Optimizing MapReduce for GPUs with Effective Shared Memory Usage

Linchuan Chen Gagan Agrawal
Department of Computer Science and Engineering
The Ohio State University
Columbus, OH 43210
{chenlinc,agrawal}@cse.ohio-state.edu

ABSTRACT

Accelerators and heterogeneous architectures in general, and GPUs in particular, have recently emerged as major players in high performance computing. For many classes of applications, MapReduce has emerged as the framework for easing parallel programming and improving programmer productivity. There have already been several efforts on implementing MapReduce on GPUs.

In this paper, we propose a new implementation of MapReduce for GPUs, which is very effective in utilizing shared memory, a small programmable cache on modern GPUs. The main idea is to use a *reduction-based* method to execute a MapReduce application. The *reduction-based* method allows us to carry out reductions in shared memory. To support a general and efficient implementation, we support the following features: a memory hierarchy for maintaining the reduction object, a *multi-group* scheme in shared memory to trade-off space requirements and locking overheads, a general and efficient data structure for the reduction object, and an efficient swapping mechanism.

We have evaluated our framework with seven commonly used MapReduce applications and compared it with the sequential implementations, MapCG, a recent MapReduce implementation on GPUs, and Ji *et al.*'s work, a recent MapReduce implementation that utilizes shared memory in a different way. The main observations from our experimental results are as follows. For four of the seven applications that can be considered as *reduction-intensive* applications, our framework has a speedup of between 5 and 200 over MapCG (for large datasets). Similarly, we achieved a speedup of between 2 and 60 over Ji *et al.*'s work.

Categories and Subject Descriptors

D.1.3 [**Software**]: PROGRAMMING TECHNIQUES—*Concurrent Programming*

General Terms

Performance

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
HPDC'12, June 18–22, 2012, Delft, The Netherlands.
Copyright 2012 ACM 978-1-4503-0805-2/12/06 ...$10.00.

Keywords

GPU, MapReduce, shared memory

1. INTRODUCTION

The work presented in this paper is driven by two recent but independent trends. First, within the last 3-4 years, GPUs have emerged as the means for achieving *extreme-scale*, *cost-effective*, and *power-efficient* high performance computing. On one hand, some of the fastest machines in the world today are based on NVIDIA GPUs. At the same time, the very favorable price to performance ratio offered by the GPUs is bringing supercomputing to the masses. It is common for the desktops and laptops today to have a GPU, which can be used for accelerating a compute-intensive application. The peak single-precision performance of a NVIDIA Fermi card today is more than 1 Teraflop, giving a price to performance ratio of $2-4 per Gigaflop. Yet another key advantage of GPUs is the very favorable power to performance ratio.

Second, the past decade has seen an unprecedented data growth as information is being continuously generated in digital format. This has sparked a new class of high-end applications, where there is a need to perform efficient data analysis on massive datasets. Such applications, with their associated data management and efficiency requirements, define the term *Data-Intensive SuperComputing* (DISC) [3]. The growing prominence of data-intensive applications has coincided with the emergence of the MapReduce paradigm for implementing this class of applications [6].

The MapReduce abstraction has also been found to be suitable for specifying a number of applications that perform significant amount of computation (e.g. machine learning and data mining algorithms). These applications can be accelerated using GPUs or other similar heterogeneous computing devices. As a result, there have been several efforts on supporting MapReduce on a GPU [4, 12, 13].

A GPU is a complex architecture and often significant effort is needed in tuning the performance of a particular application or framework on this architecture. Effective utilization of *shared memory*, a small programmable cache on each multi-processor on the GPU has been an important factor for performance for almost all applications. In comparison, there has only been a limited amount of work in tuning MapReduce implementations on a GPU to effectively utilize shared memory [15].

This paper describes a new implementation of MapReduce for GPU, which is very effective in utilizing shared memory. The main idea is to perform a *reduction-based* processing of a MapReduce application. In this approach, a key-value pair that is generated is immediately merged with the current copy of the *output results*. For this purpose, a *reduction object* is used. Since the memory requirements for the output results or the reduction object is much smaller than the requirements for storing all key-value pairs for most applications, we reduce the runtime memory requirements very significantly. This, in turn, allows us to use the shared memory effectively. For many applications, the reduction object can be entirely stored in the shared memory.

Many challenges have been addressed to create a general and efficient implementation of our approach. A mechanism for dynamic memory allocation in the reduction object, maintaining a memory hierarchy for the reduction object, use of a *multi-group* strategy, and correct swap between shared and device memory for portions of the reduction object are some of the challenges we addressed.

We have evaluated our implementation comparing it against sequential implementations, MapCG [13], and a recent implementation that does use shared memory [15]. The main observations from our experiments are as follows. For four applications that can be considered as *reduction-intensive* applications, our framework has a speedup of between 5 and 200 over MapCG (for large datasets). Similarly, we achieved a speedup of between 2 and 60 over Ji *et al.*'s work for the three applications that are available from their distribution.

