Tuning Applications for Efficient GPU Offloading to In-Memory Processing

Yudong Wu
University of California, San Diego
yuw466@eng.ucsd.edu

Mingyao Shen
University of California, San Diego
myshen@eng.ucsd.edu

Yi-Hui Chen
University of California, San Diego
yhc001@ucsd.edu

Yuanyuan Zhou
University of California, San Diego
yyzhou@eng.ucsd.edu

ABSTRACT

Data movement between processors and main memory is a critical bottleneck for data-intensive applications. This problem is more severe with Graphics Processing Units (GPUs) applications due to their massive parallel data processing characteristics. Recent research has shown that in-memory processing can greatly alleviate this data movement bottleneck by reducing traffic between GPUs and memory devices. It offloads execution to in-memory processors, and avoids transferring enormous data between memory devices and processors. However, while in-memory processing is promising, to fully take advantage of such architecture, we need to solve several issues. For example, the conventional GPU application code that is highly optimized for the locality to execute efficiently in GPU does not necessarily have good locality for in-memory processing. As such, the GPU may mistakenly offload application routines that cannot gain benefit from in-memory processing. Additionally, workload balancing cannot simply treat in-memory processors as GPU processors since its data transfer time can be significantly reduced. Finally, how to offload application routines that access the shared memory inside GPUs is still an unsolved issue.

In this paper, we explore four optimizations for GPU applications to take advantage of in-memory processors. Specifically, we propose four optimizations: application restructuring, run-time adaptation, aggressive loop offloading, and shared-memory transfer on-demand to mitigate the four unsolved issues in the GPU in-memory processing system. From our experimental evaluations with 13 applications, our approach can achieve 2.23x offloading performance improvement.

CCS CONCEPTS

• **Computer systems organization → Architectures**; • **Hardware → Emerging architectures**.

KEYWORDS

Emerging architectures, In-memory processing, GPU

ACM Reference Format:
Yudong Wu, Mingyao Shen, Yi-Hui Chen, and Yuanyuan Zhou. 2020. Tuning Applications for Efficient GPU Offloading to In-Memory Processing. In *2020 International Conference on Supercomputing (ICS '20), June 29-July 2, 2020, Barcelona, Spain.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3392717.3392760

1 INTRODUCTION

1.1 Motivation

For many GPU applications, the main memory bandwidth is a well-known critical bottleneck [14, 25, 31]. It is because most GPU applications perform massive data-parallel processing with thousands of concurrent threads [15] (e.g., up to 139264 threads on NVIDIA RTX 2080 Ti GPU [6]). When all threads on GPU load data from main memory, it introduces a large number of pipeline stalls due to memory bandwidth saturation [14, 25, 31]. A previous study [31] shows that memory bandwidth related pipeline stalls can contribute up to 61% execution time on GPU. Another study [14] shows that if we eliminate the overhead of waiting for data transfer from memory, the performance of memory-intensive GPU applications can improve by more than 460%.

To alleviate such memory bandwidth bottleneck, recent research [13, 16, 24, 32, 33] has proposed near/in-memory processing, to reduce the amount of memory-GPU transfers. The key idea of in-memory processing is to integrate processing capability into memory devices, e.g., integrating GPU Streaming Multiprocessors (SMs) in the logic layer of the 3D-stacked memory (memory stacks) (Figure 1). During application execution, GPU can offload memory-bandwidth intensive tasks to in-memory processors. The data transfer between in-memory processors and the memory can leverage data links inside a memory stack (dotted arrows in Figure 1), which provide larger bandwidth (e.g., in Figure 1, double the bandwidth) compared to the memory bus between GPU and memory.

While the above work has proposed a promising direction to address the GPU-memory bottleneck for memory-intensive applications, at its initial stage, several challenges remain open for GPU applications to fully take advantage of in-memory processing. Therefore, quite some memory-intensive applications cannot benefit from such advanced technology, i.e., in-memory processing. Figure 2 shows the performance of seven applications running on GPU with and without in-memory processing (experimental setup is similar

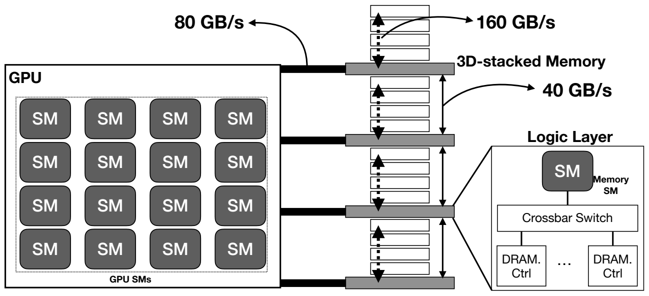

Figure 1: A typical architecture for GPU with in-memory processing support. Multiple 3D stacked memory devices connect to GPU via the memory bus, and the logic layer in each memory stack can integrate one Memory-SM (Streaming Multiprocessor in the memory). These Memory-SMs in the system are similar to GPU SMs (Streaming Multiprocessors on the GPU). All memory stacks are connected via inter-stack data links. Due to physical limitations like available data pins, the bandwidth of each data links usually satisfies: $BW_{inter-stack} < BW_{memory\ bus} < BW_{intra-stack}$. Specific bandwidth stats are referenced from [13, 16].

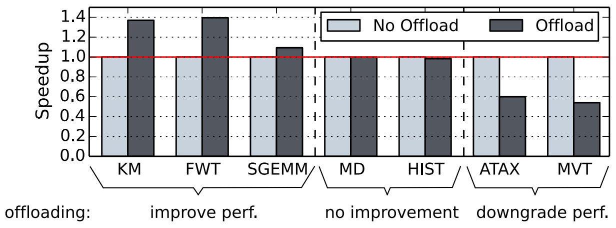

Figure 2: Application performance on GPU w/o and w/ in-memory processing. All results normalized to application execution time on GPU without offloading. We can observe that, though some applications can benefit from offloading, there are still examples that applications cannot improve via offloading. Some applications even experience performance downgrade after offloading.

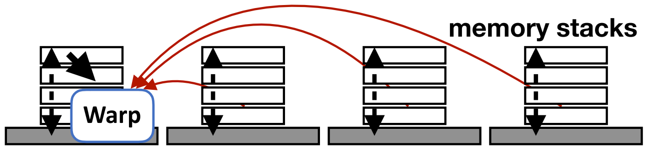

Figure 3: Memory access pattern of KMEANS when the GPU offloads a warp to in-memory processing. When GPU offloads one warp to a memory stack, the offloaded warp will obtain data from all memory stacks, lead to much inter-stack communication traffic.

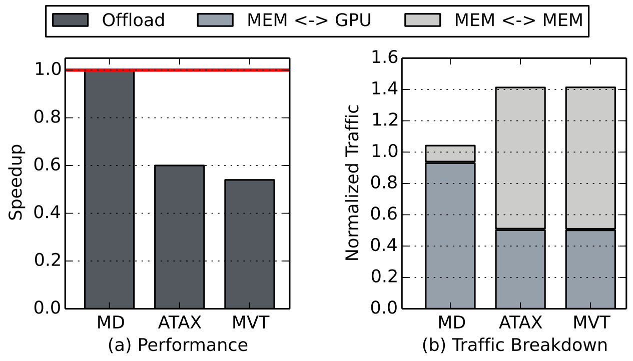

Figure 4: Performance and traffic breakdown for MD, ATAX & MVT. All results normalized to application execution time or traffic breakdown on GPU without offloading to in-memory processing. When we offload these applications, they suffer from too-much inter-stack traffic (as shown in (b)). As a result, we cannot observe any performance improvement, and even observe performance downgrade, as shown in (a).

to previous work [13, 16, 24, 32]). Although three (KM, FWT & SGEMM) of them get performance improvement with in-memory processing technology, the other four applications have no performance gain. Even worse, two of them (ATAX & MVT) experience more than 40% performance downgrade.

