Using Shared Memory to Accelerate MapReduce on Graphics Processing Units

Feng Ji  
Department of Computer Science  
North Carolina State University  
Email: fji@ncsu.edu

Xiaosong Ma  
Department of Computer Science  
North Carolina State University  
&Computer Science and Mathematics Division  
Oak Ridge National Laboratory  
Email: ma@cs.ncsu.edu

Abstract—Modern General Purpose Graphics Processing Units (GPGPUs) provide high degrees of parallelism in computation and memory access, making them suitable for data parallel applications such as those using the elastic MapReduce model. Yet designing a MapReduce framework for GPUs faces significant challenges brought by their multi-level memory hierarchy. Due to the absence of atomic operations in the earlier generations of GPUs, existing GPU MapReduce frameworks have problems in handling input/output data with varied or unpredictable sizes. Also, existing frameworks utilize mostly a single level of memory, i.e., the relatively spacious yet slow global memory.

In this work, we attempt to explore the potential benefit of enabling a GPU MapReduce framework to use multiple levels of the GPU memory hierarchy. We propose a novel GPU data staging scheme for MapReduce workloads, tailored toward the GPU memory hierarchy. Centering around the efficient utilization of the fast but very small shared memory, we designed and implemented a GPU MapReduce framework, whose key techniques include (1) shared memory staging area management, (2) thread-role partitioning, and (3) intra-block thread synchronization. We carried out evaluation with five popular MapReduce workloads and studied their performance under different GPU memory usage choices. Our results reveal that exploiting GPU shared memory is highly promising for the Map phase (with an average 2.85x speedup over using global memory only), while in the Reduce phase the benefit of using shared memory is much less pronounced, due to the high input-to-output ratio. In addition, when compared to Mars, an existing GPU MapReduce framework, our system is shown to bring a significant speedup in Map/Reduce phases.

I. INTRODUCTION

Graphics Processing Units (GPUs), originally designed for graphics rendering, have become a popular choice for parallel computing. With the introduction of general-purpose GPU computing languages, such as CUDA [1], Brook [2], [3], CTM [4] and OpenCL [5], developing parallel computing programs on GPUs has become increasingly convenient. In consequence, these programming models provide programmers with an illusion of writing source code for a scalar thread as one does with CPU programming. For example, the SIMT (Single-Instruction, Multiple-Thread) [6] cores in GPUs are in nature SIMD cores, but the vector organization is hidden from application developers.

Essentially, GPUs and CPUs are very different in their hardware architecture, which must be taken into consideration for performance purpose. Specifically, GPUs have a complex memory hierarchy, which contains large register files, small but fast on-chip shared memory, and high-bandwidth off-chip global memory. Both levels of the memory are managed explicitly in software. Due to the latency-capacity tradeoff between them, data placement is difficult in GPU program tuning. In this work, we study this problem for GPU applications using the popular MapReduce model [7], a large category of GPU parallel codes.

MapReduce was first proposed by Google researchers as a parallel computation paradigm, as well as a runtime system in their distributed computing environment [7]. Later this elastic model was exploited for heterogeneous CPU processors [8], [9] and GPU processors [10]. In particular, Mars [10], a state-of-the-art MapReduce library, has shown advantage in throughput over its CPU counterpart for certain workloads. Beside performance benefits, general-purpose MapReduce frameworks like Mars simplify GPU parallel programming. Programmers can focus on correctness of their Map/Reduce function implementation, since the underlying scheduling, fault-tolerance, and performance issues are handled by the framework.

However, designing an efficient MapReduce framework on GPUs is still challenging for two reasons. First, many MapReduce workloads have irregular memory access patterns, which prohibits their efficient execution on GPUs. Besides, MapReduce is usually utilized for data parallel jobs, which often have quite large input and output data volume. Thus, MapReduce seemingly rules out shared memory due to its tiny size. In addition, MapReduce can take input and output with variable and unpredictable record sizes/counts, which becomes another major challenge for GPUs, which lack dynamic memory management or inter-thread synchronization mechanisms.

Second, though shared memory has been shown as able to accelerate many workloads on GPU, it is still unknown how it can be leveraged for MapReduce. Among related studies, Archuleta et al. first investigated the shared memory usage for input in temporal data mining [11], a MapReduce-like workload. However, general MapReduce workloads can be different in their varying record sizes, and using shared memory for output has not been examined. In addition,
Catanzaro et al. [12] examined on GPU a machine learning MapReduce workload. Both workloads mentioned above have output size/count known a priori. The aforementioned Mars framework [10] has most full-fledged support for general-purpose MapReduce workloads. Yet due to the absence of atomic operation in the previous generations of GPUs, memory management in Mars is based on off-line calculation using one extra pass of execution. Plus, the use of shared memory is minimum in Mars.

In this work, we investigate the data placement and transfer between shared and global memory for MapReduce applications running on GPUs. Our main contributions are as follows:

- We proposed a systematic approach to utilizing GPU shared memory as a transparent staging area to buffer MapReduce input/output data. To our best knowledge, this work is the first to explore the active use of shared memory in MapReduce execution on GPUs.
- We devised several mechanisms, including shared memory space management, thread-role partitioning, and intra-block thread synchronization, to handle the dynamic memory management issues, especially for output overflow handling in the staging area.
- We implemented the above techniques in a general-purpose MapReduce framework on GPUs, which needs no two-pass execution and makes no assumption on input/output data sizes, size variations, or record counts.
- We evaluated our framework extensively with five common MapReduce workloads, and compared our framework with Mars. The results indicate that it is worthwhile to exploit the small shared memory. Especially, output staging does bring a significant performance improvement. We also conducted comparison among input staging, texture buffering, and simply using global memory, and showed that our staging input mechanism performs better than or comparably to using the texture cache in most cases.

