Power Efficient Sharing-Aware GPU Data Management

Abdulaziz Tabbakh
Electrical Engineering Department
University of Southern California
Los Angeles, USA
tabbakh@usc.edu

Murali Annavaram
Electrical Engineering Department
University of Southern California
Los Angeles, USA
annavara@usc.edu

Xuehai Qian
Electrical Engineering Department
University of Southern California
Los Angeles, USA
xuehai.qian@usc.edu

***Abstract*—The power consumed by memory system in GPUs is a significant fraction of the total chip power. As thread level parallelism increases, GPUs are likely to stress cache and memory bandwidth even more, thereby exacerbating power consumption. We observe that neighboring concurrent thread arrays (CTAs) within GPU applications share considerable amount of data. However, the default GPU scheduling policy spreads these CTAs to different streaming multiprocessor cores (SM) in a round-robin fashion. Since each SM has a private L1 cache, the shared data among CTAs are replicated across L1 caches of different SMs. Data replication reduces the effective L1 cache size which in turn increases the data movement and power consumption.**

The goal of this paper is to reduce data movement and increase effective cache space in GPUs. We propose a sharing-aware CTA scheduler that attempts to assign CTAs with data sharing to the same SM to reduce redundant storage of data in private L1 caches across SMs. We further enhance the scheduler with a sharing-aware cache allocation and replacement policy. The sharing-aware cache management approach dynamically classifies private and shared data. Private blocks are given higher priority to stay longer in L1 cache, and shared blocks are given higher priority to stay longer in L2 cache. Essentially, this approach increases the lifetime of shared blocks and private blocks in different cache levels. The experimental results show that the proposed scheme reduces the off-chip traffic by 19% which translates to an average DRAM power reduction of 10% and performance improvement of 7%.

***Keywords*-GPU Cache Management; Data Sharing; Thread Block Scheduling**

I. Introduction

General purpose computation on graphics processing units (GPUs) is increasingly popular as they are considered a power efficient approach for achieving high throughput. To support complex memory access patterns of general purpose applications, GPUs typically have a multi-level memory hierarchy consisting of a private L1 cache per streaming multiprocessor core (SM), a shared L2 cache connected to all SMs through an interconnection network, and a high bandwidth banked DRAM. Even with such a hierarchy, general purpose applications on GPUs experience significant memory access bottlenecks [1], [2]. Each SM within a GPU can execute thousands of threads simultaneously which limits the available private L1 cache space per thread to around 10 bytes; for instance 16KB L1 cache per SM that can run up to 1536 threads in some recent GPUs. Due to the massive multithreading in GPUs, even a reasonably small per-thread working set will result in premature eviction of useful data, and cache thrashing [2]. To understand the efficiency of GPU memory hierarchy, we conducted experiments to classify the cache misses (details later). Our results show that only 19% of L2 cache misses are compulsory misses. This percentage dropped to less than 9% in L1 caches.

Given that only a small fraction of cache misses are compulsory misses, there is a considerable room for improving cache efficiency. In principle, the cache efficiency could be improved by pinning live cache blocks in cache so that they can be reused before eviction. However, the total size of live blocks may exceed the cache capacity. Cache bypassing is another approach that may be used when the pinned cache blocks reach the cache capacity. While these techniques are intuitively effective, the real challenge is *how to guide cache allocation and bypassing policies in the context of GPUs* to fully derive the benefits.

In this paper, we first make a motivational observation that GPU applications share data across neighboring cooperative thread arrays (CTAs). For load balancing purposes GPUs spread these CTAs across multiple SMs which results in data replication in private L1 cache. We introduce a sharing-aware CTA scheduler design, which takes advantage of data sharing across CTAs and assigns sharers to one SM as much as possible while taking into account the load balance among SMs. This approach ensures that shared data does not get replicated across multiple L1 caches which effectively improves L1 cache size. To further reduce replication impact we propose to place only a single copy of shared data in L2 cache and let the shared data bypass L1 cache, which in turn could increase the life time of active private blocks. We bring the two ideas under a unified *sharing-aware* cache management framework for GPUs. We propose a simple mechanism to classify private and shared data dynamically and propose a cache insertion and replacement policy which enables private and shared data to stay longer in private L1 cache and shared L2 cache, respectively. Using these two innovations, the sharing-aware cache management framework increases the effective size of caches, reduces premature block eviction and increases the cache block lifetime. We implemented our proposed design in a cycle-accurate simulator and showed that our design can reduce the off-chip traffic by 19%, reduce DRAM power consumption by 10% while improving the overall performance

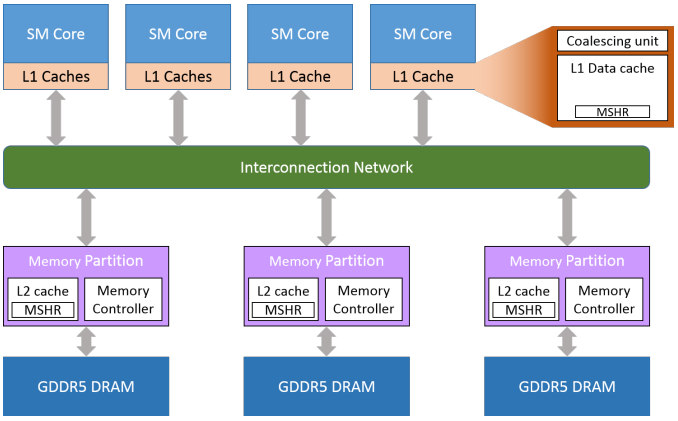

Fig. 1. Memory System Architecture in GPUs

by 7%.

The rest of this paper is organized as follows. An overview of the GPU architecture, as pertinent to this work, is presented in Section II. Section III presents the data sharing patterns across CTAs, which motivates our proposed designs. The proposed sharing-aware CTA scheduler is described in section IV. Then, we propose and discuss our sharing-aware cache management scheme in Section V. Section VI discusses the hardware implementation of the proposed techniques. Simulation setup and results are presented in Section VII, which is followed by a discussion about related work in Section VIII. Section IX concludes the paper.

II. Baseline GPU Architecture

In this section, we describe our baseline GPU architecture and modeling assumptions. Figure 1 shows the organization of GPU architecture we modeled, which is similar to modern GPUs from NVIDIA and AMD.