A natural question is why these applications cannot benefit from in-memory processing. Further investigation reveals the following four issues:

Issue 1. Most current GPU applications are highly optimized to maximize data locality on GPU and thereby do not naturally match the data locality requirement of in-memory processing. As shown in Figure 3, in current applications implementation, each warp [1] on GPU accesses data from all memory stacks in order to leverage all available memory bandwidth between GPU and memory. However, offloading such warps to in-memory processors can introduce large data traffic on links connecting different memory stacks (ie., inter-stack data links). Due to the physical limitation on memory stacks, e.g., available data pins on each stack, the bandwidth of inter-stack data links is typically as small as 50% of the main memory bus bandwidth [13, 16]. Consequently, if the offloaded execution involves too much inter-stack communication, the offloaded execution performs even worse despite the fact that the computation is inside the memory. As shown in Figure 4, three applications (MD [8], ATAX [10], and MVT [10]) are such examples. The heavy inter-stack traffic with offloading execution nullified the performance benefit of offloading. Even worse, in ATAX & MVT, offloading causes 66% performance downgrade. To address this problem would require restructuring the application to improve data locality to minimize the amount of inter-stack data transfers.

Issue 2. Besides restructuring applications to improve data locality, another challenge is how to dynamically determine which computation can truly benefit from offloading to in-memory processors. After all, not all computation can be optimized to avoid inter-stack memory traffic. Ideally, GPU should not offload computation that cannot benefit from in-memory processing. Previous work [13] has used static methods to estimate whether it is worthwhile to offload certain computation from GPUs to in-memory processors, but as shown in Figure 4 (particularly, in ATAX and MVT), static methods can often make wrong estimations, incorrectly deciding offload computation to in-memory processors. That is why these two applications perform 66% worse than the original (GPU

[1] A warp is "a vector of threads" executing the same instruction, sharing same SIMD logic in GPU SMs. It is also the hardware scheduling execution unit on GPU.

without offloading). This indicates static methods may need to combine with dynamic adaptation to achieve good offload decisions.

Issue 3. To maximize the performance, it is also important to determine how much computation to offload to in-memory processor. If we offload too little, we do not fully take advantage of in-memory processors. On the other hand, if we offload too much, in-memory processors become the bottleneck and GPUs have to wait. The estimation is not straightforward because, due to significantly shortened data access time, the execution time of the same computation on in-memory processors can be much faster than that on GPUs. In other words, our load balancing mechanisms cannot simply consider an SM (stream multiprocessor) inside the memory (Memory SMs) deliver the same performance as an SM on GPU (GPU SMs).

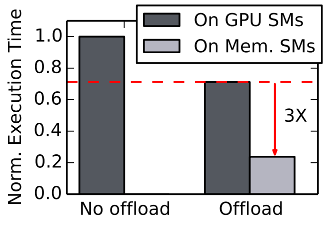

Figure 5: An example of warps' execution time difference between GPU SMs and memory SMs. The results show the execution time of *modulateKenrel()* on GPU w/o and w/ in-memory processing. All results normalized to average warp execution time on GPU SMs w/o offloading to in-memory processing. When offloading, on average: (1) warps run faster on both GPU and memory SMs compared to warps running on GPU SMs without offloading; (2) warps run 3x faster on memory SMs compared to warps remain on GPU.

As shown in Figure 5, some warps on in-memory processors can run 3x faster than the warps remain on GPU. In such a case, when warps offloaded to in-memory processors have finished execution, the warps on GPU are still busy running. At this point, the in-memory processors become idle. Unfortunately, previous work has ignored this issue, and simply offloads the same number of warps to in-memory processors based on the number of stream-multiprocessors. It ignores the fact that with good data locality, an in-memory SM processor has much faster data access and thereby can handle more computation than an SM inside the GPU.

Issue 4. In addition to main memory, computation in GPU applications often need to access data stored in shared memory, which is an on-chip cache on GPU where a GPU program can explicitly allocate and access. Naively offloading execution with shared memory access can introduce data transfer traffic between GPU and in-memory processors via bandwidth-limited memory buses. As such, previous work decides not to offload such computation. However, in some application routines, the amount of shared memory accessed is much smaller than the main memory accessed. For example, in our evaluation, for a routine in Histogram [21], each warp accesses 140×32 integers (4480 integers, or 17920 bytes), while each warp only accesses only 64 bytes shared memory space. For such a routine, it is still beneficial to leverage in-memory processing offload by sending the relevant shared memory data to the in-memory processors ahead of time. The amount of shared memory data that needs to be transferred with offloading is smaller than the amount of memory data needed to transfer from main memory to GPU without offloading, and thus can still save memory traffic.

1.2 Our Contribution

To address the above issues, we investigate four optimizations that can achieve significantly improved performance in in-memory processor offloading, especially for applications that have no performance gain or even performance degradation with offloading. The four optimizations include: (1) *Application restructuring.* To address the issue with the current GPU application implementation, we propose some simple but effective restructuring techniques to improve the data locality for in-memory processors and minimize inter-stack memory traffic. The main principle is to change the way how loops iterate over data structures in GPU applications. However, straightforwardly changing the access pattern can cause performance penalty due to increased L2 cache conflict misses (more details in § 3.2.2). We further optimize loops implementation to address this L2 cache miss issue. (2)*Run-time adaptation.* To address the issues with static offload planning, our system dynamically monitors the amount of memory traffic during offloading and performs run-time adaptation by moving non-beneficial offload with high inter-stack traffic back to the GPU. (3) *Aggressive loop offloading.* To reduce the idle cycles of in-memory processors (benefiting from its fast data access), this optimization aggressively offloads more warps to the memory side SMs once they finish the current warps, (4) *Shared-memory transfer on-demand.* For warps that access shared memory and main memory at the same time, if the accessed shared memory data is much smaller than the amount of data accessed in main memory, this optimization sends the needed shared memory data from the GPU to in-memory processors and offloads the corresponding warps to in-memory processors.

We evaluate our system with 13 GPU applications on the GPGPU-SIM [3] simulator with in-memory SM extension. The four optimizations together have achieved up to 2.23x performance improvement compared to the state-of-the-art offloading mechanisms. In particular, for the 5 of the 13 applications, previous offloading had no performance improvement, or even introduced up to 46.02% performance degradation; in comparison, our four optimizations together can achieve 1.10x-1.90x application performance improvement over the original GPU execution (without offloading). Separating each optimization on its own, application restructuring can improve offloading performance by 1.17-1.74x for eight applications; run-time adaptation can achieve offloading performance improvement by 2.22x and 2.23x for two applications; aggressive loop offload can improve offloading performance by 1.44x and 1.53x for two applications; shared-memory transfer on-demand can further improve offloading performance by 1.03x-1.23x for three applications.

2 BACKGROUND

Our work is built upon previous GPU in-memory processing systems mentioned in § 1.1 ([13, 16]). Here we first briefly describe the high level points of these systems to provide a context.

Static offload-benefit analysis. The first step of offloading is to identify the code blocks that can benefit from offloading by static analysis. For each code block, the static analyzer compares the overhead and the benefit of each code blocks, and instruments a `PIMBEGIN` instruction at the beginning of the code block (referred as an offload candidate), if the overhead is smaller than the benefit. The overhead consists of the total size of live-in and live-out registers for the code block. The benefit consists of the total size of data transferred via the memory bus for all memory load/store instructions in the code block.