II. BACKGROUND

A. GPU

The architecture of a modern GPU [6] is shown in Figure 1. A GPU device has dozens of cores with duplicate ALUs and a three-level memory hierarchy. Each core, or multiprocessor (MP), consists of a group of scalar processors (SPs), a large register file equally split to the threads running on this MP, and shared memory, a piece of on-chip software-managed scratchpad memory. Shared memory is small (typically in the KB range) but fast (latency within dozens of cycles), and is shared by threads co-running on an MP. Global memory, in contrast, is off-chip and has a much larger size (typically in the GB range) and longer latency (400 - 700 cycles). Every several (e.g., 3 in Figure 1) MPs are grouped as a Texture Processing Cluster (TPC), equipped with a texture fetch unit and a 6KB-8KB per MP 2-level set-associative read-only texture cache [6], [13]. Texture cache is used for texture loading and only caches data read from the global memory space bound to texture buffers. Designed for streaming fetches, a hit in the texture cache does not decrease fetch latency, but reduces the global memory bandwidth demand. As a read-only cache, it does not ensure coherence with writes to the same global memory space bound to the texture buffer.

When a GPU kernel is issued, a large number of threads will be launched on multiple MPs in a device. These threads are grouped as blocks in a grid. The blocks are required to execute in any order, sequentially or in parallel, which requires them to be mutually independent. When a kernel starts, as many blocks are started as possible on available MPs; the rest wait for their completion to be scheduled. Every 32 threads inside a block form a warp and run in lockstep synchronously. A warp is the basic scheduling unit in GPU, with multiple co-running warps providing an MP the chance to hide the long latency of accessing global memory. When threads in the same warp follow different execution paths, warp divergence happens, where each path is executed in serial.

Programmers need to carefully optimize their memory access patterns to achieve better GPU performance. In particular, for global memory, accesses from a half-warp (the first or the second half of a warp) can be coalesced into one transaction if the visited addresses are within an aligned 32/64/128-byte segment. Otherwise up to 16 transactions will be issued. With new GPU architectures, accesses from one entire warp can be coalesced.

B. MapReduce

MapReduce [7] was proposed as a programming model for data processing jobs on large-scale clusters. With programs written in a functional style, the MapReduce runtime takes care of data partitioning, task scheduling, inter-machine communication, and fault tolerance. The MapReduce workflow consists of three phases: Map, Shuffle, and Reduce. Map and Reduce are two routines written by user programmers, who also provide specifications of input, intermediate results, and final output. When a job starts, the framework spawns a number of Map tasks, each working...
on a subset of input records and generating intermediate results. In the shuffle phase, the framework aggregates the intermediate results into key sets, each with a distinct key. Finally, the framework spawns a Reduce task per key set and generates final output records. Input, intermediate, and output are all in the format of \( <key, value> \) pairs, though the definition of `key` and `value` for each is the choice of programmers.

The data parallel computation model of MapReduce fits GPUs well. However, compared to traditional applications running on GPUs, MapReduce workloads with variable sized input or output create unique challenges to be tackled on GPUs. This is due to the lack of efficient dynamic memory management mechanisms, so far, in either shared memory or global memory. Mars [10], a state-of-the-art GPU MapReduce framework, uses a two-pass execution mechanism, requiring extra \textit{MapCount} and \textit{ReduceCount} phases before Map and Reduce phases respectively, to deal with memory management for output. The first pass, \textit{MapCount} or \textit{ReduceCount}, is only used to compute the output sizes of each task. At the end of this pass, a prefix summing operation is executed across all threads with output size values in order to find their own starting output address. This way, output data can be written subsequently in the second ("real") pass, to memory address ranges set appropriately by the prefix summing step. Due to the lack of atomic operations on previous generations of GPUs, such a two-pass mechanism turns out to be a practical approach for variable sized output. Part of our contribution in this paper is to remove the cost of such two-pass execution, by leveraging the atomic global memory access method and our shared memory management scheme. In addition, the way that we aggressively use shared memory alleviates global memory contention caused by global atomic operations.

III. EXPLOITING SHARED MEMORY FOR MAPREDUCE

A. Rationale

With the advent of atomic operations on GPU, having an appendable output buffer in global memory becomes realizable. However, due to the large number of concurrently running GPU threads, a global lock or a critical section for accessing appendable output buffers can become a bottleneck due to severe competition (we will show this in the next section across a few workloads). Therefore we propose using shared memory as input and output buffer for MapReduce workloads to relieve this contention. The benefit of using shared memory is listed as follows.

- Coalesced global memory transactions: when data are moved by our MapReduce framework between global memory and shared memory, GPU threads can follow strictly the coalesced global memory access pattern to attain the best memory bandwidth. MapReduce programmers need not manually coalesce memory accesses when providing Map and Reduce routines, since data is already buffered in shared memory in a transparent manner.
- Fast shared memory access: Map and Reduce routines can enjoy the short latency of shared memory. When access locality exists in the computation, fewer global memory requests will be generated.
- Reduced contention on result collection: having a temporary output buffer in shared memory forms a hierarchical result collection mechanism, which can remove the severe contention caused by single-level global memory result collection from a large number of threads.

