TEMP: Thread Batch Enabled Memory Partitioning for GPU*

Mengjie Mao, Wujie Wen, Xiaoxiao Liu, Jingtong Hu[†], Danghui Wang[‡], Yiran Chen, Hai Li

Department of Electrical and Computer Engineering, University of Pittsburgh

[†]School of Electrical and Computer Engineering, Oklahoma State University

[‡]School of Computer Science, Northwestern Polytechnical University

{mem231, wuw2, yaz24, yic52, hal66}@pitt.edu [†]jth@okstate.edu [‡]wangdh@nwpu.edu.cn

ABSTRACT

As massive multi-threading in GPU imposes tremendous pressure on memory subsystems, efficient bandwidth utilization becomes a key factor affecting the GPU throughput. In this work, we propose *thread batch enabled memory partitioning* (TEMP), to improve GPU performance through the improvement of memory bandwidth utilization. In particular, TEMP clusters multiple thread blocks sharing the same set of pages into a thread batch and dispatches the entire thread batch to a stream multiprocessor. TEMP separates the memory access streams of different thread batches by OS memory management, preserving the intrinsic locality of thread batches and increasing the memory access parallelism. Experimental results show that TEMP can obtain up to 10.3% performance improvement and 14.6% DRAM energy reduction compared to a state-of-the-art scheduler without any memory-side optimizations.

1. 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 computational capability, GPU is widely employed in various platforms ranging from hand-held devices to HPC systems. GPU heavily relies on massive multi-threading to achieve high throughput. The massive multi-threading can amortize the memory access delay; on the other hand, it imposes tremendous pressure on memory subsystem.

Current GPU is usually attached with high bandwidth memories, *e.g.*, GDDR5 [2] or HBM [1] DRAM. However, those memories are bandwidth-optimized with the sacrifice of capacity and power efficiency. The power of high-end GPU can be 2×–3× higher than that of CPU with similar technology node, and DRAM power can contribute 20% to 30% of total GPU power [16]. As power becomes a primary concern of contemporary system [11], especially for data center, the memory subsystem becomes the major bottleneck limiting the scalability of GPU and/or heterogeneous system [15]. Moreover, with the ever-increased thread-level parallelism, GPU seems never satisfy with the given bandwidth [7]. Therefore, the optimization of bandwidth utilization is critical to maintain high-throughput, energy-efficient and scalable GPU.

To squeeze as much data as possible from the bandwidth

budget, *memory partitioning* is a viable approach used in CPU-centric systems. It attempts to divide and allocate non-overlapped memory resources to individual threads, each of which exclusively accesses its own memory partition, reducing the inter-thread interference [17]. Compared to other techniques like intelligent memory scheduling, memory partitioning requires much less hardware overhead because it mainly leverages OS memory management for implementation. Introducing memory partitioning to GPU is feasible and even attractive because: 1) The pageable unified memory address space in current heterogeneous systems allows OS to freely allocate the pages accessed by GPU; 2) The near-homogeneous multi-threading in GPU guarantees the memory access fairness and parallelism among partitions, if the workloads are evenly distributed to GPU cores called *stream multiprocessors* (SMs). This statement, however, may not be valid in multi-core systems due to the different bandwidth requirements of heterogeneous threads [23].

However, two observations prevent the direct deployment of memory partitioning in GPU: 1) The number of the threads in GPU could be several orders of magnitude greater than that in CPU. It is not realistic to assign memory resources in the degree of individual threads. 2) All the threads in a GPU application share an unified address space. It has not been learned if it is feasible to separate the memory access streams of GPU threads.

In this work, we propose *thread batch enabled memory partitioning* (TEMP), a memory partitioning solution which is tailored for better bandwidth utilization in GPU. TEMP allocates memory banks to SMs, and tends to direct the memory accesses from a SM to its bound banks¹. The thread blocks that share the same set of pages are in a *thread batch*; the thread batch is then dispatched as a whole to a SM. Through page coloring [17], the pages accessed by the thread batch can be placed to the banks linking to the SM where the thread batch runs. Consequently, the memory access interweaving between SMs and banks is minimized while the memory access locality and parallelism are maximized. Experimental results show that after applying TEMP, the performance of a set of GPU applications is improved by 10.3% and the DRAM energy consumption is reduced by 14.6%, w.r.t. a state-of-the-art baseline without memory partitioning.