A. Execution Model

GPU applications consist of kernels. Each kernel consists of blocks of threads, called *cooperative thread arrays (CTA)*, or *work group*, which is considered the basic unit of work that can be assigned to a *streaming multiprocessor (SM)* core. Each CTA is split further into subgroups called *warps* that are executed in a lockstep fashion. The maximum number of CTAs that can be assigned to an SM is limited by the resource availability, such as the total number of threads, size of shared memory, and size of register file. Thus the number of CTAs that may be assigned to a single SM varies between kernels depending on per CTA resource demand. After calculating the maximum number of CTAs that can be accommodated per SM, current GPUs use CTA schedulers to determine how to assign CTAs to SMs. On NVIDIA GPUs [3], a GigaThread Engine is responsible for CTA scheduling. However, there is limited public disclosure about how this engine works. But prior works have stated that the CTA scheduler is a round-robin (RR) CTA scheduler where CTAs are assigned to each SM in a round-robin manner [4], [5]. Figure 2 shows how RR works. In this approach CTA 1 is assigned to SM1, CTA 2 is assigned to SM2, and so on, until all SMs are assigned

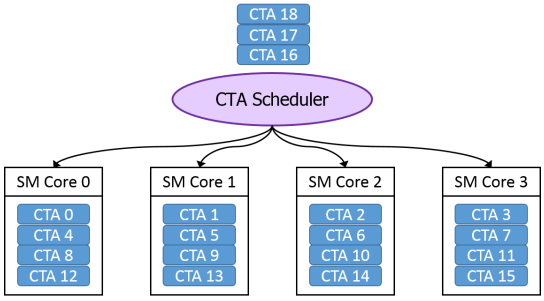

Fig. 2. Round-Robin CTA Scheduler

a CTA. The same procedure is repeated until each SM is assigned the maximum number of CTAs. After this initial allocation, new CTAs are allocated after an existing CTA finishes execution. This scheduling scheme guarantees fair load distribution between SMs and also maximizes utilization of a single SM.

B. Memory Hierarchy

GPU memory hierarchy consists of register file, L1 caches, L2 cache, and off-chip GDDR DRAM [6], [3]. L1 caches are private per SM and can be accessed by all warps in that core. L2 cache is shared between all cores and can be accessed by any thread in the kernel. L2 cache is partitioned into multiple banks and each bank is typically connected to a separate DRAM channel and a portion of the DRAM memory. The cores are connected to the L2 cache banks by an interconnection network. Each L2 cache bank can cache the blocks that are fetched from the DRAM channel connected to it [6], [3]. In both cache levels, Miss Holding Status Registers (MSHRs) record pending misses.

C. Caches Hardware and Policies

GPU L2 cache uses write-back with write allocation policy for last level cache (LLC). L2 cache evicts the victim block upon a miss in order to free and reserve space for the miss request. The space is then marked as RESERVED for the requested block until it is serviced by the main memory. The status of the block is then changed to VALID, or MODIFIED if the block is involved in an atomic operation.

GPU L1 data cache adopts *ON_FILL* allocation policy where the cache line allocation happens upon receiving the refilled data. Hence, while the miss is being serviced the victim cache line can continue to be accessed [7]. L1 cache adopts a write-through with write allocation for global data and write-back for local data. Note that GPU kernels explicitly tag data as either local or global. Both types of data may be cached in L1 cache but local data is accessible only by one thread. Global data is accessible to all threads in the kernel. Global data is accessed by a thread using the *ld.global* instruction, while local data is accessed using the *ld.local* instruction. Using write-through for global data allows other cores to observe the most recently updated data through shared L2 cache while there is no need to share the updated local data since they can be accessed by threads within a CTA only, which are all co-located on the same SM.

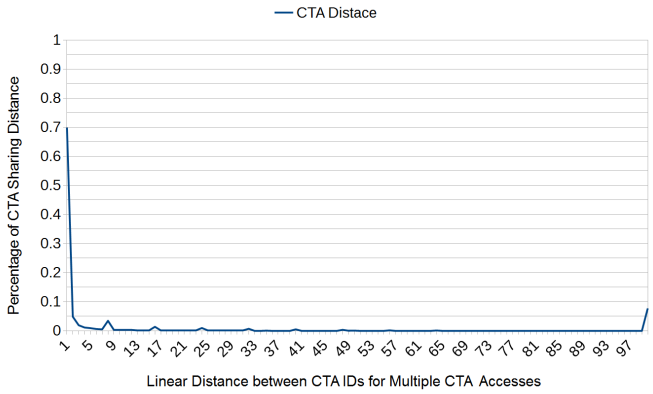

Fig. 3. CTA Distance for Shared Data Blocks

III. Data Sharing in GPUs

A. Inter-CTAs Data Sharing

In this section we present the motivational data to quantify how much data is shared across CTAs within a kernel. We use a linearized CTA ID to identify each CTA in this study. The linear CTA ID is calculated as: *CtaId = BlockId.x + (BlockId.y × GridDim.x) + (BlockId.z × GridDim.y × GridDim.x)*. Figure 3 shows inter-CTA data sharing. The Y-axis shows the fraction of total data that is accessed by the kernel that is shared by at least two CTAs. The X-axis shows the distance between the two sharer CTAs, in terms of the linearized CTA ID difference. The data shows that around 70% of data sharing occurs between *neighboring CTAs*; in particular, sharing is prevalent amongst CTAs that have a linear distance of less than five. These results present the average computed over a wide range of GPU benchmarks (details presented later). This data shows that sharing is prevalent across CTAs and most of the data sharing occurs within a short CTA distance.

To understand the reasons behind data sharing, we analyzed the CUDA source code for several benchmarks. The reason for the vast amount of sharing is that many load addresses are calculated from parametrized data such as thread ids, constant parameters loaded by $ld.param$ such as block dimensions and grid dimensions, or other constants passed during kernel initialization. Figure 4 shows prominent data access segments from three different benchmarks where the addresses of loads or stores are determined by constants, and thread ids, and CTA ids only. In the code segment from *NN*, the array ($Layer4\_Neurons\_GPU$) is indexed using $blockIdx.y$ only which means that neighboring CTAs that have consecutive values of $blockIdx.x$ will access the same memory location. In the *Backprop* benchmark, the array $input\_cuda$ is indexed using $index\_in$ which is computed using only $blockIdx.y$ and hence all threads that belong to the same $blockIdx.x$ will use the same index to access the array. In *LUD*, the varaible $global\_row\_id$ used to index array m is computed in a similar way to $index\_in$ in *Backprop*.