An ideal workflow of MapReduce on GPU incorporating the use of shared memory, as shown in Figure 2, consists of 3 steps:

\textbf{Staging in:} All threads in a block cooperate on moving portions of input data from global memory to an input area within shared memory. In this step, without knowledge of records, these threads only see the data as a contiguous range of bytes in global memory. Using the start and end addresses of the range, neighboring threads from a warp always move contiguous primitive-type data to allow coalesced global memory accesses. Staging in action is repeated until the shared memory input area is full.

\textbf{Computing:} Each thread picks a record at a time, locating its address according to the directory indices (also staged into shared memory), and starts the user-provided Map/Reduce function on each record. Results are written into the output area, also in shared memory. The three input records shown in Figure 2 have 2, 3, and 2 bytes respectively, and generate three output records with sizes of 1, 3, and 2 bytes respectively.

\textbf{Staging out:} All the threads in the block cooperate on flushing output data from shared memory to global memory. Again, global memory accesses are coalesced.

The above three steps are repeated until all input records are processed. Here we assume global memory is large enough to hold the output, considering today’s global memory size typically at the GB level. Otherwise, batched processing is again possible at another level and it is possible to overlap GPU kernel execution with host-device data transfer. However, as mentioned earlier, a unique challenge for using the small shared memory for input/output staging

![Figure 2. Ideal Workflow of Map/Reduce Kernel empowered with Shared Memory. Step 1: Stage in; Step 2: Computing; Step 3: Staging out. (Solid line: record boundary; dotted line: bytes)](image-url)
is to coordinate the dynamic space usage. In particular, considering the lack of powerful inter-thread communication/synchronization schemes, as well as that MapReduce result sizes are often not known a priori, output overflow handling needs to be addressed carefully.

The complete view of our MapReduce framework on GPU to use shared memory for buffering input and output data is similar to the ideal workflow with 3 steps. Yet because of the possible overflow in the staging area, now it may have multiple computing and staging out steps per input iteration as shown in Figure 3. As an example, imagine a Map kernel launching with 256 threads per block, and in an input iteration, 118 input records are staged into the input area in the shared memory belonging to block 0. Then the first 118 threads will each pick one input record to start the user-provided Map function. During this process, these threads keep emitting intermediate results in the output staging area and may find that there is no space left. At that moment, all 256 threads will flush the collected results in the output area to an output buffer in global memory. After the flushing is done, the 118 threads will resume the Map computation until another overflow or the final flushing at the end of this iteration. The detailed mechanism of our framework will be given in the following sections.

B. Shared Memory Layout

First, we partition shared memory into an input and an output area. Figure 4(a) illustrate the shared memory layout in our design. The size ratio between the input and output areas is a parameter dependent on workloads. There is a tradeoff in that a larger input area increases computation concurrency by allowing more threads to work simultaneously due to more resident input records, while a larger output area leads to fewer overflows and less overflow handling overhead. One can roughly estimate the input and output ratio of a specific workload to decide the parameter value. In our future work, we plan to make this configuration automatic and adaptive.

In the shared memory input area, four separate buffers are allocated to store keys, values, key indices, and value indices. These buffers are mapped to contiguous segments in their corresponding data buffers in the global memory, so input staging benefits from coalesced memory accesses. A separate small working area is allocated to each thread, for the storage of temporary variables used in Map/Reduce computation. All these input buffers are statically managed before the computing step starts.

The output area, or result collection area (Figure 4(b)), is managed dynamically. We define a warp result as the collection of results (either intermediate results after Map or final results after Reduce) generated by the threads of one warp in one result record generation - threads in a warp run in lockstep, which makes their results generation occur at the same time. Output in MapReduce has the same format with input and contains two types of data: 1) size-predictable, structured key indices and value indices; and 2) size-unpredictable, unstructured key and value pairs. To efficiently use the shared memory space, we use a double-ended stack to store the size-predictable structured data from the left end and the size-unpredictable unstructured data from the right end. For example, in Figure 4(b), two warp results are shown. Each warp result has two arrays of pointers, i.e. key indices and value indices, on the left, among which only the first key index and value index are drawn here. Keys and values of the same warp result are stored contiguously on the right. Compared to using four separate buffers as in input area, which easily causes internal fragmentation, this design will only see overflows when the total remaining output space is smaller than a newly generated warp result. Unlike in the input area, the four types of data across all results are not stored all contiguously here. However, when flushing a warp result back to global memory, coalesced global memory writing is still attainable within the scope of one warp result.

C. Intra-block Thread Partitioning and Synchronization

GPU supports a great number of threads per block. However, the concurrency in computation is limited by the input area size and the input record granularity. Due to the
Therefore, we devised a mechanism to reduce the polling as many states exposed to software as CPU threads do. But GPU threads do not have running (blocked or sleep) state lest they compete for cycles.

In our framework, threads within a block are partitioned into compute threads, which carry out Map/Reduce computation, and helper threads, which remain idle during computation but cooperatively handle result overflows. To avoid warp divergence, we divide them between warps, so that a warp is either a compute warp or a helper warp. As the concurrency may not be a multiple of the warp size, we increase the number of compute threads to the nearest multiple of the warp size - the last few compute threads of the last compute warp may not get an input record.

It is not too hard to enable a producer-consumer type of synchronization between the compute and helper threads with multi-threading programs on a CPU. On GPUs, however, there lack lower-level primitives for inter-thread coordination. Essentially, though there is a block-wise barrier (__syncthreads()), no wait-signal type synchronization is available. Note that when threads enter the computation phase, the compute and helper threads branch into different execution paths. This makes the block-wise barrier inapplicable as its occurrence in branched paths leads to a hang or undefined result [6].