2. PRELIMINARY

2.1 Heterogeneous CC-NUMA

The heterogeneous CPU-GPU integrated systems are evolving towards unified memory address space [20]. 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 [7]. 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

*This work is supported in part by NSF awards 1311706 and 1337198; Danghui Wang is supported by the NSFC (61472322) and Fundamental Research Funds for Central Universities of China (3102014JSJ0001)

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DAC '16, June 05 - 09 2016, Austin, TX, USA

Copyright is held by the owner/author(s). Publication rights licensed to ACM. ACM 978-1-4503-4236-0/16/06\$15.00

DOI: <http://dx.doi.org/10.1145/2897937.2898103>

¹We use memory bank and bank interchangeably.

address space. Figure 1 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 possible in the local memory. By using *local* policy we can avoid most bandwidth contentions between GPUs and CPUs in heterogeneous CC-NUMA.

GPU programming model such as CUDA defines the workload offloaded to GPU as a *kernel*. A kernel is highly multi-threaded where all the threads are encapsulated in a *grid*. Within a grid, all the threads are partitioned into thread blocks, each of which contains up to hundreds of threads. During execution, each thread block is dispatched as a whole to a SM. During execution, a thread block is further partitioned into fixed-size *warps* that are automatically scheduled by the *warp scheduler* and executed in SIMD fashion. The shared L2 cache is banked and placed in memory partitions. Each memory partition has its own memory channel.

2.2 DRAM Basics

Despite various DRAM interfaces targeting at various aspects, the DRAM primitives are similar regardless of the interfaces. Figure 2(a) depicts the basic organization of a DRAM channel. A channel may be comprised of multiple DIMMs. Within each DIMM, there can be one or more ranks depending on the particular interface. A rank consists of multiple DRAM devices. A GDDR5 DRAM device contains 8 or 16 banks. The data of each bank are always pre-loaded to its private row buffer before being accessed.

The address mapping shown in Figure 2(b) is used as the default DRAM address mapping [9] used in this work. The length of each field in the address depends on the DRAM interface and the capacity the interface supports. The address mapping in Figure 2(c) is used for the page coloring mechanism. If the number of page offset bits is no greater than the sum of column and byte offset bits, page coloring can place a GPU page to any arbitrary channel, rank, bank, or row.

Efficient DRAM bandwidth utilization is mainly determined by *bank-level parallelism* (BLP) and *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 need to perform through the row buffer. Memory access latency and energy can be reduced

Figure 2: (a) The DRAM channel organization; (b) The default DRAM address mapping; (c) DRAM address mapping for page coloring (shadow fields are the color bits).

if the accesses hit on the row buffer, because no row activation is needed. In this work, we attempt to improve the DRAM bandwidth utilization in GPU by minimizing the inter-SM interference of memory accesses, which is the root reason of low BLP and RBHR of DRAM accesses [13].

2.3 Related Works

In multi-core systems, *memory bank partitioning* binds a thread to one or more memory banks. Every thread accesses its own private banks to avoid the interference from other threads. Jeong *et al.* [13] used sub-ranking to overcome the BLP degradation on single thread after applying memory bank partitioning. Liu *et al.* [17] designed a pure software memory bank partitioning based on page coloring. They also explored the utilization of memory bank partitioning in a multi-threaded application but the result was not very promising because of the inter-thread data sharing. Xie *et al.* [23] 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 the problem. Unlike the multi-program environment where each thread exclusively accesses its address space, the SIMT of GPU allows thousands of threads to share a unified address space. Accordingly, TEMP mainly focuses on resolving the memory interweaving of GPU threads.

Compiler-assisted data layout transformation [24] proactively prevents unbalanced accesses to DRAM components by carefully reorganizing the data or thread block indices. However, this method is not aware of any hardware details, *e.g.*, warp scheduling and DRAM address mapping, both of which may offset the efforts paid by compiler. Some examples of hardware approaches for enhanced DRAM bandwidth utilization in GPU or heterogeneous CPU-GPU systems include better memory schedulers [8] or warp scheduler [14]. Instead of thorough hardware redesign, TEMP relies on software/hardware co-design, including flexible OS memory management and a low-overhead hardware augmentation for DRAM efficiency.