B. Inter-SM Data Sharing

While inter-CTA data sharing is a function of the kernel code, it does not necessarily mean that these CTAs also share data using the L1 cache. If these CTAs are allocated to different SMs then inter-CTA data sharing transforms into inter-SM data sharing. Data is considered shared across different SMs when it is being accessed by threads that are executed on different SMs. Inter-SM data sharing is the main culprit for creating redundant copies of data across multiple private caches. Figure 5 shows CDF of the sharing behavior of cached blocked between SMs in a sample of the evaluated benchmarks. The figure shows how inter-CTA data sharing between neighboring CTA is manifested into inter-SM sharing between multiple SMs using RR CTA scheduler. Our analysis shows that 60% of data blocks are in fact shared across SMs. Recall that 70% of the data is shared across CTAs and unfortunately a vast majority of this sharing translates into inter-SM data sharing, which in turn causes unnecessary data duplication in the private L1 cache of each SM. In fact, each shared data block is accessed on average by 2.41 SMs.

```
...
for (int i=0; i<100; ++i ){
  result+=Layer4_Neurons_GPU[i+(100*blockIdx.y)]*
    Layer4_Weights_GPU[weightBegin+i];
}
...
Layer5_Neurons_GPU[blockIdx.x+(10*blockIdx.y)]=result;
...
```

(a). NN

```
...
int index_in = HEIGHT * blockIdx.y + 1;
...
input_node[threadIdx.y] = input_cuda[index_in] ;
...
hidden_partial_sum[blockIdx.y * hid + threadIdx.y] =
    weight_matrix[threadIdx.x][threadIdx.y];
...
```

(b). backprop

```
...
int global_row_id = offset + (blockIdx.y+1)*BLOCK_SIZE;
int global_col_id = offset + (blockIdx.x+1)*BLOCK_SIZE;

peri_row[threadIdx.y][threadIdx.x] =
    m[(offset+threadIdx.y)*matrix_dim+global_col_id+threadIdx.x];
peri_col[threadIdx.y][threadIdx.x] =
    m[(global_row_id+threadIdx.y)*matrix_dim+offset+threadIdx.x];
...
  m[(global_row_id+threadIdx.y)*matrix_dim+global_col_id+threadIdx.x]
-= sum;
...
```

(c). LUD

Fig. 4. Sample CUDA Codes from Three Different Benchmarks.

IV. Sharing-Aware CTA Scheduler

Even though sharing is prevalent among CTAs, as shown in Section III-A the conventional round-robin CTA scheduler perturbs this sharing by assigning consecutive CTAs to different SMs. In order to exploit the inter-CTA data sharing, we propose a sharing-aware CTA scheduler. The proposed scheduler splits CTAs into groups, each group has N consecutive CTAs as identified by the linearized CTA id, and assigns each group for execution on a specific SM. Figure 6 shows the proposed CTA-scheduler. Instead of round-robin allocation on a per-CTA basis, the allocation is done at a

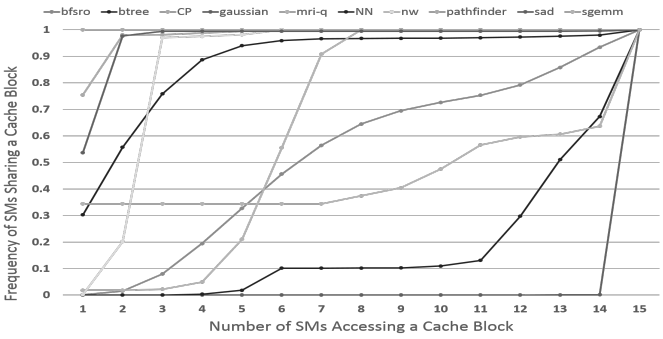

Fig. 5. The CDF of Shared L2 Data blocks vs. Number of Sharers

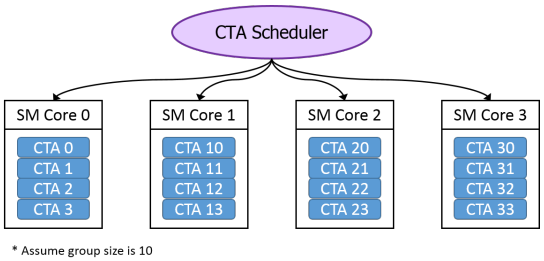

Fig. 6. Proposed CTA scheduler

coarse granularity of CTA groups. The scheduler maintains the current and end pointer for each CTA group. The scheduler starts by issuing the maximum allowed CTAs to each SM from the CTA group assigned to that SM. When one of the assigned CTAs completes its execution then the scheduler picks the next CTA from the same group. Hence, the current pointer in the CTA group for each SM is advanced by 1 after each CTA assignment until the current pointer reaches the end of the CTA group. When the CTA-pointer reaches the end of the CTA group, then a new CTA group is allocated for that SM. This CTA-assignment policy assures that neighboring CTAs are assigned to the same SM thereby enabling shared data to be brought into just one L1 cache.

The group size (N) must be chosen based on the trade-off analysis between the benefits of the CTA scheduling scheme and the potential negative effects of load imbalance and resource under utilization. When the group size is too large then the execution time of slowest group will determine the overall kernel execution time. If the group size is too small then neighboring CTAs may span different groups and these discontinuities at the end of each group lead to data replication across different SMs. We propose to set the group size depending on kernel grid size ($gridDim$) since memory addresses are mostly based on the grid size specification. Following the CTA linearized ID used in our analysis in III-A, we will use $gridDim.x$ as the size of the CTA assignment group. Thus each group is as large as the X-dimensionality of the grid. With this group size choice, all CTAs in the group have the same $blockIDx.y$ and their linearized IDs differ by 1.

If the number of CTA groups is not a multiple of the number of SMs in the GPU, some SMs will be assigned more groups

```
1. __global__ void kernel2(float *m_cuda, float *a_cuda, float *b_cuda,int Size,
   int j1, int t){
2.          if(threadIdx.x + blockIdx.x * blockDim.x >= Size-1-t) return;
3.          if(threadIdx.y + blockIdx.y * blockDim.y >= Size-t) return;

4.          int xidx = blockIdx.x * blockDim.x + threadIdx.x;
5.          int yidx = blockIdx.y * blockDim.y + threadIdx.y;

6.          a_cuda[Size*(xidx+1+t)+(yidx+t)] -= m_cuda[Size*(xidx+1+t)+t] *
   a_cuda[Size*t+(yidx+t)];
7.          if(yidx == 0){
8.                  b_cuda[xidx+1+t] -= m_cuda[Size*(xidx+1+t)+(yidx+t)] *
   b_cuda[t];
9.          }
10.}
```

Fig. 7. Code sample of (Gaussian) Benchmark

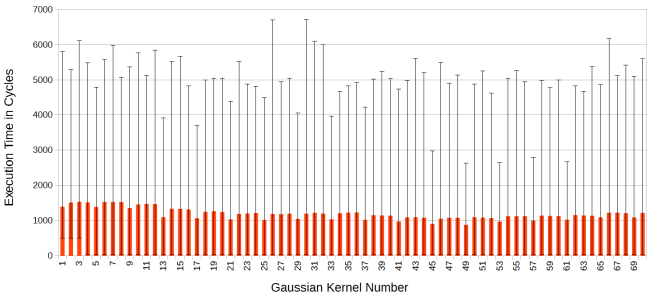

Fig. 8. Execution Time variation of CTAs for 70 Kernels in Gaussian Benchmark

than the others and hence there is a possibility that those SMs with more CTA groups will become the execution bottleneck. For example, assume a 16-SM GPU, and a kernel with 17 CTA groups. According to the proposed CTA scheduler, all SMs will be assigned 1 CTA group except one SM that will get 2. The one SM with 2 CTA groups will determine the overall kernel execution time; note that the basic assumption in this argument is that the execution time of each CTA group is roughly the same. But there is another possible source of workload imbalance, namely the unequal amount of work performed by different threads in a CTA. The number of instructions (memory, ALU and SFU instructions) executed per thread may vary because of divergence. Figure 7 shows a code snippet from the *Gaussian* benchmark as an example of load imbalance due to divergence. The thread that does not pass the first two conditional statements does not execute any instructions (i.e. neither memory operations nor ALU/SFU instructions).