To solve this problem, we designed a software-based wait-signal primitive for synchronizations between the compute and helper threads. The condition is implemented with a set of flag variables in shared memory, each representing the status of one warp, considering the synchronous nature of its threads. We define wait group as the group of warps that wait on a condition, and signal group as the group of warps that signal on the condition.

Initially, all wait group and signal group flags are cleared. In a wait operation, the wait group threads poll at the signal flags until they see them all raised by signal group threads. Then the wait group warps change their own flags notifying the signal group that the signal is seen. When the signal group threads find all wait group warps in the “seen” state, they leave the signal routine and reset their flags. After setting their own “seen” state, the wait group threads check their peers’ states. The wait group warp which finds all its peers in the “seen” state, i.e., the last “seen” warp, resets all wait flags after seeing all signal flags clear, and leaves the wait routine; the other waiting group warps simply leave the wait routine when they find they are not the last one. Note that the size of the two groups must be known in advance. In our case, the partitioning of thread roles (compute vs. helper) is determined at the end of each input staging operation.

For efficiency we must let the helper threads enter a non-running (blocked or sleep) state lest they compete for cycles with the compute threads. But GPU threads do not have as many states exposed to software as CPU threads do. Therefore, we devised a mechanism to reduce the polling frequency. Since a global memory access has a large latency, the hardware scheduler swaps out the threads waiting for their operands ready, so that such global memory access latency can be hidden. With this we provide a yield operation for GPU threads by executing a dummy global memory read and write to a preallocated dummy global memory buffer. This will cause the calling threads to be scheduled off the MP.

Aside from synchronization, memory consistency is required: shared memory data updates by the signal group before signaling must be visible to the wait group when they see the raised flags. This requires processor consistency, a very weak consistency model which we have confirmed with Nvidia regarding their GPUs. Under this consistency model, all threads see writes (in shared memory in our case) from one thread in the same order as they were issued by that thread. Considering potential future architectural evolution to even weaker consistency models, a threadfence_block() at the entrance of the signal routine ensures the consistency requirement of our synchronization and adds an ignorable (<1%) performance overhead.

With our synchronization mechanism, the workflow can now have multiple output staging steps per input-computation-output iteration to handle output overflows, as illustrated in Figure 3. In the staging in and staging out steps, all threads participate in moving input or output between global memory and shared memory. When a compute step starts, only compute threads participate in the Map/Reduce computation, and the helper threads are “idle” (shown in dashed arrows), waiting on the overflow condition. When the helper threads see the signal of overflow, all threads start the staging output step. At the end of staging out, similar synchronization through another condition, overflow-handled, lets the compute threads continue with their execution and helper threads return to waiting.

D. Result Collection and Overflow Handling

We designed a hierarchical result collection mechanism, which happens both at the warp level and at the block level. At the warp level, results emitted by threads are required to be temporarily in an addressable place before being passed into our framework as references; they can be saved either at their original locations in the input area (e.g. Word Count has an output key as part of its input) or in the preallocated working area (e.g. Matrix Multiplication saves a floating point number as output per thread). These results will be grouped into one warp result and stored in the shared memory output area. To enable parallel output record writing, a prefix summing process is conducted in the warp to get the relative starting address for each thread’s output. Since a warp of threads run in lockstep, no explicit synchronization is required among them during this process.
At the block level, the first thread in a warp takes two actions atomically: 1) append the structured data portion of a new warp result from the left end of the result collection buffer, and 2) reserve the total size of the unstructured portion of the warp result from the right end. When this is done, all threads from this warp can freely update its own pointer indices at the left end, and copy its own output key-value record from the addressable place to the reserved area at the right end. Again no synchronization is needed.

An overflow happens when there is not enough space for a new warp result. Overflow handling is started with the participation of all threads within a block. When staging-out starts, the first thread of the block calculates the total size requirement for all the warp results collected in shared memory, atomically increases the length of results stored in global memory, and reserves the appropriate global memory address range for all warp results. Our wait-signal primitive is used again here to coordinate the operations among threads. When the reservation is done, all warps iteratively flush all warp results to the reserved area in global memory, each responsible for dumping one warp result at a time.

IV. Evaluation

A. Testbed

Our test bed is an AMD Athlon Dual-core 3800+ machine with 1GB main memory and an Nvidia GeForce GTX 280 card. GTX 280 card has 30 multiprocessor cores, each with 16384 32-bit registers, 16KB shared memory, and 1GB global memory. The OS is 64-bit Fedora 10 with Linux kernel 2.6.27, and Nvidia’s 64-bit Linux driver version 190.18.

B. Workloads

We select a set of five MapReduce workloads, shown in Table I. They involve both variable and fixed sized records and have been used for evaluating existing frameworks such as Mars [10]. Our implementation follows the same algorithms as in Mars, with several different memory usage models, to be detailed in Section IV-C.

Among the workloads, **Word Count (WC)** goes through a set of documents and calculates the number of occurrences for every distinct word. Each Map task takes a part of the input and emits a `<word, I>` pair for each word it sees. Each Reduce task takes one distinct key (word) and sums all the values sharing the same key into an output key-value pair. **Matrix Multiplication (MM)** calculates the product of two input matrices. Each Map task takes one row and one column from the two input matrices, respectively, and calculates the value of one element in the result matrix. No Reduce phase is needed for MM. **String Match (SM)** searches for a given keyword in a document. Each Map task takes a line and searches for the keyword. If a keyword is found, the line is emitted as a result. No Reduce phase is needed for SM. **Inverted Index (II)** goes through a set of html files and finds all URL links. Each Map task takes one part of the input, and searches for a link. Whenever it finds one, it emits the link as well as the link’s position in the document. Again no Reduce phase is needed for II. **KMeans (KM)** groups a set of vectors into $K$ clusters according to their distance in the space. Each Map task takes one vector and calculates its distance to $K$ centroid vectors of existing clusters, and then emits as an intermediate result the id of the nearest cluster and the vector itself. Each Reduce task takes one cluster, and computes its new centroid.