The routine of offloading on GPU. During execution, when a GPU SM encounters a `PIMBEGIN` instruction, the GPU SM marks the warp as waiting to offload. The GPU will not offload the offload candidate until the GPU encounters the first memory access instruction in the offloading candidate block. At this point, the GPU tries to offload the candidate to the memory-SM on one memory stack (destination memory stack). The destination memory stack is the same memory stack as the destination of the first memory access instruction in this offloading candidate block. If the destination memory stack cannot accept any new warps due to full occupation, the GPU SMs will continue to execute this warp, without any further offloading trial on this offloading candidate block in this warp. If the destination memory stack can accept a new warp, the GPU stops the warp's execution on the GPU SM, and sends all live-in registers and the start PC to the destination memory stack. The memory SM on the destination memory stack creates a new warp, sets relevant registers' value, Finally, the offloading candidate starts executing on the memory SMs.

Once the offloaded warp finishes its (offloading) execution on the memory SMs, the memory SMs interrupts the GPU, and sends all live-out registers back to the GPU. When GPU receives interrupt from the memory stack, it sets all the live-out register values on the GPU SM where this warp resides before offloading, and resumes the execution on this GPU SM.

Note that, for the convenience of the discussion, in the following sections, when we mention offloading a warp, we actually refer to offload a warp that executes an offload candidate block.

3 DESIGN AND IMPLEMENTATION

3.1 Target applications

In this paper, we mainly focus on optimizing loops. We do *NOT* optimize non-loop routines because of three reasons:

In-memory processing prefers loops. As mentioned in § 1.1 and § 2, by offloading to in-memory processors, the GPU trades the overhead of sending execution context (live-in/out registers), for the benefit of saving data traffics for load/store on the memory bus. Intuitively, in-memory processors prefer routines that have massive amounts of memory access and only a limited amount of live-in/out context. Loops naturally fit this intuition. The loop with load/store can provide large amounts of global memory access. Compared to the sequential instruction sequence providing the same amount of global memory access, the loop's live-in/live-out registers are usually fewer. This intuition is also backed by our evaluation. During our evaluation, only 8 of 68 (11.76%) total offload candidates identified are non-loop candidates.

Loops are commonly seen in GPU applications. GPU applications usually aim to parallelly process a huge amount of data. It is not surprising that GPU applications use loops to process multiple data elements in the same thread.

Loop Counter Value	0	1	2	3	4	...
Memory Stack ID	0	3	2	1	0	...

Figure 6: Memory stacks accessed by a warp with different loop counter in KMEANS. For different loop counter value (different loop iterations), the warp accesses different memory stacks. If we directly offload the warp to any specific memory stack, 3/4 of the memory access is inter-stack communication.

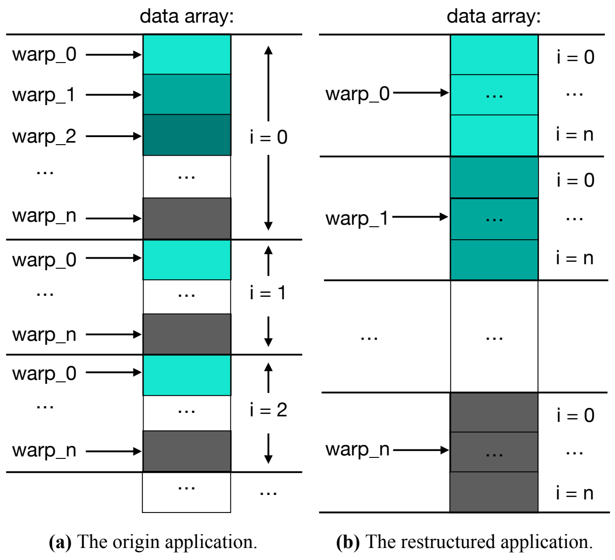

Figure 7: Memory access pattern before & after application restructure. (a) current GPU applications' access pattern; (b)access pattern after restructure. *i* stands for loop counters in a loop, and *warp_x* stands for the warp id of different warps.

Loops in original GPU applications easily lead to inter-stack communication. In the current GPU application, the accessed data locations in the loop spray in the data structure sparsely. These sparsed locations can easily fall into multiple memory stacks (Figure 6 and § 3.2).

3.2 Optimization 1: application restructuring

In current GPU applications, warps access data from a data array under the thread-interleaving pattern. In the thread-interleaving pattern, all running warps access consecutive locations in the data structures when they have the same loop counter value (Figure 7(a)). It has two benefits: First, the thread-interleaving pattern can guarantee good memory access coalescing. and thus reduce the necessary access request traffic sent to DRAM[19]. Second, for the same loop iteration, all warps' access evenly distributes to all memory stacks. This access happens around the same time, thus they can leverage all available memory-bus data links.

However, the thread-interleaving pattern becomes problematic in the in-memory processing scenario. In the thread-interleaving

```
…
for (j=0; j < nfeatures; j++)
{
    int addr = point_id + j * npoints;
    float diff = features[addr] -
                 clusters[cluster_base_index + j];
    ans += diff*diff;
}
…
```

Listing 1: Example from KMEANS in Rodinia[4]. We can change the indexes each warp will access in `features` array by changing how `addr` is computed in the source code. If a warp accesses *different indexes* in the array `features`, the value of `ans` will change. The value of `ans` impacts the following clustering process in KMEANS, and generates incorrect clustering output.

pattern, each warp accesses discrete addresses, because there is always a fixed gap between two consecutive access issued by the same warp. For example, in Listing 1, there is always a gap of *npoints* features between two features accessed by the same warp consecutively. This discreteness on accesses makes the warp more easily to access different memory stacks.

Figure 6 shows an example of this issue. We traced the memory access from one warp during its execution, and we observed that this warp load data from all memory stacks during the loop. If this warp is running on GPU, loading data from different memory stacks will not cause any problem, since memory bus between GPU and each memory stack has the same bandwidth. When we offload this warp, the inter-stack data load produces high performance penalty, since there is only limited bandwidth on data-links between memory stacks ([13, 16]). The inter-stack access penalty could neutralize, if not overdraw, the benefit from in-memory processing.

As aforementioned, this issue is caused by the discreteness of accessed memory address from the same warp. If each warp access continuous region in the physical memory space, it has a higher chance to target the same memory stack. As a result, it reduces inter-stack memory access. To approach the access pattern change, we restructured the implementation of the existing GPU applications.

Application restructuring. Figure 7(b) shows the changes to the data array access with application restructure. We observe that the application handles array data structures in most cases. Thus, we restructure the application by changing the indexes in the array each warp accesses. First, we insert statements that calculate an index range that each warp should access based on warp_id. Then, we change how the loop counter changes during the loop in applications source code. Before the loop, we set the loop counter to be the start of the index range. After each loop iteration, we plus the loop counter by 1. After the restructure, we make each warp processing consecutive elements in the array. For each thread in the warp, we still adopt a RowMajor [19] pattern, to ensure good memory coalescing.

However, for application restructuring, there are two challenges to resolve: 1) **Ensure semantic correctness** (§ 3.2.1) and 2) **Avoid GPU performance downgrade.** (§ 3.2.2).

3.2.1 *Data re-arrangement.*

When restructuring the application, we change the indexes in the data array that each warp accesses. However, in some applications, changing these locations changes the application results. For example, in Listing 1, the value of `ans` will change, if we change indexes accessed by each warp. The `ans` value finally decides how clustering is formed. As a result, it will impact the final results and generates incorrect output if we only naively change the indexes without changing the data layout in the arrays in Listing 1.

Thus, besides changing the indexes, we also need to change the data layout in the array before data processing. We change the kernel that pre-processes data from the input file in the GPU applications to change the data layout. If there is no such a kernel, we will implement a new kernel to change the data layout. In our evaluation, 10 of the 13 evaluated applications require data layout changes, and 4 of them require implementing new kernels. We faithfully include the execution overhead to re-layout the data as part of the offloading execution time in our evaluation.