Figure 8 shows an example of the execution time variation between CTAs in different kernels of *Gaussian* benchmark. It shows the average, maximum, and minimum execution time of CTAs in 70 different kernels in Gaussian benchmark. There are multiple CTAs that take longer to execute within each kernel. However, further analysis showed that these slowest executing CTAs are equally dispersed amongst all the CTA groups. As a result, even though there are variations in CTA execution time the execution time of a CTA group remained roughly the same. Hence, in general our approach of assigning a group of consecutive CTAs (delineated by their X-dimension of the input grid) does not lead to load imbalance.

However, in order to avoid any workload imbalance issue that may arise in exceptional cases, we propose a modification to the sharing-aware CTA scheduler. The modified scheduler

switches back to the conventional RR CTA scheduler towards the end of kernel execution. In particular, the scheduler assigns CTA groups to SMs until the number of remaining CTA groups is less than the number of SMs. At that time it switches to the conventional RR CTA scheduler. The RR CTA scheduler used at the end of kernel execution reduces the impact of load imbalance. It normalizes the amount of work executed by all the SMs as it assigns fewer CTAs to an SM that takes longer to execute a specific CTA.

V. Sharing-Aware GPU Cache Management

The previous section described an approach to increase the probability that two CTAs that share data may be assigned to the same SM. But in cases where the neighboring CTAs span different CTA groups, the problem of data replication still persists. We still need a mechanism to improve L1 cache efficiency by avoiding data replication. In this section, we will describe a sharing-aware cache management scheme that achieves this goal.

GPU threads can access data from multiple memory spaces during their execution. In particular, each thread has its own private local memory space, while all threads in a kernel may access the same global memory space. Both the local and global memory spaces may be cached in L1 and L2 caches. Our goal is to separate the local memory, which is by definition private data, from global memory, some of which may be shared across CTAs.

A. The Need for Better Cache Management

GPU caches do not enforce inclusion. In fact, NVIDIA GPU caches adopt non-inclusive, non-exclusive caches, meaning that cache blocks cached in L1 are also cached in L2 but eviction of a block in L2 cache does not necessary cause an eviction of all of its copies in L1 caches [8]. This scheme causes possible data redundancy between L1 and L2 cache. Figure 9 shows an example of redundancy. In this example, block *A* is only requested by *SM3* and has two copies: one in *SM*'s L1 cache and another one in L2 cache. The copy in L2 cache will not be accessed unless the copy in L1 is evicted. On the other hand, block *B* has five copies: four in L1 caches (due to inter-CTA data sharing where CTAs are spread across SMs) and one copy in L2 cache. We can eliminate this duplication by either moving all the sharers to one SM or making all the SMs access a single copy of that block. Our proposed sharing-aware CTA scheduler already targets moving sharers to one SM. We need to further eliminate any replication when sharers are assigned to different SMs.

In our example, if we are able to eliminate all the replicated copies, we would be able to increase the effective cache size by $3.5\times$ (i.e. use only 2 cachelines to cache these data block instead of using 7 cachelines). Avoiding cache block duplication and managing block placement policies can help improve cache performance and increase the effective cache size. Since cache access latency (L1 and L2 caches) is much faster than main memory access latency, increasing the effective cache size, and hence the number of distinct data blocks cached, would reduce the average access latency, increase the cumulative cache hit rate, and therefore reduce the main memory traffic.

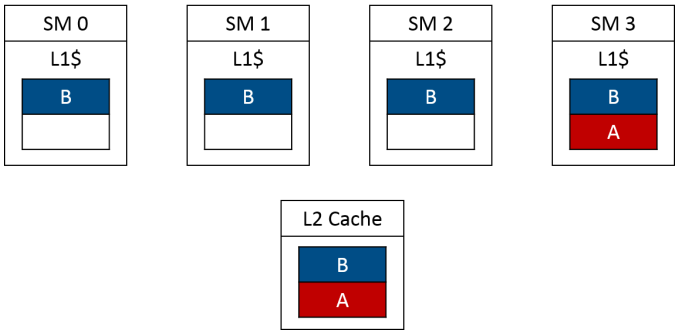

Fig. 9. Cache Block Placement in GPU Memory System

B. Cache Management Policies

We propose data sharing-aware cache management scheme guided by the principle of avoiding cache block duplication.

To design such a cache management scheme, we rely on classifying private and shared cache blocks. We define private and shared blocks as follows:

Private blocks are defined as those blocks that are accessed by a thread or multiple threads that are executing on the same SM.

Shared blocks are blocks accessed by multiple threads that are executing on different SMs. It is important to note that private blocks do not mean that they accessed by a single thread. Even if a block is shared by multiple CTAs, if all the CTAs execute on a single SM then that block is classified as private under our definition. Figure 9 shows an example of shared and private cache blocks (block *A* is private while block *B* is shared).

Our goal is to *maximize* the chance that *private L1 caches keep private blocks while shared L2 caches keep shared blocks*. Specifically, private blocks are given higher priority to stay longer in L1 cache and shared blocks are given higher priority to stay longer in L2 cache. Moreover, at the cache block insertion time, we avoid polluting L1 caches with shared data in order to reduce the premature eviction of private data. Similarly, at the L2 cache level we allow private data to bypass L2 thereby increasing the opportunity for shared data to stay longer in L2 cache. We achieve this goal using the following set of enhancements to cache allocation and replacement policies:

1) Tracking Data Sharing: As stated earlier, CUDA uses the notion of local and global memory to control data visibility across threads. The cache blocks in local memory space are known to be private to a single thread and will not be accessed by another SM. We consider local cache blocks as private as defined by the CUDA programming model. Since local memory accesses use a different instruction pneumonic (*ld.local*) versus global memory access (*ld.global*), it is easy to classify the cache block accessed through *ld.local* as private. In contrast, global cache blocks can be accessed by all threads but if a global block is accessed by threads residing only on a

single SM then that block is treated as private. Thus we need a mechanism to track how many SMs are sharing global cache blocks.