For all of our workload results, we tested with three problem sizes (see Table I), which are similar to those used in previous systems such as Mars. Table II summarizes the characteristics of these workloads, in terms of the mean and variance of record sizes for input, intermediate, and output, as well as the input-to-output record count ratios of the Map and Reduce phases. E.g., WC’s Map phase takes a string of characters, including words and punctuations, as an input key record. Such an input key has on average 32.44 bytes (with a standard deviation of 2.59). In addition, there is a 4-byte integer record index number as an input value record. WC generates words as intermediate result keys, with an average size of 5.46 letters and a standard deviation of 2.53, as well as a value “1” for each word. The input-to-output record count ratio means that there are 4.98 words per input record on average.

C. Memory Usage Modes Evaluated

We examine the effectiveness of using shared memory in different components of MapReduce computation, by evaluating the following combinations of memory usage options:

- **SI**: staging both input and output in shared memory, using the shared memory layout and data handling approach described in Section III.
- **SO**: staging only output in shared memory, but reading input directly from global memory. Intra-block thread synchronization is needed for output overflow handling.
- **SI**: staging input only in shared memory. Each thread is responsible for writing its own output record to global memory. In order to avoid extensive contention incurred by atomic accesses, only the first thread of each warp atomically increases the output size in global memory by the total size of all output records from its warp, calculated through in-warp prefix summing. This global

<table>
<thead>
<tr>
<th>Workload</th>
<th>Problem Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word Count (WC)</td>
<td>16MB / 32MB / 64MB</td>
</tr>
<tr>
<td>Matrix Multiplication (MM)</td>
<td>512x512 / 1024x1024 / 2048x2048</td>
</tr>
<tr>
<td>String Match (SM)</td>
<td>16MB / 32MB / 64MB</td>
</tr>
<tr>
<td>Inverted Indexing (II)</td>
<td>16MB / 32MB / 64MB</td>
</tr>
<tr>
<td>Kmeans (KM)</td>
<td>4MB / 16MB / 64MB</td>
</tr>
</tbody>
</table>

Table I

WORKLOADS AND PROBLEM SIZES
memory range is then broadcast to other threads in the warp through shared memory.

- **G**: no staging. This mode is similar to the Mars scheme. However, it avoids the two-pass execution by using atomic operations in the same way as SI does.

The wait-signal intra-block synchronization primitive is only used when we stage output in shared memory (in SIO and SO). Also note that at least a warp (32 threads) are needed to enable intra-block synchronization. In SI and SIO, both with input staged, the number of compute threads running concurrently is subject to the thread block size (the number of threads within a block) and the input area size.

Side comparisons exist between using the texture cache and the SI or G modes. Staging output cannot be compared since the texture cache is readonly. Therefore, in addition to the four memory usage modes mentioned above, we have implemented a GT mode, which is similar to G, but binds the input in global memory to texture buffers. Please note that this mode requires a different version of user-provided Map/Reduction functions because their general memory accesses, i.e., pointer operations, in global memory or shared memory, must be replaced with explicit texture fetch directives.

Note that the MM Map kernel is different in that only the indices for a row/column vector can be staged into shared memory. Otherwise, the huge record (a row or column vector of a matrix) will reduce the concurrency to fewer than 8 threads, due to the small size of shared memory.

For Reduce phase, we implemented and evaluated two reduction strategies: thread-level reduction (TR) and block-level reduction (BR) [11]. With TR, each thread works on a distinct key set by running user-provided sequential code. With BR, a block of threads work on a distinct key set in parallel, by following a predefined reducing order, e.g., a tree-like order that reduces two children values at every node of the tree. Among existing MapReduce frameworks, TR is adopted by Mars and Hadoop [14], while BR is adopted in Catanzaro’s MapReduce framework on GPU [12]. Note that the TR implementation cannot stage input, as by definition it processes a complete key set at a time, which can be arbitrarily large. Map input, on the other hand, has reasonably small records to be processed one at a time, with the exception of workloads like MM. Besides, for BR kernels, we cannot implement the GT mode. This is because BR kernels update values in their original places, which becomes a problem as the texture cache coherence is not guaranteed with global memory writes in one kernel invocation.

### D. Map Results

Figure 5(a)-5(e) show the results of Map kernels with different memory usage modes in the large problem sizes. The small and medium problem size results possess similar trends and are omitted due to the space limit.

From these experiments, we observe that utilizing GPU shared memory through input/output data staging does bring a significant improvement to the performance of Map kernels. In particular, SIO, by combining input and output staging, delivers the best or close-to-best performance in most cases. Compared to G, SIO achieves an average 2.85x speedup, with a maximum of 7.5x.

However, the figures also reveal that the source of such performance benefit heavily depends on workload characteristics. For WC and SM, staging output makes more difference, while for II, staging input creates a larger impact. For KM, it seems that only by combining SO and SI can we receive a significant improvement.

With WC and SM, SO brings about large performance gains (in most cases more than 2x speedup compared to G). This is due to their large number of Map output results. With frequent result generation, when output is not staged through shared memory, there is severe contention over a critical section involved in atomic global memory updates, for both output data and associated indices. Though we have reduced this contention by performing in-warp prefix summing, G and SI both suffer from the overhead and unscalability of this bottleneck. In particular, with these two workloads, both G and SI produce longer Map execution time when the number of threads per block increases, while SO and SIO benefit from the increased concurrency brought by more threads. WC, due to its large output-to-input ratio, receives more performance gain than SM from SO, while SM does benefit slightly from SI as it has more access locality when processing the input data.