3.2.2 *Index randomization.*

Straightforward restructure increases L2 conflict misses. The application restructuring should not severely downgrade the application performance on GPU. The in-memory processing power is much smaller compared to that on GPU (e.g, one SM in each memory stack, in total 4 SMs on the memory side, compared to 64 SMs on GPU). When offloading, the GPU cannot offload all warps to in-memory processors. If the application performance on GPU downgrade heavily warps on GPU can become the performance bottleneck.

For all warps, the memory access in the same loop step happens roughly at the same time [20]. When the total data size that a warp accesses is $N_{L2\_sets} \times Size_{cache\_line}$ aligned [2], different warps conflict on the same L2 cache set when the loop counter is the same, and thus increasing the L2 cache conflict misses. In our evaluation on FastWalshTransformation, after restructuring the L2 cache reservation fails are increased around 4X due to L2 conflict misses, and the performance on GPU can downgrade by 40%.

Index randomization. To mitigate L2 conflict miss in restructured applications, we further optimize loops with index randomization optimization. In most common cases, when current GPU applications perform loops, their loop counters are usually initialized with the same value. With index randomization, we randomize the initialize value when we restructure the application source code. For any *warp_x*, we make sure that different warps start with a different random loop counter that is calculated based on the *warp_id*. Thus, memory accesses issued by different warps around the same time are less likely to conflict on the same L2 cache set.

3.3 Towards automatically restructure

In the current stage, we only manually restructure the applications. From our experience, manually restructuring the applications does not require too heavy effort. It does not require too much experience in CUDA or too much time to restructure an application. In our evaluations, the restructure only takes *at most 2.5 hours* for a

[2]This requirement can easily meet if the data size and the total threads number are both powers of 2.

first-year graduate student, for the 13 applications we evaluated. However, the restructuring effort and overhead could raise problems for more complicated applications used in industries' production environments.

With the help of software developer's annotation, potentially the restructure routine can be automated. The patterns we restructure are 1) all threads in the application interleaving access an array; 2) all threads in a CUDA CTA (**C**ooperative **T**hread **A**rray) interleaving access an array. With application developer annotates 1) The start address of the array; 2) Whether the array is shared by all threads or only by a CUDA CTA. During compilation, the static analyzer restructures the access to the annotated array from the thread-interleaving pattern to the restructured pattern (§ 3.2).

However, if the application restructuring relies on the application semantics, then automating restructuring can be challenging. We left the solution of application-semantic-related restructuring for future investigation.

3.4 Optimization 2: run-time adaptation

Static offload benefit analysis can go wrong. As mentioned in § 2, when analyzing the benefit obtaining from offloading a code block, the compiler performs a static analysis on the bandwidth consumption assuming that **all** memory access during offloading is intra-stack traffic. However, this assumption can be wrong during the runtime. Particularly, as mentioned in § 1.1, though two applications (ATAX & MVT) suffer from insane performance penalty (> 66%) from inter-stack communication traffic, the static analyzer still judges them worthy to offload, since the traffic characteristic is not available at the time of static analysis.

To address this challenge, the GPU system should combine dynamic traffic information to decide whether the offloading is beneficial or not. The Memory SM is better to monitor the traffic status of the offloading warps, and stops the offloading execution for the warps introducing too much inter-stack communication traffic.

Offloading runtime traffic monitoring and adaptation. We add two new components in the Memory SMs to support our optimization mechanism (Figure 8): 1) A Traffic History Buffer to record the traffic history of each warp on the Memory SM. 2) A Memory Traffic Monitor to track the intra-stack and inter-stack traffic for each warp on the Memory SM.

During the execution on Memory SMs, the Memory Traffic Monitor tracks the destination of each memory access for each warp. The Memory Traffic Monitor keeps the amount of inter-stack and intra-stack memory access into an internal temporary buffer. Once the execution reaches the beginning of a new loop round, the Memory Traffic Monitor first pushes the traffic record of last loop round into Traffic History Buffer, and then checks in the last three loop rounds, whether the ratio between total inter-stack traffic and total intra-stack traffic is greater than a predefined threshold. If it is, the Memory Traffic Monitor stops the execution for the current warp, and send this warp back to GPU.

To support our mechanism the Memory SM needs to identify the beginning of the loop round. We simply insert a special instruction as the first instruction of a loop round. As shown in Figure 8, once the issue logic in the pipeline encounters the special instruction, the issue logic sends the adaptation request to the Memory Traffic

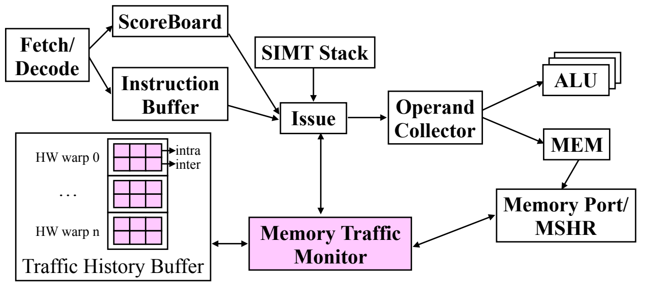

Figure 8: Block diagram of the run-time adaptation hardware and its interactions with the typical GPU pipeline [3, 20] in the Memory SMs. In our design, we add two new components: Memory Traffic Monitor and Traffic History Buffer, to the Memory SMs. During offloading, Memory Traffic Monitor records a short period of traffic status (#inter-stack traffic requests and #inner-stack traffic requests) of previous execution, and stores them into Traffic History Buffer. Later, Memory Traffic Monitor will check the ratio between recorded # of inter-stack and # of inner-stack traffic requests. Memory Traffic Monitor will stop the offloading execution and return the offloaded warp to GPU, if this ratio is higher than a pre-defined threshold.

Monitor, and let the Memory Traffic Monitor performs all aforementioned checks. In our implementation, we just let `PIMBEGIN` be the special instruction for run-time adaptation since, in the aggressive loop offload optimization, we also move the `PIMBEGIN` instruction to the location of first loop instruction (discussed in § 3.5).

3.5 Optimization 3: aggressive loop offloading

Unbalanced execution time between GPU and in-memory processors. When offloading a memory-intensive warp to in-memory processing, the warps on Memory SMs execute much faster than warps on the GPU side. For example, in FWT, the loop in *modulateKenrel()* executes 3x more cycles to finish compared to the warp execution on the memory side. This is not a surprising result. The loop to be offloaded is memory-intensive, while the in-memory processors can provide much larger memory bandwidth, as well as smaller memory load latency due to shorter data movement distance.

For each warp, if the GPU only tries to offload the loop once, it can waste idle time on in-memory processors. For example, in FWT, when warps on in-memory processors finished their execution, warps remain on GPU are still far from finishing. In-memory processors become idle, while GPU has computation tasks that can offload. Unfortunately, all warps on GPU have already checked whether to offload before entering the loop, and will not check again. The unbalance execution time between Memory SMs and GPU processors expose potential offloading opportunities, but the current offloading design cannot utilize it.

Aggressive loop offloading. To utilize the aforementioned offloading opportunities, the GPU should aggressively offload more warps once any in-memory processors become idle. With aggressive loop offloading, the offloading benefit analyzer inserts the `PIMBEGIN` instruction as the first instruction in the offloading candidate loop, instead of right before the offloading loop starting point. After the change, for an offloading candidate remains on GPU, at the beginning of each loop round, the GPU executes `PIMBEGIN` instruction, which starts an offloading trial. If the target memory stack is available to accept a new warp, the GPU offloads the current warp to in-memory processors. The only exception is when the warp on GPU was sent back by Memory SMs due to run-time adaptation. In such a case, the GPU will not try to offload this warp again, since runtime stats already show that this warp cannot benefit from offloading (§ 3.4).