Cache block sharing could be tracked at the L2 level at low cost. When an L2 block is accessed by an SM, we simply tag that block as being accessed by that SM. In the future, if a different SM accesses that block then that block can be marked as shared (detailed implementation described in the next section). Note that on an L2 cache miss, an MSHR entry is allocated for the L2 miss. The fact that an MSHR entry is allocated in L2 on a cache miss ensures that even when a L2 cache miss request is being serviced from memory, the L2 cache can still identify when more than one SM requests the same cache block.

2) Sharing-Aware Cache Allocation Policy: Our cache block allocation and replacement policies ensure two important properties:

P1. Only private data can cause private data evictions in L1 cache.

P2. Only possible shared data can cause shared data evictions in L2 cache.

Any block identified as shared block is only inserted in private L1 if the victim block in L1 is invalid or if the victim block itself is another shared block. Thus a shared block does not cause any private data evictions from L1 cache. In reality only rarely L1 cache blocks are invalid and hence vast majority of the shared blocks stay only in L2 cache.

For private blocks, the priority is to place them in private L1 cache. They are only inserted in L2 if the victim block is invalid or if the victim block itself is another private block. Thus private data does not cause a shared block eviction from L2 cache.

The precise allocation and replacement policies are shown in the flow chart in Figure 10. The flow chart is triggered on a load instruction executed by an SM. If the load uses *ld.local* instruction then it is unambiguously classified as a private block. If the block is a hit in L1 then the data is provided and no further action is taken. If the block misses in L2 then the block is fetched from memory and it bypasses the L2 cache entirely, since this data is guaranteed to be private and accessed only by the requesting SM. In the rare case if the block hits in L2 then the block is fetched from L2 to L1 and no further action is taken.

If the load uses *ld.global* instruction then that block may or may not be private depending on whether other SMs access the block. First if the block is a hit in L1, data is provided to the SM and no further action is taken. If the block hits in L2, then the L2 cache checks whether the requesting SM is different than the SM that originally brought the cache block into L2. If so, that block is marked as shared. And the block is bypassed from the L1 cache of the requesting SM. If the block misses in L2 it is fetched from main memory. The block is placed in L2 cache and is tentatively marked as private and it is then delivered to L1. The assumption is that on L2 miss the cache block is treated as *most likely* private but because of the lack of complete information we do keep the block in L2.

Note that in the flow chart above bypassing L2 cache means that no L2 cache resources (i.e. victim cache line or MSHR entry) are allocated for the request. The request is sent to main memory and has the ID of the requesting SM. With this information, the response from memory is directly sent from DRAM to interconnection network without placing the data into L2 cache. Similarly, bypassing L1 cache means no L1 cache resources are allocated to the request. Instead the data is directly delivered to the destination register of the load instruction within the SM.

3) Sharing-Aware Cache Replacement Policy: The flow chart above describes the cache allocation policy. We now describe the replacement policy. We modify the basic *least recently used (LRU)* in L1 and L2 cache to design a sharing-aware replacement scheme. For this purpose we divide L1 and L2 cache sets into two logical sub-sets. One sub-set holds all the private blocks and the other sub-set holds the data marked as shared. Replacement policy in L2 cache favors replacing private blocks over shared ones. When a replacement request reaches L2 cache controller, the cache blocks in the designated set are checked. The least recently used private block is picked as a replacement candidate. If there are no private blocks (i.e. all cache blocks in the set are shared), the least recently used shared block is picked. This policy extends the lifetime of shared blocks in L2 cache by prioritizing the replacement of private blocks over shared ones.

On the other hand, replacement policy in L1 caches favors replacing shared blocks over private ones. Upon receiving a replacement request, L1 cache controller checks cache blocks in the designated set. The least recently used shared cache block is picked first as a replacement candidate and if there are no shared blocks, then the least recently used private block is picked. Figure 11 shows the mechanism to choose the replacement candidate in L1 cache. It also includes the dead block detection mechanism described later in Section VI-B (in the dotted box). L2 caches use similar mechanism except it does not include the dead block detection mechanism but L2 cache swaps private and shared block handling.

VI. Implementation and Discussion

A. Microarchitecture Implementation

To implement the sharing tracking mechanism, each block in L2 cache is augmented with a 4-bit owner SM field ($ownerId$), indicating the ID of the SM that triggered the first L2 miss and initiates the memory request. Each L2 cache block also keeps a 1-bit sharing flag (SF) (initialized to 0) to indicate whether the block is shared. On a L2 cache hit the requesting SM ID is checked against the $ownerId$ field to determine if a block is shared or private. The SF bit is set when they do not match.

To support the replacement algorithm in L1 cache, each L1 cache block keeps a 1-bit flag (*private flag (PF)*) which is set if the block is deemed private. Figure 13 highlights the additional cache tag bits needed for our design.

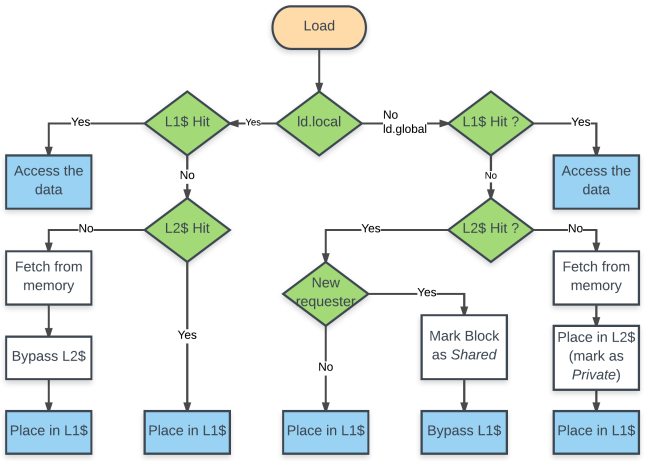

Fig. 10. The Flowchart of The Proposed Allocation Mechanism in Caches

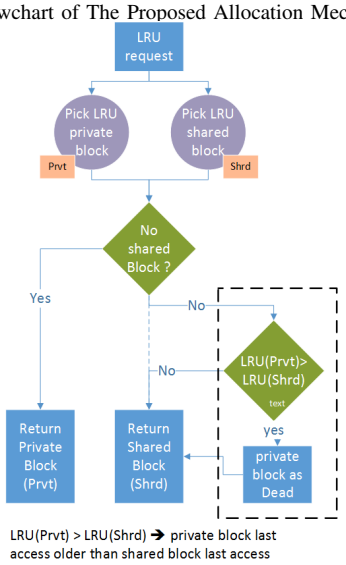

Fig. 11. Proposed Replacement Candidate Selection Flowchart for L1 Cache

B. Preventing Dead Blocks

Prioritizing the replacement decision by sharing information may lead to dead blocks. Dead blocks are blocks that are no longer used by any current threads but are not replaced.