3. TEMP

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 3.1) and *dispatches* them to the same SM (Section 3.2) so as to minimize the inter-SM inter-

Figure 1: Organization of a heterogeneous CC-NUMA system.

```

__global__ void Mapper(char* inputKeys, char* inputVals, int4*
    inputOffsetSizes, int recordsPerThread ...) {
    // calculate the global block and thread ID
    int bid = gridDim.y * blockDim.x + threadIdx.y;
    int tid = threadIdx.x;
    int index = bid * blockDim.x + threadIdx.x;
    // a linear, continuous 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 3: Annotated code snippet of Mapper kernel in Mars library.

ference 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.

3.1 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 blocks². The first major type of thread-data mappings is: the data accessed by each thread block is clustered over a sequential address space. Figure 3 shows the skeleton of the **Mapper** kernel in MapReduce engine of Mars [12]. 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**.

Figure 4 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 4(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 place the pages accessed by a SM to its bound banks, as depicted in Figure 4(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 4(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 4(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 4(e).

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 5 shows the code snippet of the **cenenergy** kernel in the CUTCP benchmark [22]. 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 indices. All threads form a 2D grid which is further tiled with 2D thread blocks. Figure 6 demonstrates a simplified thread-data mapping in this 2D grid. The thread organization and accessed data matrix can be found in Figure 6(a) and (b), respectively. Here we again assume one grid with four thread blocks and each thread block has four

Figure 4: (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) with or (e) with thread batching.

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 6(c) gives an thread batching example where every matrix row in Figure 6(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.

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 4(c) and 6(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 7 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. On average, 96% of kernels achieve the minimum cross-batch page sharing through a fixed thread block stride. Some kernels in LBM cannot fitted with a

```

__global__ void cenenergy(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 5: Annotated code snippet of cenenergy kernel in CUTCP.

²In 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 1).

Figure 6: (a) A 2D grid – 2D thread blocks; (b) The accessed data matrix; (c) The memory access footprint with thread batching.

simple stride 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.

3.2 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. We profile some GPU applications in Table 1 to measure the accumulated percentage of the pages shared by different sizes of consecutive thread batches. The profile results is shown in Figure 8. Horizontal axis shows the maximal distance of the thread batches sharing pages. Among all the accessed pages, more than 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, 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 [6], 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

Figure 7: The distribution of thread block stride.

Figure 8: The accumulated percentage of page sharing between thread batches.

block according to the received thread block id. The 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 [18], 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.

4. EXPERIMENT METHODOLOGY

We adopt a set of diverse GPU applications from [5, 10, 12, 22]. The characteristics of these applications are summarized in Table 1. All applications are divided into three categories and will be explained in Section 5.1. 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 GPGPU-Sim [9], and its simulation parameters are listed in Table 2. The page size is set to 4KB. To avoid the bottleneck of GPU TLB and expose the limitations of DRAM bandwidth, we optimize the GPU TLB design including per-SM TLB, highly-

Table 1: The characteristics of GPU applications. The used thread hierarchy labeled as (grid dimension, block dimension).

Application	Abbr.	Category	Hierarchy
InvertedIndex	II	C1	(1D, 1D)
Kmeans Clustering	KM	C1	(1D/2D, 1D)
PageViewCount	PVC	C1	(1D, 1D)
Fast Walsh Transform	FWT	C1	(1D, 1D)
Coulombic Potential	CUTCP	C1	(2D, 2D)
Seven Point Stencil	STEN	C1	(2D, 2D)
Back Propagation	BP	C1	(1D, 2D)
Discrete Cosine Transform	DCT	C1	(2D, 2D)
Scalar Products	SP	C1	(1D, 1D)
Hotspot	HOT	C2	(2D, 2D)
Sorting Networks	SN	C2	(1D, 1D)
Lattice-Boltzmann Method	LBM	C3	(2D, 1D)
LIBOR Monte Carlo	LIB	C3	(1D, 1D)

Table 2: Simulation configuration.