Note that, with run-time adaptation and aggressive loop offloading Optimization, we move the `PIMBEGIN` instruction as the first instruction of the loop, and assign it with new semantics. Now the `PIMBEGIN` instruction executes on both the GPU side and the memory side. On the GPU side, `PIMBEGIN` starts an offloading trial. On the memory side, `PIMBEGIN` performs run-time adaptation.

3.6 Optimization 4: shared-memory transfer on-demand

In previous proposals, when a basic block performs shared memory access, the GPU cannot offload this block to in-memory processing. Shared memory is a special on-chip cache that can be explicitly allocated by application code, and shared memory can be accessed by multiple warps in the same CUDA CTA. Since shared memory is an on-chip cache, offloading basic blocks accessing shared memory, without making a copy of shared memory to the Memory SMs, can lead to massive communication traffic between Memory SMs and host GPU. Thus, previous proposals do not allow offload basic blocks with shared memory access.

However, from our observation, in GPU applications, shared memory access has the following characteristics:

- **Shared memory is used for accumulation**. Shared memory can be used as a communication channel among all threads in the same CTA. Usually, it is used as a counter, or accumulator, to accumulate a series of numbers in a CUDA CTA.
- For basic blocks accessed both the shared memory and the global memory, the size of shared memory used by each warp is **smaller** than the global memory accessed. For instance, in Histogram [21], in a loop, each warp accesses 140×32 integers (17920 bytes), while each warp only accesses 64 bytes shared memory.

For basic blocks satisfy the aforementioned two charasteristics, and only use one piece of the shared memory region, we can offload them to in-memory processing by sending data in the shared memory back and forth before and after offloading to exploit the offload benefit. We adopt the following way to offload basic blocks with shared memory access.

(1) Application developers annotate loops with both the global memory and the shared memory access. The annotation includes the size of shared memory, and the size of the global memory accessed by each warp.
(2) When offloading, the Memory SMs allocates the same size of shared memory for the warp. If it is used as a counter, initialize all data in the allocated shared memory as zero; otherwise, the GPU transfers data in the shared memory to memory SMs at the time sending the offloading request.
(3) After offloading ends, the Memory SMs send the live-out registers, and the memory side SMs back to host GPU. Since the operation on shared memory is accumulation, the host SMs just add the content in the memory side shared memory to the relevant location.

Main GPU	
Core Number	68 SMs for baseline GPU 64 SMs for GPU w/ PIM
Core Configuration	1200 MHz, 64 warps/SM 32 threads/warp, 65536 registers/SM 32 CTAs/SM
Shared Memory	96 KB/SM can be reconfigured as L1
Private L1 Cache	32KB, 4-way, write through
Shared L2 Cache	1MB, 16-way, write through
Clock Frequency	Interconnect 2000 MHz, L2 1200 MHz
Off-chip Links	
GPU to Memory	80 GB/s per link, 320 GB/s total
Memory to Memory	40 GB/s per link, fully connected
Memory Stack	
SM in Memory Stack	1 SM per memory stack, 64 warps/SM
Memory Stack Configuration	4 memory stacks, 16 vaults/stack 16 banks/vault, 64 TSVs/vault 1.25 Gb/s TSV signaling rate
Internal Memory Bandwidth	160 GB/s per memory stack 640 GB/s total
DRAM Sched. Policy	FR-FCFS [26, 34]
DRAM Timing	DDR3-1600 [29]

Table 1: Major simulation parameters.

Name	Source	Name	Source
SP	[21]	CFD	[4]
CORR	[10]	HW	[4]
HS	[4]	RD	[21]
HIST	[21]	KM	[4, 27]
FWT	[21]	MD	[8]
SGEMM	[11]	MVT	[10]
ATAX	[10]		

Table 2: Summary of 13 evaluated applications.

4 EVALUATION SETUP

We simulate the in-memory processing GPU system by modifying GPGPU-SIM 4.0.0 [3] to include in-memory processing extension. The inter-stack data links are modeled using BookSim2 [7], a cycle-accurate interconnection network simulator that is also used in the GPGPU-Sim for on-chip interconnect simulation. In terms of bandwidth of data links, similar to previous works [13, 16], we assume the inter-stack communication bandwidth is 0.5x of the GPU memory bus bandwidth. We assume the internal data bandwidth between logic layer SMs and the memory layers in the same memory stack is 2x of the GPU memory bus bandwidth.

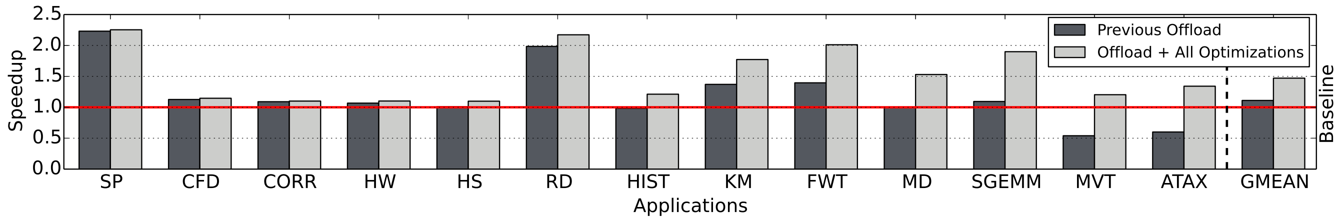

Figure 9: Performance comparison between previous offload and offload with all our four optimizations. The baseline is the execution time of the non-restructured applications on GPU. Previous Offload stands for all previous offloading proposal in [13]. Offload + All Optimizations stands for offloading applying all optimization discussed in § 3.

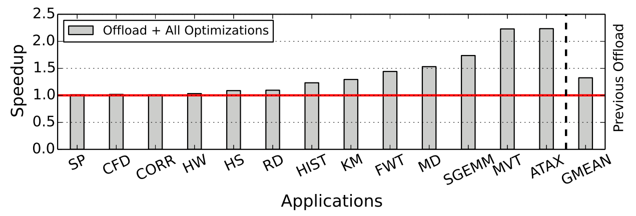

Figure 10: Performance speedup compared to previous offloading proposals. The baseline in this graph is the performance of previous offload proposals.

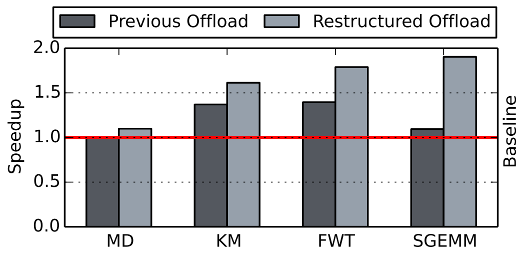

Figure 11: Performance of previous offloading and restructured offloading. (No obvious changes in other applications).

4.1 Evaluated Applications and Metrics

Table 1 summarizes our simulation parameters. To make a fair comparison between the baseline GPU system and the in-memory processing GPU system, we use the same number of SMs in two different systems, to make sure these two systems have the same computation power.

Table 2 summarizes our evaluated applications. We evaluated 13 applications from Rodinia [4], scientific workloads used by GPGPU-Sim [3], CUDA samples [21], Parboil [11], shoc [8] and Poly [10].

In our evaluation, we measure the execution time of the applications on GPU (in the simulator). When we compared the performance of our optimized offloading to the previous offloading, we also consider the execution overhead of data re-arrangement (§ 3.2.1) in our optimized proposals. All the data re-arrangement process is implemented as a CUDA kernel. Its execution time is also included in the total application execution time in our optimized offloading.

5 EVALUATION RESULTS

We evaluated the effect of four optimizations on the GPU offloading performance. Unless specified, all application execution time results are normalized to the execution time of non-restructured applications running on GPU without offloading.

5.1 Effect on performance

Figure 9 shows the application performance changes when applying four optimizations on GPU application offloading. With all four optimizations, we achieve a speedup on performance by 1.47x on average (up to 2.25x) over the baseline.

