpgpu," in *Proceedings of the International Symposium on Low Power Electronics and Design(ISLPED)*, 2013, pp. 3–8.

[76] N. Goswami, B. Cao, and T. Li, "Power-performance co-optimization of throughput core architecture using resistive memory," in *Proceedings of the 19th IEEE International Symposium on High Performance Computer Architecture(HPCA)*, 2013, pp. 342–353.

[77] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "Energy reduction for stt-ram using early write termination," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design(ICCAD)*, 2009, pp. 264–268.

[78] M. Abdel-Majeed and M. Annavaram, "Warped register file: A power efficient register file for gpgpus." in *Proceedings of the 19th International Conference on High-Performance Computer Architecture(HPCA)*, 2013, pp. 412–423.

[79] L. Berger, "Exchange interaction between ferromagnetic domain wall and electric current in very thin metallic films," *Journal of Applied Physics*, vol. 55, no. 6, pp. 1954–1956, 1984.

[80] S. S. Parkin, "Shiftable magnetic shift register and method of using the same," Dec. 21 2004, uS Patent 6,834,005.

[81] D. A. A. et al, "Magnetic domain-wall logic," *Science*, vol. 309, no. 5741, pp. 1688–1692, 2005.

[82] L. T. et al, "Racetrack memory: a high-performance, low-cost, non-volatile memory based on magnetic domain walls," in *IEDM*. IEEE, 2011, pp. 24–2.

[83] A. J. A. et al, "Racetrack memory cell array with integrated magnetic tunnel junction readout," in *IEDM*. IEEE, 2011, pp. 24–3.

[84] R. Venkatesan, S. G. Ramasubramanian, S. Venkataramani, K. Roy, and A. Raghunathan, "Stag: Spintronic-tape architecture for gpgpu cache hierarchies," in *ISCA*, 2014, pp. 253–264.

[85] M. Moeng, H. Xu, R. Melhem, and A. K. Jones, "Contextprerf: Enhancing the performance and energy of gpus with nonuniform register access," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 1, pp. 343–347, 2015.

[86] D. Kline Jr, H. Xu, R. Melhem, and A. K. Jones, "Domain-wall memory buffer for low-energy nocs," in *Proceedings of the 52nd Annual Design Automation Conference*. ACM, 2015, p. 11.

[87] H. Xu, Y. Li, R. Melhem, and A. K. Jones, "Multilane racetrack caches: Improving efficiency through compression and independent shifting," in *Design Automation Conference (ASP-DAC), 2015 20th Asia and South Pacific*. IEEE, 2015, pp. 417–422.

[88] D. Kline Jr, H. Xu, R. Melhem, and A. K. Jones, "Fusedcache: A naturally inclusive, racetrack memory, dual-level private cache," vol. pp, no. 99. IEEE, 2016.

[89] J. Chung, K. Ramclam, J. Park, and S. Ghosh, "Domain wall memory based digital signal processors for area and energy-efficiency," in *Proceedings of the 52nd Annual Design Automation Conference*. ACM, 2015, p. 64.

[90] Y. Yang, P. Xiang, J. Kong, and H. Zhou, "A gpgpu compiler for memory optimization and parallelism management," in *PLDI*, 2010, pp. 86–97.

[91] I.-J. Sung, J. A. Stratton, and W.-M. W. Hwu, "Data layout transformation exploiting memory-level parallelism in structured grid many-core applications," in *PACT*, 2010, pp. 513–522.

[92] M. K. Jeong, M. Erez, C. Sudanthi, and N. Paver, "A qos-aware memory controller for dynamically balancing gpu and cpu bandwidth use in an mpsoc," in *DAC*, 2012, pp. 850–855.