Number of SMs	15
SM Clock	700MHz
L1 Data Cache	16KB
Warp Size	32
Max Number of Threads	1536/SM
Max Thread Blocks	8/SM
TLB	64-entry L1, 8KB page walk cache, 512-entry shared L2
System interconnection	100 GPU cycles
GDDR5 1.5GB	
# of memory channels	6
Memory controller	FR-FCFS, open-page
Banks	16
L2 cache	128KB/channel
Timing	from [2]
DDR3 4GB	
# of memory channels	2
Memory controller	FR-FCFS, open-page
Banks	8
L2 cache	1MB/channel
Timing	from [3]

threaded PTW and shared L2 TLB [19]. Each GDDR5 DRAM chip has 16 banks grouped in 4 4-bank groups. Page coloring binds a SM to one to two bank groups. The DRAM model in GPGPU-Sim is modified to support the heterogeneous memory setting and a fixed 100 GPU cycles [7] is applied when GPU requests a remote page from DDR3 DRAM. We estimate the GDDR5 DRAM energy consumption through a modified MICRON DRAM power calculator [4] based on the datasheet [2]; the DDR3 DRAM energy consumption is directly obtained from MICRON DRAM power calculator by feeding the runtime statistic generated from GPGPU-Sim.

5. RESULTS

5.1 Performance

We choose the state-of-the-art *cache-concise-wavefront-scheduler* (CCWS) [21], combined with our proposed TEMP (denoted as TEMP) for effectiveness and adaptivity demonstration. Figure 9 shows the performance of different configurations. CCWS serves as the baseline, all the results are normalized to CCWS that does not have any optimization on DRAM subsystem. TEMP introduces 10.3% speedup compared to CCWS. CCWS is designed for improving the L1 cache locality in GPU. It captures the intra-warp locality by decreasing the L1 cache thrashing. Indeed, the potential of TEMP is hidden by CCWS, because CCWS filters a high volume of DRAM accesses in some cache-sensitive applications. We still select CCWS as the baseline for ruling out the concern that our simulation configuration favors TEMP.

We also compare TEMP to OWL [14]. OWL targets on cache performance through intelligent warp scheduling. It

Figure 9: The performance of OWL, TEMP, BW-AWARE, and TEMP+BW-AWARE.

Figure 10: The local access ratio of memory accesses.

also tries to improve the BLP of memory accesses by prioritizing different-numbered thread blocks in consecutive SMs. From our evaluation, OWL is 92.4% 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 19.4% better than OWL.

Figure 10 shows the local access ratio of all the memory accesses, including local and remote accesses. Here *local access* denotes the memory access from a SM to its bound banks while *remote access* denotes the access from non-associated SMs. To better understand the respective efficacy of TEMP, we divide 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 $> 99.6\%$) and obtain substantial performance improvement.
- **C2:** Similar to C1, the applications in C2 also demonstrate high local access rate ($> 93.9\%$). Introducing TEMP to these applications, however, results in slight performance degradation ($\sim 1\%$).
- **C3:** The applications in C3 cannot obtain very high local access ratio due to irregular thread-data mapping. An overall performance lost is observed from this category.

We summarize the DRAM statistic as well as the *stall on reply network* from memory partitions to SMs in Figure 11. All the values are normalized to CCWS. The BLP of TEMP is significantly improved by 57.4% in C1, while the RBHR is increased by 18.8%. 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 13.3% BLP and 9.7% 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 30.6% 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 18.2% reduc-

Figure 11: The statistic of DRAM subsystem.

Figure 12: The normalized DRAM energy consumption of different configurations.

tion in DRAM access delay with TEMP in C1. Consequently, TEMP bursts 16.6% performance, 24.2% higher than OWL in C1.

With TEMP, C2 achieves a high local access rate. However, it is hard to further increase its BLP because it approaches the theoretical upper bound. For instance, some kernels in H0T have only a few thread blocks: the number is even lower than the bank count. Applying TEMP does not gain much BLP improvement. Note that it is difficult to formalize the thread-data mapping of the applications in C3. TEMP is not able to reduce inter-SM interference of memory accesses, and serial thread block dispatching enlarges the network congestion. Thus, applying TEMP aggravates their DRAM access delay.

We also consider a synergistic page placement policy, *bandwidth-aware* (BW-AWARE) [7] 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 9 we can see BW-AWARE gains 5.7% extra performance on top of CCWS. The performance gain is compliance to the value reported in [7] given the similar bandwidth ratio. TEMP 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. As a result, TEMP+BW-AWARE has better BLP and RBHR compared to BW-AWARE, and further improve the performance by 15.2% compared to CCWS.

5.2 Energy

The normalized DRAM energy consumption of all configurations is shown in Figure 12. 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 compared to CCWS, the DRAM energy savings of TEMP is 14.6%. OWL saves less DRAM energy, 5.7% because the higher row activation plus inferior performance. TEMP+BW-AWARE achieves the most energy saving by 17.3%. No energy is saved in C3 with TEMP, the RBHR as well as the DRAM activate energy keeps stable. TEMP does not successfully separate the inter-SM memory access streams in these applications in C3.