In L2 cache, the modified replacement policy does not cause dead blocks. All global cache blocks are placed in L2 cache conservatively assuming they may be shared, therefore any cache block could become eviction candidate when it is the least recently used and all other blocks in that set are either reserved for incoming block or shared.

In L1 cache, since the victim selection decision is made when the block is filled, a private block can become a dead block and may still not get evicted, particularly when the cache is not highly contended. In order to detect dead private blocks in L1 cache and evict them, we use the LRU pointer. If the LRU pointer points to a private block, the block is declared *dead* but the next shared block in LRU order is picked. Figure 12 shows an example for dead block detection and resolution in L1 cache. In the example, blocks B and D are

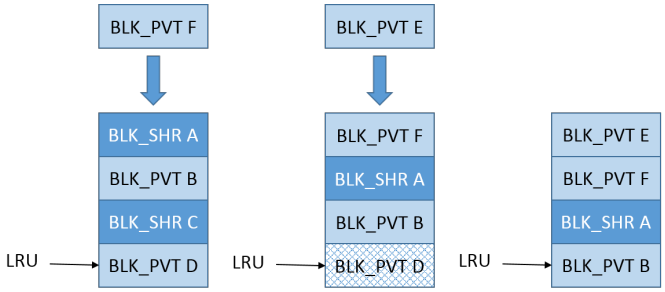

Fig. 12. Cache Freezing Issue Example

private whereas blocks A and C are shared. The LRU pointer points to block D which is private so block D is declared dead but block C is replaced instead. When block E is being filled the block D is again the least recently used and it has already been declared dead by a prior access, but it was given one more opportunity to stay. Since that opportunity window has passed block D is replaced next. A 1-bit flag (*dead block flag (DBF)*) is used to indicate dead data blocks in L1 cache in order to replace them in subsequent cache blocks fill operations.

C. L1 Cache Bypassing and Performance

According to our policy, a shared cache block may bypass L1 cache and is only placed into L2 cache. This approach may seem counter-intuitive since shared blocks may take longer to access. In the following, we explain the reason why our approach improves performance.

Better data placement on different cache levels increases the effective cache size which means that more cache blocks can be present in the cache at the same time. This approach allows more memory requests to be service by either L1 or L2 cache. Even though this may increase the access latency of accesses that are serviced by L2 cache, it reduces the average access latency and the cumulative miss rate of L1 and L2 caches. Moreover, our analysis shows that more than 44% of memory requests to a shared cache block hits in the MSHR. It means that these accesses comes in a short window between requesting the block from lower level of memory system until the block is serviced. Such behavior makes it possible to provide similar performance without letting shared blocks occupy L1 cache space: the close-by requests could be served directly from MSHR and bypass L1 cache. On the other hand, our analysis also shows that nearly 97% of the future accesses to a shared block occur at least a 1000 cycles after that block is filled. It means that even if we allocate L1 cache space for a shared block, when accessed again in future, it is more likely to have been replaced already in the long intervening time window.

VII. Evaluation

We use GPGPU-SIM v3.2.2 [7] to model and evaluate our data-sharing-aware CTA scheduler and cache management scheme. We use GPUWattch [9] to estimate and compare the power and energy consumption of our design. The simulation configuration is shown in table I. The baseline machine has exactly the same configuration except it uses LRU replacement

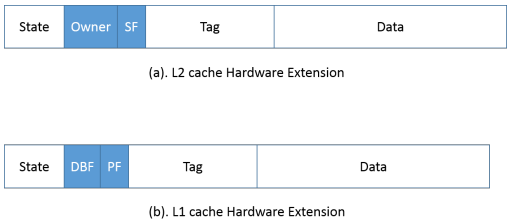

Fig. 13. Hardware Extension Needed for Our Design

TABLE I
GPGPU-SIM CONFIGURATION

Simulation Configuration	
Number of Cores	15
Core Configuration	32 SIMT lanes, 1.4GHz, GTO Warp Scheduler
L1 D-Cache per SM	16KB, 4-way assoc, 128B block
L2 Unified Cache	768KB total, 128KB/channel 8-way assoc, 128B block
Instruction Cache	2KB, 4-way assoc, 128B block, LRU
Texure Cache	12KB, 24-way assoc, 128B block, LRU
Constant Cache	8KB, 2-way assoc, 64B block, LRU
Registers/Core	32768
Interconnection Configuration	2D mesh, 1.4GHz, 32B channel width
DRAM Model	FR-FCFS (32 queue/channel) 6MCs, Channel BW=8B/cycle,
GDDR5 Timing	tCL=12, tRP=12, tRC=40, RAS=28, tRCD=12, tRRD=6

policy for both L1 data cache and L2 unified cache. In addition, it uses a conventional round-robin CTA scheduler to assign CTAs to cores. We selected 20 benchmarks from Parboil [10], Rodinia [11], and ISPASS-2009 [7] benchmark suits.

A. CTA Scheduler Evaluation

This section shows the performance of our proposed CTA scheduler. In order to provide a thorough evaluation, we evaluate three different CTA group sizes: 8 (*q8*), 16 (*q16*), and $gridDim.x$ (*qx*). These schedulers are evaluated with and without the workload balancing optimization where a round-robin scheduling is used towards the end of kernel execution. Figure 14 shows the normalized execution time and the average number of cores accessing a cache block. In the figure, *q8, q16, qx* represent the schedulers with group sizes of 8,16, and $gridDim.x$ without the round robin scheduling optimization, whereas *q8_adpt, q16_adpt, qx_adpt* represents the schedulers with group sizes of 8, 16, and $gridDim.x$ with a special round robin scheduling optimization at the end, respectively. Figure 14 shows that the best execution time can be achieved by *qx_adpt* while the lowest number of sharers per cache block is achieved by *qx*; and *qx_adpt* comes second. The figure also shows that round robin scheduling when only a few CTA groups are left to execute does improve performance by slightly increasing load balance at the cost of some redundant copying of data.