II benefits significantly and solely from staging input. This can be attributed to several of its traits: 1) very large

### Table II

**MapReduce Workloads Characteristics**, which shows the statistics collected with the "large" problem size. (1. All keys and values represent mean / standard deviation of record sizes. 2. Ratios are input:output record count ratios. 3. '-' means there is no Reduce phase.)

<table>
<thead>
<tr>
<th>Workload</th>
<th>Input Key</th>
<th>Input Value</th>
<th>Map Ratio</th>
<th>Intern Key</th>
<th>Intern Value</th>
<th>Reduce Ratio</th>
<th>Output Key</th>
<th>Output Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>WC</td>
<td>32.44 / 2.59</td>
<td>4 / 0</td>
<td>14.98</td>
<td>5.46 / 2.53</td>
<td>4 / 0</td>
<td>68.21 / 1</td>
<td>9.01 / 3.11</td>
<td>4 / 0</td>
</tr>
<tr>
<td>MM</td>
<td>8192 / 0</td>
<td>8192 / 0</td>
<td>1.1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>8 / 0</td>
<td>4 / 0</td>
</tr>
<tr>
<td>SM</td>
<td>44.52 / 2.68</td>
<td>4 / 0</td>
<td>3.83.1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>4 / 0</td>
<td>4 / 0</td>
</tr>
<tr>
<td>II</td>
<td>8 / 0</td>
<td>63.9 / 123.2</td>
<td>7.94.1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>31.67 / 17.34</td>
<td>8 / 0</td>
</tr>
<tr>
<td>KM</td>
<td>0 / 0</td>
<td>32 / 0</td>
<td>1.1</td>
<td>4 / 0</td>
<td>32 / 0</td>
<td>69905:1</td>
<td>4 / 0</td>
<td>32 / 0</td>
</tr>
</tbody>
</table>
The marginal decrease in G execution time gets smaller, losing to SIO by at least a factor of 1.5.

Though helpful in most cases, compared to SI, which never degrades performance, SO may bring new overhead and can even hurt the overall Map execution time sometimes (e.g., with II). Such overhead comes from two aspects: 1) Even with our synchronization mechanism based on dummy global memory accesses, the helper threads are still performing busy waiting during the computation phases. Such an overhead is more pronounced when the computation step is longer, as more busy waiting operations are conducted. 2) Early finished compute threads are busy waiting on their peers, to participate in potential overflow handling in the future. For the former, the overhead can be reduced by shortening the computation phases. Note that staging input can effectively reduce the computation phase for most cases, so SIO is usually better than or equal to SO duration. For the latter, the variance in input record sizes is the primary reason for the uneven Map computation time (e.g., II-M).
KM possesses certain common characteristics with both WC and II, making it benefit most from the combination of SI and SO. On one hand, it has fixed sized input/output records, with an input/output size ratio close to 1. Due to the substantial output sizes, SO helps to relieve the problem of unscalable global memory accesses seen in G, especially with large thread blocks. On the other hand, it has strong access locality, by repeatedly accessing an input vector for each centroid, making staging input desirable. Most importantly, by leveraging such locality and shortening the computation phases with input staged into shared memory, the busy waiting overhead of staging output is reduced, making SIO a clear winner.

Finally, MM reads data anyway from global memory, bringing the four modes closer in performance. Further, as the workload is memory-bound, all four modes show similar unscalability with increased thread count per block. Small as the difference is, staging output can bring certain advantage here, when the thread block is large. With a block size of 64, the benefit of SIO and SO is offset by the fact that they have to leave a warp of 32 threads as helper threads, which halves the threads available for computation.

As for the effect of texture cache, at the cost of modification to Map function code, GT never loses to G. Comparing SI to GT, except for MM-M, where input cannot be completely staged, staging input is better or equal to texture in most cases. While WC-M and SM-M have similar results, opposite results are shown in II-M and KM-M. While the texture cache is designed to reduce the global memory bandwidth demand, its latency is still longer than shared memory. Therefore when there are long, complex computation phases with conditional branches and a large variance in input as in II-M, bandwidth is not a problem, and the short latency of shared memory makes SI much better. But in KM-M, featured with fixed-size input and almost equally long computation for every thread, the latency of texture fetch is well hidden and the GT mode wins because hardware cache brings the slightest overhead compared to explicit staging. However SI gradually catches up with GT when the thread block is larger, showing that the overhead of SI is finally hidden with more warps. Besides, MM-M’s GT mode shows superior performance over SI because in GT, row/column vectors can be cached with the hardware-managed replacement policy, while SI can only stage the row/column indices.

E. Reduce Results

Figure 5(f)-5(i) shows the Reduce kernel execution time for the two workloads with Reduce phase, WC and KM, with two different reduce strategies.

Overall, here staging input/output does not appear as helpful as for Map kernels, and we see less diversity in behavior across different workloads/strategies. In the majority of cases, G (or GT when texture is applicable) works the best. As can be seen from Table II, the input-to-output ratio is quite large. Therefore, the benefit of staging output through shared memory cannot offset its overhead, and G (or GT) outperforms SO in all cases.