6. CONCLUSION

To optimize the utilization of DRAM bandwidth in GPU, we propose TEMP which retains the memory access locality and parallelism. Based on the extensive application profiling, we found that it is possible to form regular thread batches based on two dominant thread-data mappings. TEMP dispatches the thread blocks serially and colors the pages in SM-level. Most memory accesses from a SM can be directed to the bound banks. Our results show that TEMP can effectively resolve inter-SM interference of memory accesses, achieving up to 10.3% system performance improvement and 14.6% DRAM energy saving.

7. REFERENCES

- [1] “High bandwidth memory (hbm) dram.” <http://www.jedec.org/standards-documents/results/jesd235/>.
- [2] “Hynix 1gb (32mx32) gddr5 sgram h5gq1h24afr,” http://www.hynix.com/datasheet/eng/graphics/details/graphics_26.H5GQ1H24AFR.jsp/.
- [3] “Micron ddr3 sdram part mt41j256m8.” <http://www.micron.com/products/dram/ddr3-sdram/>.
- [4] “Micron system power calculators.” <http://www.micron.com/products/support/power-calc/>.
- [5] “Nvidia cuda sdk.” <https://developer.nvidia.com/cuda-downloads/>.
- [6] “Nvidia fermi architecture.” <http://www.nvidia.com/object/fermi-architecture.html/>.
- [7] N. Agarwal *et al.*, “Page placement strategies for gpus within heterogeneous memory systems,” in *ASPLOS*, 2015, pp. 607–618.
- [8] R. Ausavarungnirun *et al.*, “Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems,” in *ISCA*, 2012, pp. 416–427.
- [9] A. Bakhoda *et al.*, “Analyzing cuda workloads using a detailed gpu simulator,” in *ISPASS*, 2009, pp. 163–174.
- [10] S. Che *et al.*, “Rodinia: A benchmark suite for heterogeneous computing,” in *IISWC*, 2009, pp. 44–54.
- [11] H. Esmailzadeh *et al.*, “Power challenges may end the multicore era,” *Communications of the ACM*, vol. 56, no. 2, pp. 93–102, 2013.
- [12] B. He *et al.*, “Mars: a mapreduce framework on graphics processors,” in *PACT*, 2008, pp. 260–269.
- [13] M. K. Jeong *et al.*, “Balancing dram locality and parallelism in shared memory cmp systems,” in *HPCA*, 2012, pp. 1–12.
- [14] A. Jog *et al.*, “Owl: cooperative thread array aware scheduling techniques for improving gpgpu performance,” in *ASPLOS*, 2013, pp. 395–406.
- [15] S. W. Keckler *et al.*, “Gpus and the future of parallel computing,” *IEEE Micro*, vol. 31, no. 5, pp. 7–17, 2011.
- [16] J. Leng *et al.*, “Gpuwatch: enabling energy optimizations in gpgpus,” in *ISCA*, 2013, pp. 487–498.
- [17] L. Liu *et al.*, “A software memory partition approach for eliminating bank-level interference in multicore systems,” in *PACT*, 2012, pp. 367–376.
- [18] M. Mao *et al.*, “Vws: a versatile warp scheduler for exploring diverse cache localities of gpgpu applications,” in *DAC*, 2015, pp. 1–6.
- [19] J. Power, M. Hill, and D. Wood, “Supporting x86-64 address translation for 100s of gpu lanes,” in *HPCA*, 2014, pp. 568–578.
- [20] P. Rogers and C. Fellow, “Amd heterogeneous uniform memory access,” 2013.
- [21] T. G. Rogers, M. O’Connor, and T. M. Aamodt, “Cache-conscious wavefront scheduling,” in *MICRO*, 2012, pp. 72–83.
- [22] J. A. Stratton *et al.*, “Parboil: A revised benchmark suite for scientific and commercial throughput computing,” *Center for Reliable and High-Performance Computing*, 2012.
- [23] M. Xie *et al.*, “Improving system throughput and fairness simultaneously in shared memory cmp systems via dynamic bank partitioning,” in *HPCA*, 2014, pp. 344–355.
- [24] Y. Yang *et al.*, “A gpgpu compiler for memory optimization and parallelism management,” in *PLDI*, 2010, pp. 86–97.