We also implemented the Block CTA Scheduler (BCS) proposed by Lee *et al.* [4]. BCS schedules sequential set of CTAs for each SM during kernel initiation and then it schedules CTA in pairs. It delays CTA scheduling until there

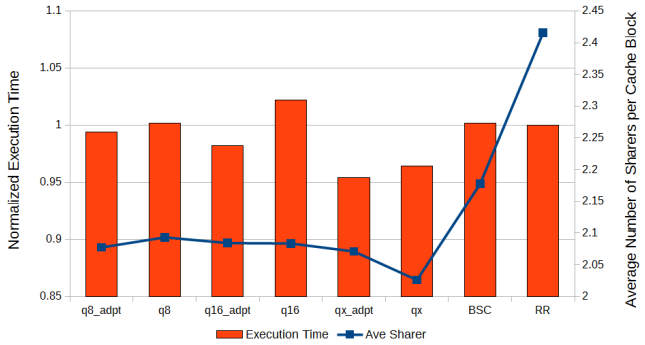

Fig. 14. Effect of CTA Scheduler Queue size

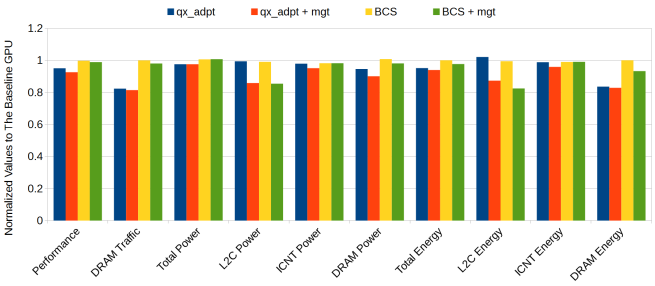

Fig. 15. Execution Time, Power and Energy consumption between the proposed CTA scheduler and BCS with and without cache management technique

are two available CTA contexts available in the SM and then schedules two CTAs at a time. BCS is similar to round-robin scheduler except that it schedules CTAs at a granularity of 2 CTAs at a time. The execution time and number of sharers data is shown under the label BCS in the figure.

Figure 15 shows a more detailed comparison between our CTA scheduler and BCS. The figure shows the average improvements along multiple dimensions over all the 20 benchmarks normalized to our baseline GPU. Across a wide range of metrics, such as performance, power, energy, interconnect (ICNT) power, sharing-aware CTA scheduler outperforms BCS. Overall, sharing-aware CTA scheduler achieves better cache block locality and outperforms BCS by 7%. In fact for a majority of the presented metrics BCS is no better than the baseline GPU. We also incorporated our cache allocation and replacement policies on top of BCS. This data is presented under the label $BCS + mgt$. Our allocation and replacement policies do in fact improve the power and performance of baseline BCS scheduler. Thus our cache allocation and replacement policies can also be applied independently on top of other CTA schedulers to improve performance.

B. Cache Allocation and Replacement Policy Evaluation

Our evaluation shows that our new cache management scheme with CTA scheduling group size of (*gridDim.x*) is able to enhance the performance of both L1 and L2 caches. We evaluate the performance of L1 cache in terms of miss rate and the performance of L2 cache in terms of the misses-per-thousand-instructions (MPKI). Figure 16 shows the normalized values of both L1 miss rate and L2 MPKI along with the

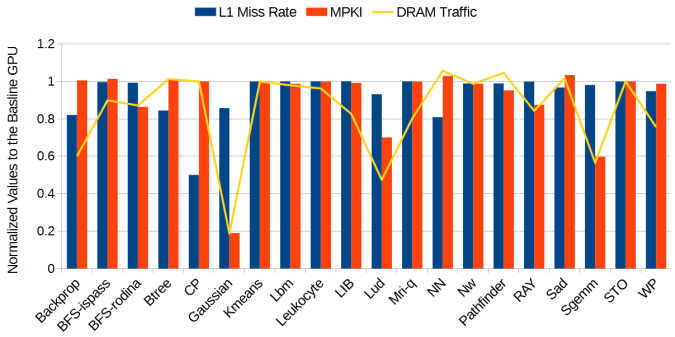

Fig. 16. Normalized L1 Miss Rate, L2 MPKI, and DRAM Traffic with *qx_adpt* Scheduler

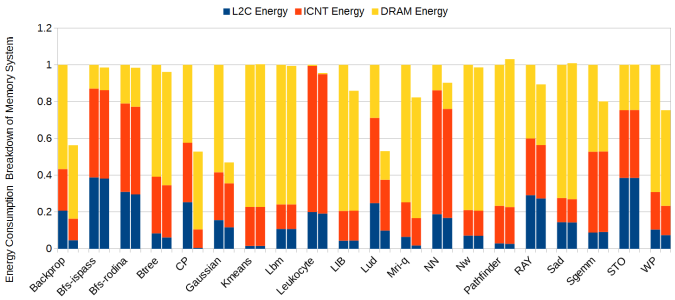

Fig. 17. Normalized Energy Consumption of L2 cache, NoC, and DRAM with *qx_adpt* Scheduler

DRAM traffic, normalized to the baseline GPU. The miss rate of L1 data caches has dropped by 13% over baseline. MSHR hits increased by 11% for shared block L1 cache misses which indicates that more warps/threads access the same block within a small time window while the miss is being serviced. MSHR hits reduce the interconnection network bandwidth since the GPUs do not send multiple requests for the same block/address to the lower level cache. L2 cache performance also improved as the number of misses per thousand instruction (MPKI) in L2 cache dropped by 47% compared to the baseline GPU. Also the non-compulsory misses (capacity and conflict misses) in L2 cache are reduced from 81% to 76%. The improved cache performance reduces the off-chip traffic by 19% causing the DRAM energy to go down by 17%. The overall energy of GPU is reduced by 6% and the combined energy consumed by L2 cache, interconnection network, and DRAM is reduced by 14%. The overall execution time of evaluated benchmarks has improved by 7% over the baseline machine.

Figure 17 shows the breakdown of the energy consumed by L2 cache, interconnection network and DRAM normalized to the baseline GPU energy consumption. Based on our results, these three components are accounted for roughly 35% of the total energy consumed. The figure shows that our scheme is able to reduce the average energy consumed by these components by 10%. It also shows that our scheme reduces the energy of the L2 cache (as in Backprop, CP, and LUD for example), the energy of interconnection network (as in CP, NN, and LUD), and the DRAM energy (as in Gaussian, WP, and RAY).

C. Kepler Simulations

Newer GPU architectures like Kepler and Maxwell have larger L2 cache (2MB in Kepler and Maxwell). To study the impact of larger L2 caches we simulated our design on a Kepler-like configuration. Simulation results show that our design can save up to 8% of DRAM energy, 8% of interconnect energy, 22% of L2 cache energy while enhancing the overall performance by 3%. The DRAM traffic is reduced by 10%. Even with the increase in L2 cache size the proposed scheme still sustains most of the befits, since the working set of many workloads still does not fit the larger L2 cache. Hence, as long as the dataset grows faster than cache size we expect that the benefits of the proposed scheme will continue to grow.