Staging input does appear to generate a considerable benefit in the case of KM-BR, by an average speedup of 2.25x over G. Here the size of each value type, i.e. a vector, is dependent on the dimension of the space and usually large enough to reach dozens of bytes. With G, multiple global memory transactions are needed for accessing each vector: data accessed for a half-warp at a time span across several 128-byte segments. Staging input improves performance by reducing the number of global memory transactions. This does not happen to WC, as the value type there, integer, makes global memory access already coalesced.

Comparing TR and BR, it appears that BR works better for KM, while TR performs better for WC. The reason lies in their input characteristics. KM has dozens of large distinct key sets, while WC has 10,000s of small ones. Therefore, TR achieves more parallelism with WC across key sets and BR with KM within each key set. This agrees with the finding from temporal data mining [11] that BR works better on small problem sizes, and TR on large problem sizes. The rather small number of key sets in KM, which limits the maximum parallelism of TR kernels, also explains the scalability problem observed in KM-TR. The G mode’s performance is flat, as an increased block size only makes more threads idle in the block. The SO mode’s performance degrades as more threads waste cycles in synchronization, but brings no benefit due to the high input-to-output ratio.

Texture cache usage for two TR kernels gets an improvement of around 1.27x for WC-TR and around 1.12x for KM-TR over the G mode. Optimized for throughput requirement, the texture cache brings more benefit to WC-TR than to KM-TR. This is again caused by the aforementioned characteristics of the two TR kernels. WC-TR has a higher degree of parallelism, and hence more potential to take advantage of the texture cache.
F. Comparison with Mars

In this section, we compare our approach with Mars [10], the state-of-the-art, general-purpose MapReduce framework on GPU known by us.

Our framework and Mars share the same data transmission between host and device, as well as the same shuffle phase. The same Map and Reduce algorithms are implemented except for the Reduce phase of Word Count. WC in Mars does not have a Reduce phase since the number of occurrence of each word can be found as the number of records in each distinct key set after the shuffle phase. Yet for the purpose of making a fair comparison, we extended WC in Mars by 1) enabling the Reduce phase, and 2) providing a Reduce function that follows a commonly accepted algorithm [7].

For the two workloads that have a Reduce phase, as Mars supports only thread-level reduction (TR), we only compare the TR implementation in our framework. For each memory usage model, we use the optimal thread block size found in the state-of-the-art, general-purpose MapReduce framework.

First, we take a look at the impact of the five memory usage modes, enabled by our framework, on the Map and Reduce kernel performance. Figure 7 shows the speedup over Mars brought by all the memory usage modes in our framework. In most cases, even without using shared memory, our framework outperforms Mars by avoiding the two-pass execution. The G mode beats Mars by a factor of 2 at most (1.1 on average). The reason for the negative speedup in WC and SM is that both workloads have large quantities of results. In this situation, the two-pass running is better, which avoids the bottleneck in atomic updates in the global memory.

The G mode also delivers better performance for the two Reduce kernels, compared to Mars as well as to the SO mode. For the Map kernel, however, more advantage is obtained by our active use of shared memory as a staging area. In particular, with the help of output staging, the SO and SIO modes significantly outperform Mars, where G alone is not effective, as discussed above. The SIO modes, which has been shown in Section IV-D to be overall effective, achieves a speedup between 1.3x and 3.73x over Mars, with an average of 2.67x. This demonstrates that SIO appears to be a good choice for Map kernels.

Then we examine the total MapReduce execution time of the compared systems (Mars, and the five memory usage modes with our framework), as shown in Figure 6. Each stacked bar gives the breakdown of separate phases in MapReduce: the Map, shuffle, and Reduce phases, as well as the I/O time spent in transferring data back and force between the GPU global memory and the CPU host memory. Here we show results from all three problem sizes that we tested: small, medium, and large. Note the reduce phases in SI and SIO modes are actually that of G and SO respectively as input staging does not apply to the TR kernels.

The complete speedup is dampened by the portion of non-Map/Reduce phases in the end-to-end elapsed time. Word Count is an example where there is a large shuffle phase. Still, overall the G mode and SIO mode in our framework outperforms Mars by 34% and 64% on average, respectively. Yet a better approach is to adopt different memory modes in different phases adaptively, which we plan to explore in
our future work. There are visible differences in the I/O time between Mars and our framework, due to that the data definition and indices can have slight difference.

G. Synchronization on GPU

Finally, we examine the effectiveness of our intra-block thread synchronization scheme, which allows waiting threads to yield using dummy global memory accesses. Figure 8 shows the benefit of such a scheme in busy waiting. Compared with the case where the waiting helper threads never yield, the kernel execution time improves between $-1.2\%$ and $13\%$ for all SIO Map kernels but MM-M: these kernels never go to global memory to fetch input data in the computation phase. Note that the benefit starts to appear after there are 128 threads within a block, when the GPU processor resource utilization is high enough, and the performance gain increases when the thread count per block grows. Yielding is especially beneficial for II-M, which has long and complex computation phases.

V. RELATED WORK

MapReduce [7] was first proposed as a parallel programming model for distributed parallel computing. In the past several years it has attracted much attention as an elastic programming model to develop large-scale parallel processing jobs, and has been studied in other parallel computing environments. Phoenix [8] proves MapReduce can be used to program shared memory multi-core and multiprocessor systems. Merge [15] uses MapReduce as a high-level language to program heterogeneous systems with both CPU and accelerators. Similarly, Sequoia [16] borrows MapReduce and builds memory hierarchy awareness into language support for portable parallel programs. On specialized processors, De Kruijf et al. presents a MapReduce system on Cell BE processor [17], and Papagiannis et al. revamps the design to minimize the control overhead and avoid redundant memory copies [18]. CellMR [9], [19] builds MapReduce on Cell BE processors in a cluster setting, and shows the benefit of streaming data across cluster nodes and asymmetric cores. On GPUs, besides Mars [10], which has been discussed and evaluated earlier in this paper, Jiang et al. and Ravi et al. propose generalized reduction as an alternative API to MapReduce on Multi-core CPU and GPU [20], [21]. Other work on GPU [11], [12] have also applied the MapReduce model to their algorithms. Among the above, our work is most similar to Mars as a general-purpose MapReduce framework for GPUs, though we are the first to explore the active use of shared memory in the GPU MapReduce workflow.