Compared to the previous offloading proposal (Figure 10), we achieve up to 2.23x performance improvement. Especially, we improve the offloading performance in ATAX & MVT from 0.60x & 0.54x (more than 40% slowdown) to 1.34x & 1.20x (more than 2.22x speedup over the previous offloading proposals), respectively. The detailed benefit analysis is listed below:

With application restructure for better stack data locality, we can achieve substantial performance improvement (> 10% performance improvement) in 4 applications (KM, FWT, SGEMM, MD, shown in Figure 11). Most of the benefits we gained come from the reduction of the inter-stack data traffic. For example, in KM, after applying application restructure optimization, inter-stack data traffic during offload execution is reduced by 59.20%. The inter-stack traffic reduction in KM leads to 1.18x performance improvement. Overall, compared to the previous offloading proposal, with application restructure, we on average reach 1.07x (up to 1.74x) performance improvement for 13 applications.

With aggressive loop offloading (ALO), we can further improve the offloading performance by 4.15%. This is achieved by offloading more execution tasks to fill the available idle cycles on logic layer SMs. For example, in MD (shown in Figure 13), with ALO, we can offload more than 4x more memory requests (9.49% to 40.37% of total memory requests) and instructions (9.13% to 38.55% of total instructions during runtime) to the in-memory processing, and reach 0.44x performance improvement after applying ALO.

The previous offloading proposal severely hurts performance in 2 applications (ATAX & MVT). Application restructuring cannot help in these two applications. As mentioned in § 3.4, the application performance downgrade happens because of incorrect static

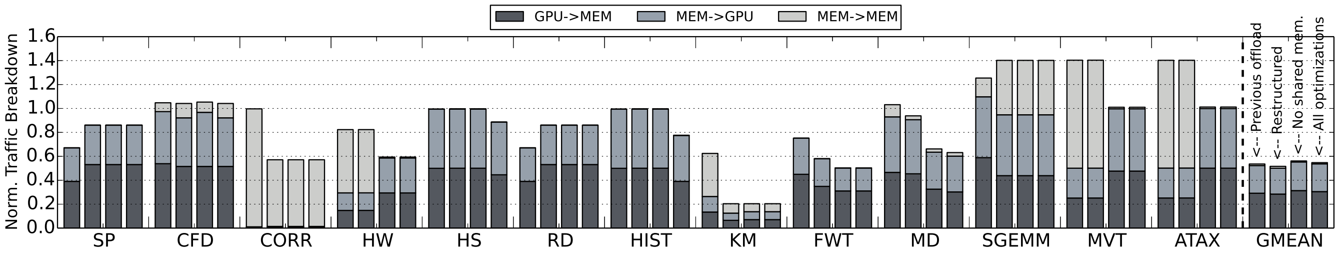

Figure 12: Data traffic comparison with different optimizations for all 13 applications. Each application has four data bar. From left to right, they represent traffic stats for Previous Offload, Restructured Offload, No Shared Memory Offload and All optimizations. *No Shared Memory Offload* stands for offloading with all optimizations in § 3 except shared-memory transfer on-demand. *All optimizations* stands for offloading with all optimizations mentioned in § 3.

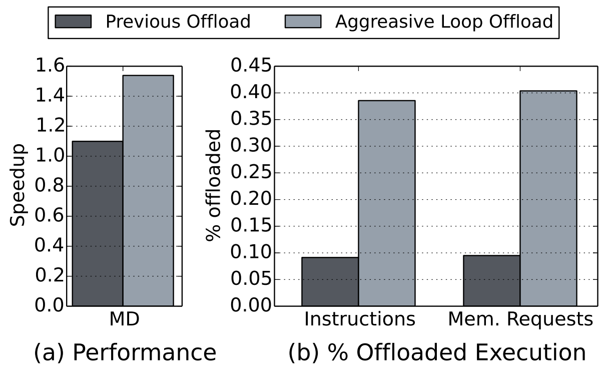

Figure 13: The performance and the % of offloaded execution, w/ and w/o Aggressive Loop Offload in MD. (No obvious changes in other applications).

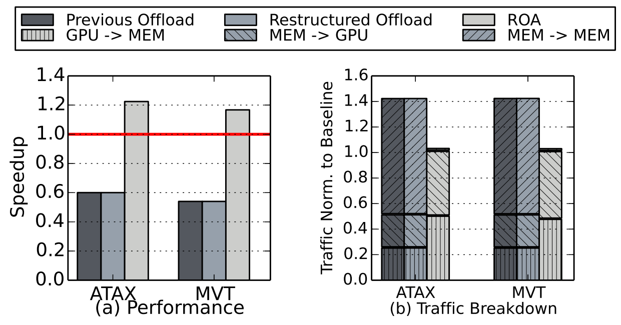

Figure 14: The effect of run-time adaptation optimization on performance and data traffic. (No obvious changes in other applications).

benefit estimation. During offloading, dominated inter-stack memory access leads to application performance downgrade.

We address incorrect estimation issues with run-time adaptation (RA) optimization. As shown in Figure 14(a), in these two applications, with RA optimization, we can improve the offloading speedup from 0.60x (ATAX) and 0.53x (MVT), to 1.22x (ATAX) and

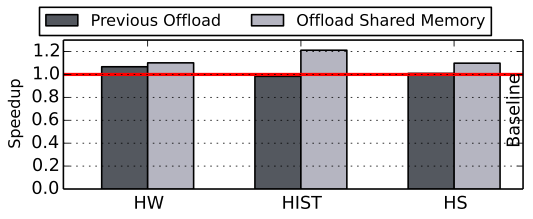

Figure 15: The effect on the performance of shared-memory transfer on-demand optimization. (No obvious changes in other applications).

1.17x (MVT) improvement over the baseline, respectively. The performance improvement comes from the elimination of inter-stack communication after applying RA (from $> 80\%$ of the baseline total memory traffic to $< 1.3\%$ after applying RA, as shown in Figure 14(b)).

For HIST and HS, all previous optimization does not improve offloading performance. This is because, in these two applications, the major memory bandwidth overhead came with shared memory operations, which all of the previous work cannot deal with. Figure 15 shows the performance improvement of shared-memory transfer on-demand. With shared-memory transfer on-demand, the offloading performance of these two applications is improved from almost no improvement (less than 1%) to 1.21x (HIST) and 1.10x (HS), respectively. The improvement primarily comes from offloading basic blocks with shared memory access. For example, after applying shared-memory transfer on-demand, HIST offloads 8x more instructions and memory access requests to the in-memory processing, leads to 1.21x performance improvement regards to the baseline. Without shared-memory transfer on-demand, HIST cannot gain performance improvement from offloading regards to the baseline. Besides, the shared-memory transfer on-demand also improves the HW's performance by 3.90%.

5.2 Effect on data traffic

Figure 12 shows the effect of offloading optimizations on data traffic. All results are normalized to the data traffic when running non-restructured applications on GPU without offloading. We make two observations:

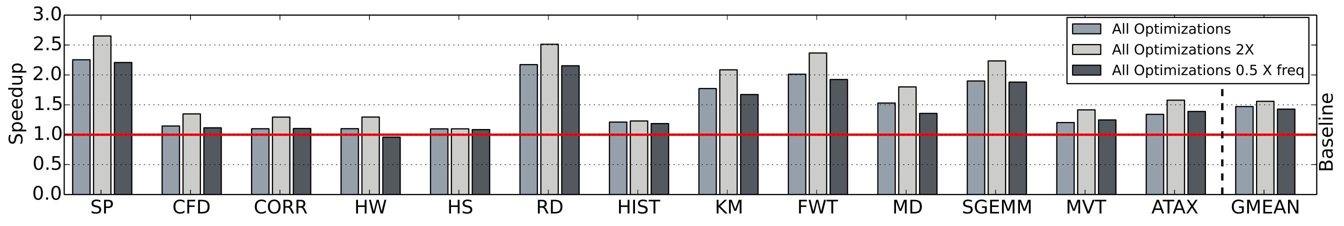

Figure 16: Execution time with different in-memory SMs processing power configuration. All Optimizations stands for offloading applying all optimizations mentioned in § 3. *2X* stands for the case that in-memory SMs can handle twice as many warps as that in § 5.1, and *0.5 X frequency* stands for in-memory SMs' clock frequency is only 50% of that on the GPU.