VIII. Related Work

A. CTA Scheduling and Management

To improve the performance of GPUs and their memory subsystems, various warp schedulers have been proposed to reduce memory latency. Cache-conscious warp scheduler (CCWS) [1] and cooperative thread array aware warp scheduler (OWL) [12] are examples of these schedulers. OWL prioritizes warps from a set of CTAs in order to increase the data locality and avoid bank conflicts. In CCWS, the warp scheduler controls the number of warps that are allowed to be scheduled to improve the performance of L1 cache. *Victim tag array* is used to collect the *lost locality score* which indicates the severity of inter-warp contention. DAWS [13] is a modification to CCWS where it uses cache foot-print prediction. When warps lose locality due to contention, the scheduler suspends some warps. Kayiran *et al.* [14] proposed controlling the TLP by changing the number of CTAs assigned to a core. When the application shows memory-intensive behavior, the number of CTAs is lowered in order to reduce cache, memory, and network contention. On the other hand, when the application is in a computationally intensive phase, the number of CTAs is increased to exploit more TLP. They also proposed *CTA pausing* where the warps belonging to the most recently assigned CTA are deprioritized when the optimal number of CTAs per core is changed at runtime. CTA-Core assignment policy is not changed in any of these designs. Lee *et al.* [4] proposed a CTA and warp schedulers that aim to exploit inter-CTA locality. The block CTA scheduler (BCS) assigns a block of sequential CTAs to the same core. After that, it uses another scheduler (lazy CTA scheduler) that determines the optimal number of CTAs per core to boost the performance and it assigns CTAs to SMs in pairs.

B. Cache Management and Bypassing in GPUs

Many papers have discussed and proposed techniques for cache management in GPUs. Chen *et al.* [15] adopted cache bypassing and selective insertion of cache blocks in order to extend the lifetime of a cache line. Their insertion and bypassing decisions are based on block re-reference prediction for each block. Coordinated bypassing and warp throttling (CBWT) management scheme is proposed in [8]. In this scheme, cache bypassing is used to protect cache hot lines and

alleviate cache contention. Each line has a *protection distance (PD)* to indicate how many accesses this line is protected from and it is used to trigger bypassing. The bypass policy is coordinated with warp throttling to control parallelism on contention or congestion. MRPB [16] uses FIFO buffers to reorder memory requests before they are sent to L1 caches. It also employs cache bypassing, which is triggered when stalls are caused by lack of resources, to reduce intra-warp contention in L1 caches. Although CBWT combines bypassing and warp throttling to preserve locality, it can work together with MRPB to further improve performance and energy efficiency. Li *et al.* [17] proposed a cache bypassing scheme on top of CCWS called Priority-based Cache Allocation (PCAL), which activates more warps when the NoC is underutilized. Those extra warps are given lower priority in the cache to eliminate cache contention.

Keshtegar *et al.* [18] proposed a cache communication mechanism to reduce average memory access time. The main purpose of this mechanism, implemented in Logical Management Unit (LMU), is to forward the missed requests from L1 cache to neighboring L1 caches before sending that request to lower level cache. Although LMU reduces the traffic in the interconnection network, it increases number of accesses in the L1 caches since the source of L1 traffic now comes from both the SM itself and the LMU. Sharing tracker proposed by Tarjan *et al.* [19] offers an effective latency-tolerance mechanism to share cache blocks between multiple private caches. Sharing tracker uses a statistical method to track copies of cache blocks in private caches and tries to service memory requests from private caches before sending the request to next level in the memory hierarchy. The outcome of the tracker can lead to false negative or false positive results since it relies on partial information.

IX. Conclusion

We observe that neighboring CTAs within GPU applications share considerable amount of data. Unfortunately, the default GPU scheduling policy reduces the effective L1 cache size by unnecessary data duplication which in turn increases the data movement and power consumption. Based on this observation, we propose a sharing-aware CTA scheduler that attempts to assign CTAs with data sharing to the same SM to improve temporal and spatial locality. We then augment the scheduler with a sharing-aware cache management scheme. The scheme dynamically classifies private and shared data and proposes to prioritize storing private data in L1 cache and shared data in L2 cache. Our experimental results show that the proposed scheme reduces the off-chip traffic by 19% which translates to an average DRAM power reduction of 10% and performance improvement of 7%.

X. Acknowledgment

This work was supported by three grants: CCF-1657333, DARPAPERFECT-HR0011-12-2-0020 and NSF-CAREER-0954211

References

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

[2] M. Rhu, M. Sullivan, J. Leng, and M. Erez, "A locality-aware memory hierarchy for energy-efficient gpu architectures," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2013, pp. 86–98.

[3] F. NVIDIA, "Nvidia's next generation cuda compute architecture," *NVIDIA, Santa Clara, Calif, USA*, 2009.

[4] M. Lee, S. Song, J. Moon, J.-H. Kim, W. Seo, Y. Cho, and S. Ryu, "Improving gpgpu resource utilization through alternative thread block scheduling," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*. IEEE, 2014, pp. 260–271.

[5] K. Abdalla, L. Shah, J. Duluk, T. Purcell, T. Mandal, and G. Hirota, "Scheduling and execution of compute tasks," Jul. 18 2013, uS Patent App. 13/353,155. [Online]. Available: http://www.google.tl/patents/US20130185725

[6] A. Corporation, "Amd graphics cores next (gcn) architecture white paper," AMD Corporation, Tech. Rep., 2012.

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

[8] X. Chen, L.-W. Chang, C. I. Rodrigues, J. Lv, Z. Wang, and W.-M. Hwu, "Adaptive cache management for energy-efficient gpu computing," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2014, pp. 343–355.

[9] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "Gpuwattch: enabling energy optimizations in gpgpus," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, pp. 487–498, 2013.

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

[11] 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 *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. IEEE, 2009, pp. 44–54.

[12] 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," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1, pp. 395–406, 2013.

[13] 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*. ACM, 2013, pp. 99–110.

[14] O. Kayıran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither more nor less: optimizing thread-level parallelism for gpgpus," in *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*. IEEE Press, 2013, pp. 157–166.

[15] X. Chen, S. Wu, L.-W. Chang, W.-S. Huang, C. Pearson, Z. Wang, and W.-M. W. Hwu, "Adaptive cache bypass and insertion for many-core accelerators," in *Proceedings of International Workshop on Manycore Embedded Systems*. ACM, 2014, p. 1.

[16] W. Jia, K. Shaw, M. Martonosi *et al.*, "Mrpb: Memory request prioritization for massively parallel processors," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*. IEEE, 2014, pp. 272–283.

[17] D. Li *et al.*, "Orchestrating thread scheduling and cache management to improve memory system throughput in throughput processors," Ph.D. dissertation, University of Texas at Austin, 2014.

[18] M. M. Keshtegar, H. Falahati, and S. Hessabi, "Cluster-based approach for improving graphics processing unit performance by inter streaming multiprocessors locality," *IET Computers & Digital Techniques*, vol. 9, no. 5, pp. 275–282, 2015.

[19] D. Tarjan and K. Skadron, "The sharing tracker: Using ideas from cache coherence hardware to reduce off-chip memory traffic with non-coherent caches," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, 2010, pp. 1–10.