Most recently, MapCG [22] was proposed to abstract programming on both CPU and GPU with the MapReduce model, whose advantage over Phoenix [8] (CPU only) and Mars [10] (GPU only) was shown. For the GPU side, the authors also addressed the problem of two-phase running in Mars by using a light-weight memory allocator, which is similar to our G mode. Yet they did not use shared memory aggressively and their performance gain over Mars is primarily from building a hash table in the Map phase and replacing sorting with hash table lookups, which can be leveraged in our framework in the future.

As for synchronization on GPU, Xiao et al. [23] studies 3 different inter-block communication mechanisms on GPU. Our work on intra-block synchronization targets a different scope within one thread block and is complementary to their work. About explicitly managed shared memory, Silberstein et al. [24] propose a software-managed cache layer for sum-product algorithms. Our design of shared memory management is similar, but is tailored for one dimensional data buffer and the overflow handling problem.

Because of its SIMD nature and novel memory subsystem design, GPU performance optimization brings new challenges. Ryoo et al. [25], [26] studied CUDA program optimizations and showed several principles for effective usage of the GPU memory subsystem. We followed these rules in developing our own prototype framework for MapReduce workloads.

Aside from a rich body of work on using GPU for specific algorithms, recently more research interest is attracted to runtime and programming language support for GPU-enabled systems. Mars [10] and our work fall into this category. Ma develops a translational framework for data mining algorithms on GPU [27]. Sundaram et al. and Satish et al. use domain templates expressed in graphs to attack the problem of total data size not fitting on GPU and minimizing data transfer between host and GPU [28], [29]. Our framework complements them by addressing efficient use of the GPU shared memory.

In addition, Zhang el al. proposes to use idle CPU cycles for thread reference redirection and data layout transformation to avoid GPU threads divergence [30]. Targeting a different problem, our technique of thread roles partitioning shows some resemblance to their thread reference redirection; the difference is that we have all things done at the GPU side.
VI. CONCLUSIONS AND FUTURE WORK

In this paper, we discussed our approaches to overcoming the hurdles of enabling the aggressive utilization of GPU shared memory in MapReduce computation. We present a MapReduce framework that adaptively manages the limited shared memory space without requiring programmers to explicitly consider data placement in the GPU memory hierarchy. From our evaluation, we have found that despite its small size, the shared memory is worth exploiting and in particular, there is in most cases a significant performance gain when it is used to stage output data. Also, the Map phase of MapReduce workloads receives more benefit from using shared memory, mainly due to larger output sizes.

We plan to extend this work in the future. Most importantly, we will investigate solutions to make our framework autonomous, by enabling it to automatically and transparently adapt to individual workloads. Leveraging empirical observations and general guidelines found in this work, an intelligent MapReduce framework should be able to perform runtime, automatic configuration of parameters such as the shared memory space partition sizes and the thread block size. We also plan to extend our work to the newer GPU architecture, which has a global memory cache and offers a large size. We also plan to extend our work to the newer GPU architecture, which has a global memory cache and offers a large size. We also plan to extend our work to the newer GPU architecture, which has a global memory cache and offers a large size. We also plan to extend our work to the newer GPU architecture, which has a global memory cache and offers a large size. We also plan to extend our work to the newer GPU architecture, which has a global memory cache and offers a large size. We also plan to extend our work to the newer GPU architecture, which has a global memory cache and offers a large size. We also plan to extend our work to the newer GPU architecture, which has a global memory cache and offers a large size. We also plan to extend our work to the newer GPU architecture, which has a global memory cache and offers a large size. We also plan to extend our work to the newer GPU architecture, which has a global memory cache and offers a large size. We also plan to extend our work to the newer GPU architecture, which has a global memory cache and offers a large size. We also plan to extend our work to the newer GPU architecture, which has a global memory cache and offers a large size. We also plan to extend our work to the newer GPU architecture, which has a global memory cache and offers a large size. We also plan to extend our work to the newer GPU architecture, which has a global memory cache and offers a large size. We also plan to extend our work to the newer GPU architecture, which has a global memory cache and offers a large size. We also plan to extend our work to the newer GPU architecture, which has a global memory cache and offers a large size. We also plan to extend our work to the newer GPU architecture, which has a global memory cache and offers a large size. We also plan to extend our work to the newer GPU architecture, which has a global memory cache and offers a large size. We also plan to extend our work to the newer GPU architecture, which has a global memory cache and offers a large size. We also plan to extend our work to the newer GPU architecture, which has a global memory cache and offers a large size. We also plan to extend our work to the newer GPU architecture, which has a global memory cache and offers a large size.

ACKNOWLEDGMENT

We greatly appreciate the anonymous reviewers for their valuable comments and suggestions. We also thank He-shan Lin, Frank Mueller, and Vincent Freeh for helpful discussions and feedback. This work has been sponsored in part by an NSF CAREER Award (CNS-0546301), an NSF award (CNS-0915861), Xiaosong Ma’s joint appointment between NCSU and ORNL, and the K. C. Wong Education Foundation, Hong Kong.

REFERENCES