First, as expected, application restructure and run-time adaptation can effectively reduce inter-stack traffic. For application restructure, this is because application restructure is aim to make the application more optimized for in-memory processing: each warp only accesses data in a single memory stack in the offload candidate code blocks. In KM and CORR, inter-stack data traffic is reduced by 28.13% and 42.89% respectively. For run-time adaptation, this is because the in-memory processing will stop offloading execution for warps with huge inter-stack data traffic. In ATAX and MVT, inter-stack data traffic is reduced from more than 90.02% to less than 1.30%, after applying run-time adaptation.

Second, other optimizations effectively reduce GPU – Memory traffic. This is achieved by offloading more execution tasks to the in-memory processing part. More specifically: 1) Aggressive loop offload reduces GPU – Memory traffic by 3.91% averagely (up to 39.13% in MD); 2) Applying shared-memory transfer on-demand optimization can reduce the GPU – Memory traffic in HIST by 22.07%, and in HS by 9.28%.

5.3 Sensitive to in-memory processing capability

When offloading to in-memory processing, how many warps each memory stack can handle impacts the performance benefit of these optimizations. The more warps that the in-memory processors can handle, intuitively the application should perform better when offloading.

Figure 16 shows the performance result if in-memory SMs' warp capacity is doubled, and if the in-memory SMs' clock frequency is only 50% of the GPU's clock frequency. If the in-memory SMs clock frequency is only 50% of the GPU clock frequency, on average the performance downgrade is 0.98% with all optimizations. This result indicates that a slower in-memory processor will not downgrade the overall offloading performance.

Double the warp capacity of in-memory SMs can averagely further improve the performance by 3.3% when applying all optimizations. ATAX and MVT are exceptions. With more warps offloading to in-memory SMs, and without run-time adaptation, the overall performance of these two applications is downgraded by around 2.63% (ATAX) and 7.50% (MVT) for Aggressive Loop Offload. This is because, without run-time adaptation, the majority of data traffic when offloading these two is inter-stack traffic.

5.4 Overhead discussion

In this section, we discuss the overhead that comes from our design and optimizations. Besides the overhead of restructuring (§ 3.3), we mainly discuss the hardware overhead introduced by our design, and the run-time overhead brought by our optimizations.

5.4.1 Estimated Area Overhead. In our architecture design, the area overhead mainly comes from the traffic history buffer to support run-time adaptation (§ 3.4). For each in-memory SM, the traffic history buffer introduces 384 registers (64 warps $\times 6$ registers/warp). We estimate it as 0.59% (384 / 65536) area overhead on the logic layer of each memory stack.

5.4.2 Optimization run-time overhead. Our optimizations could introduce extra run-time overhead as well. The main overhead comes from the runtime information tracking. Our in-memory processing design needs to track information like traffic status (for run-time adaptation optimization, § 3.4), or how many warps that can be offloaded to each Memory-SMs (for aggressive loop offloading optimization, § 3.5). In our design, we *asynchronously* track that information by hardware. Specifically, to track traffic status on in-memory SMs (§ 3.4), each time a memory access request is sent, the memory port can asynchronously send extra information to the Memory Traffic Monitor (§ 3.4). To track the available warps on each in-memory SMs, the GPU can asynchronously update this information when a warp is offloaded to Memory-SMs (in this case, one available offloading slot is taken), or a warp is returned from Memory-SMs (in this case, one taken offloading slot becomes available). Since information tracking operation is done by hardware asynchronously, we consider them only introducing negligible overhead on the critical path to the application.

5.5 Applications cannot improve

Our design cannot make all applications benefit from in-memory processing. After all, no designs can be perfect and solve all the problems. Figure 17 shows three examples. Neither previous offloading proposals nor our proposals could improve the overall performance of these applications.

For LIB and MC, we observe that these two applications present a high L2 cache hit rate on GPU. High L2-hit rate indicates that these two applications naturally cannot benefit from offloading to in-memory processing. Though they are also loop-dominated GPU applications, they are not the target applications for in-memory processing acceleration.

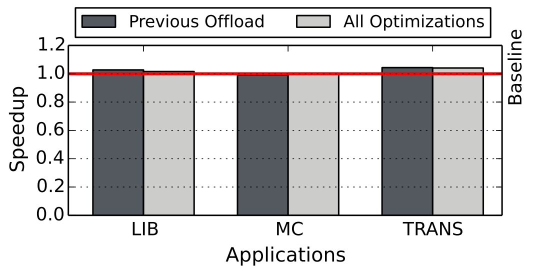

Figure 17: Applications cannot benefit from previous offloading, and our optimizations.

For TRANS, the majority of main memory accesses are combined with shared memory access in the same code block. Ideally, shared-memory transfer on-demand can exploit the benefit of offloading in Transpose. Unfortunately, in transpose shared memory is used as a *communication channel* between different threads, and for each loop, the size of Shared Memory consumed is the same as the memory traffic on the memory bus. Offloading transpose cannot save data traffic on the memory bus, thus no optimizations can improve performance.

In the current stage, our proposal cannot identify the access pattern similar to TRANS. This limitation can be solved with more detailed traffic analysis during the benefit analysis phase (§ 2). We plan to investigate it in the future.

6 RELATED WORK

In-memory processing on the GPU platform. Several previous works are exploring in-memory processing on the GPU platform [13, 16, 24, 32, 33]. TOP-PIM [33] performs a design space exploration upon the GPU in-memory processing system running HPC and graph processing applications. Xie et al. [32] show promising performance improvement for 3D rendering applications when accelerated with in-memory processing on GPU. Kim et al. [16] explore how to provide virtual memory supports and cache coherence supports without making any assumptions on the data layout. Our work differs from these works because we focus on optimizations that work for any *general* GPU applications, instead of domain-specific optimizations.

TOM [13] explores how to transparently (without programmers' involvement) decide which part of the code should be offloaded, and how to map multiple data arrays into multiple memory stacks. TOM adopts a static analysis method to decide which code block is worthy to offload, and it adopts a profiling method to learn the best data mapping for in-memory processing during runtime.

Our work is complementary to TOM. First, TOM focuses on *multiple data source* problem (e.g., multiple data arrays accessed by the offload candidate), while we focus on *consecutive memory access in the loop*. Second, TOM's offload decision is based on static offload-benefit analysis, and limited runtime information like the memory bus utilization and the in-memory SMs occupation. In this paper, we leverage dynamic information including in-memory SMs' utilization and in-memory SMs' traffic characteristics in the system decision making and adapt dynamically.

Pattnaik et al. [24] explore techniques to maximize in-memory processor utilizations. They try to maximize hardware utilization by concurrently schedule multiple GPU kernels to in-memory processors. They solved the resource management problem in in-memory processing. We solve a different problem. We try to address problems raised by current GPU application implementation. We answered the question of how to make GPU application more suitable to run on GPU in-memory processing systems.

In-memory processing on the CPU platform. In-memory processing on CPU platform has been explored for integrating variance of processing units with the conventional DRAM chip (e.g., [9, 30]), the emerging 3D-stacked memory devices (e.g., [1, 2]), or with the non-volatile memory (e.g., [5, 17]). These works focus on showing promising performance and energy consumption benefits for specific application types running on a CPU in-memory processing system. Our work differs from these works because our work focuses on the GPU host platform, where main memory bandwidth is a well known critical bottleneck for performance and energy consumption, and its applications are particularly data-intensive. Though one of our four issues (shared memory access) is specific to the GPU, the rest issues apply to the CPU.

Restructure GPU applications for performance improvement. There are works exploring how to optimize GPU application performance from the implementation point of view. Ryoo et al. [28] explore principles to optimize CUDA [22] GPU application implementation. Several NVIDIA tutorials [12, 18, 23] also show code optimization for specific applications or specific NVIDIA GPU architectures. Our work differs from them because of the difference in the hardware platform. We focus on restructure GPU applications to an optimal way for *in-memory processing*, which has different data transfer architecture compared to running on GPU. In-memory processing has different optimal memory access pattern requirements than GPU, thus our application restructuring optimizations adopt the principle of aggregating a warp's data into consecutive physical memory space.

7 CONCLUSION

In this paper, we explore four unsolved issues in the GPU in-memory processing systems. We propose four optimizations to mitigate the four issues respectively. Our solution first explores how to restructure GPU application implementation to better fit in-memory processing. Then, we explore how to leverage runtime information whether a warp should run on in-memory processors, and how many warps should run on them. Finally, we show our design to offload warps shared memory access.

From our evaluation with 13 applications, the four optimizations have achieved up to up to 2.23x performance improvement on GPU in-memory processing systems compared to previous offloading mechanisms. We conclude that the four optimizations can effectively resolve the four issues in the GPU in-memory processing system.

ACKNOWLEDGEMENT

We thank all anonymous reviewers for their valuable feedback. We thank members from the OPERA group and the Sysnet, UC

San Diego, for their valuable comments, suggestions, and proofreading on the draft versions of this paper. This work was supported by CRISP, one of six centers in JUMP, an SRC program sponsored by DARPA.

REFERENCES

[1] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2016. A scalable processing-in-memory accelerator for parallel graph processing. *ACM SIGARCH Computer Architecture News* 43, 3 (2016), 105–117.

[2] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. PIM-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture. In *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*. IEEE, 336–348.

[3] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE.

[4] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*. Ieee, 44–54.

[5] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. 2016. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. In *ACM SIGARCH Computer Architecture News*, Vol. 44. IEEE Press, 27–39.

[6] NVIDIA Corporation. 2018. NVIDIA TURING GPU ARCHITECTURE. https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[7] William James Dally and Brian Patrick Towles. 2004. *Principles and practices of interconnection networks*. Elsevier.

[8] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S Meredith, Philip C Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S Vetter. 2010. The scalable heterogeneous computing (SHOC) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. ACM.

[9] Joseph Gebis, Sam Williams, David Patterson, and Christos Kozyrakis. 2004. Viram1: A media-oriented vector processor with embedded dram. *DAC04* (2004).

[10] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. 2012. Auto-tuning a high-level language targeted to GPU codes. In *2012 Innovative Parallel Computing (InPar)*. Ieee.

[11] IMPACT Research Group. 2010. *Download Parboil*. http://impact.crhc.illinois.edu/parboil/parboil_download_page.aspx

[12] Mark Harris. 2019. *Optimizing Parallel Reduction in CUDA*. https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

[13] Kevin Hsieh, Eiman Ebrahimi, Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, Nandita Vijaykumar, Onur Mutlu, and Stephen W Keckler. 2016. Transparent offloading and mapping (TOM): Enabling programmer-transparent near-data processing in GPU systems. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 204–216.

[14] Adwait Jog, Onur Kayiran, Nachiappan Chidambaram Nachiappan, Asit K Mishra, Mahmut T Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R Das. 2013. OWL: cooperative thread array aware scheduling techniques for improving GPGPU performance. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 395–406.

[15] Stephen W Keckler, William J Dally, Brucek Khailany, Michael Garland, and David Glasco. 2011. GPUs and the future of parallel computing. *IEEE Micro* 31, 5 (2011), 7–17.

[16] Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, and Kevin Hsieh. 2017. Toward standardized near-data processing with unrestricted data placement for GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 24.

[17] Shuangchen Li, Cong Xu, Qiaosha Zou, Jishen Zhao, Yu Lu, and Yuan Xie. 2016. Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories. In *Proceedings of the 53rd Annual Design Automation Conference*. ACM, 173.

[18] Vishal Mehta and Maxim Milakov. 2018. OPTIMIZING CUDA APPLICATIONS FOR THE VOLTA/TURING ARCHITECTURE. http://on-demand.gputechconf.com/gtc-il/2018/pdf/sil8140-optimizing-cuda-applications-for-the-volta-turing-gpu-architecture.pdf.

[19] Paulius Micikevicius. 2012. GPU performance analysis and optimization. In *GPU technology conference*, Vol. 84.

[20] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N Patt. 2011. Improving GPU performance via large warps and two-level warp scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM.

[21] NVIDIA. 2019. *CUDA Toolkit 9.1 Download - Archived*. https://developer.nvidia.com/cuda-91-download-archive

[22] NVIDIA. 2019. *CUDA Zone*. http://developer.nvidia.com/object/cuda.html

[23] NVIDIA. 2019. Optimizing CUDA applications. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#optimizing-cuda-applications.

[24] Ashutosh Pattnaik, Xulong Tang, Adwait Jog, Onur Kayiran, Asit K Mishra, Mahmut T Kandemir, Onur Mutlu, and Chita R Das. 2016. Scheduling techniques for GPU architectures with processing-in-memory capabilities. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*. ACM, 31–44.

[25] Gennady Pekhimenko, Evgeny Bolotin, Nandita Vijaykumar, Onur Mutlu, Todd C Mowry, and Stephen W Keckler. 2016. A case for toggle-aware compression for GPU systems. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 188–200.

[26] Scott Rixner, William J Dally, Ujval J Kapasi, Peter Mattson, and John D Owens. 2000. Memory access scheduling. In *ACM SIGARCH Computer Architecture News*, Vol. 28. ACM, 128–138.

[27] Timothy G Rogers, Mike O'Connor, and Tor M Aamodt. 2012. Cache-conscious wavefront scheduling. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society.

[28] Shane Ryoo, Christopher I Rodrigues, Sara S Baghsorkhi, Sam S Stone, David B Kirk, and Wen-mei W Hwu. 2008. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. ACM, 73–82.

[29] Micron Technology. 2011. *4Gb: x4, x8, x16 DDR3 SDRAM*. 4Gb:x4,x8,x16DDR3SDRAM

[30] Josep Torrellas. 2012. FlexRAM: Toward an advanced Intelligent Memory system: A retrospective paper. In *2012 IEEE 30th International Conference on Computer Design (ICCD)*. IEEE, 3–4.

[31] Nandita Vijaykumar, Gennady Pekhimenko, Adwait Jog, Abhishek Bhowmick, Rachata Ausavarungnirun, Chita Das, Mahmut Kandemir, Todd C Mowry, and Onur Mutlu. 2015. A case for core-assisted bottleneck acceleration in GPUs: enabling flexible data compression with assist warps. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 41–53.

[32] Chenhao Xie, Shuaiwen Leon Song, Jing Wang, Weigong Zhang, and Xin Fu. 2017. Processing-in-memory enabled graphics processors for 3d rendering. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 637–648.

[33] Dongping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L Greathouse, Lifan Xu, and Michael Ignatowski. 2014. TOP-PIM: throughput-oriented programmable processing in memory. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. ACM.

[34] William K Zuravleff and Timothy Robinson. 1997. Controller for a synchronous DRAM that maximizes throughput by allowing memory requests and commands to be issued out of order. US Patent 5,630,096.