2. BACKGROUND

This section provides background information on GPU architectures and MapReduce.

2.1 GPU Architecture

A modern Graphical Processing Unit (GPU) architecture consists of two major components, i.e., the processing component and the memory component. The processing component in a typical GPU is composed of a certain number of streaming multiprocessors. Each streaming multiprocessor, in turn, contains a set of simple cores that perform in-order processing of the instructions. To achieve high performance, a large number of threads, typically a few tens of thousands, are launched. These threads execute the same operation on different sets of data. A block of threads are mapped to and executed on a streaming multiprocessor. Furthermore, threads within a block are divided into multiple groups, termed as *warp*. Each warp of threads are co-scheduled on the streaming multiprocessor and execute the same instruction in a given clock cycle (SIMD execution).

The memory component of a modern GPU-based computing system typically contains several layers. One is the *host memory*, which is available on the CPU main memory. This is essential as any general purpose GPU computation can only be launched from the CPU. The second layer is the *device memory*, which resides on the GPU card. This represents the global memory on a GPU and is accessible across all streaming multiprocessors. The device memory is interconnected with the host through a PCI-Express card (version can vary depending upon the card). This interconnectivity enables DMA transfer of data between host and device memory. From the origin of CUDA-based GPUs, a scratch-pad memory, which is programmable and supports high-speed access, has been available private to a streaming multiprocessor. The scratch-pad memory is termed as *shared memory* on NVIDIA cards. Till recently, the size of this shared memory was 16 KB.

2.2 MapReduce

MapReduce [6] was proposed by Google for application development on data-centers with thousands of computing nodes. It can be viewed as a middleware system that enables easy development of applications that process vast amounts of data on large clusters. Through a simple interface of two functions, *map* and *reduce*, this model facilitates parallel implementations of many real-world tasks, ranging from data processing for search engine support to machine learning [5, 9].

MapReduce expresses the computation as two user-defined functions: *map* and *reduce*. The *map* function takes a set of input instances and generates a set of corresponding intermediate output $(key, value)$ pairs. The MapReduce library groups together all of the intermediate values associated with the same key and shuffles them to the *reduce* function. The *reduce* function, also written by the users, accepts a key and a set of values associated with that key. It merges together these values to form a possibly smaller set of values. Typically, just zero or one output value is produced per *reduce* invocation.

The main benefits of this model are in its simplicity and robustness. MapReduce allows programmers to write functional style code that is easily parallelized and scheduled in a cluster environment. One can view MapReduce as offering two important components [19]: a *practical programming model* that allows users to develop applications at a high level and an *efficient runtime system* that deals with the low-level details. Parallelization, concurrency control, resource management, fault tolerance, and other related issues are handled by the MapReduce runtime.

Recent years have seen a number of efforts to implement MapReduce frameworks on GPUs. Mars [12] was the first known MapReduce framework on a GPU. It spends extra time on counting the size of intermediate results. MapCG [13] has been shown to outperform Mars. It proposed a light-weight memory allocator, which enables efficient dynamic memory allocation on the GPU. Also, it used a hash table to store the key-value pairs, which avoids the overhead of shuffling.

Ji *et al.* have proposed a MapReduce framework which utilizes the shared memory on a GPU [15]. They used shared memory as a buffer to stage the input and output of *map* and *reduce* stages. The shared memory buffer enables fast data access and coalesced I/O to the device memory. They still store all key-value pairs generated by the *map* stage, and shuffling is required, which is time consuming. GPMR [21] was a project to leverage the power of GPU clusters for MapReduce. In this project, *partial* reduction and accumulation were used to reduce the communication overhead between different GPUs. However, shared memory was not used during reduction or accumulation to improve efficiency of processing on a single GPU.

3. SYSTEM DESIGN

This section describes the implementation of our system. The main emphasis in our approach is using the shared memory of a GPU effectively.

Initially, we describe why it is both important and challenging to effectively utilize shared memory while executing a MapReduce application. As we stated previously, shared memory supports much faster read and write operations. In addition, execution of an application with the MapReduce not only needs read/write operations, but also a large number of atomic operations, which are used to synchronize data accesses and updates by different threads. There is an even greater difference between the performance of atomic operations on shared memory and their performance on device memory. Thus, a large number of atomic operations on device memory can greatly undermine the performance of a MapReduce application.

The key challenge, however, in effectively utilizing shared memory is that the capacity of shared memory is very limited. Most MapReduce applications generate a large number of intermediate *key-value* pairs at the end of the Map phase. For these applications, it is impossible to keep all the key-value pairs in shared memory.

A recent implementation of MapReduce has demonstrated one mechanism for using shared memory. Particularly, the work from Ji *et al.* involves an innovative approach to using shared memory as a buffer to stage the input and output of both the *map* and *reduce* stages [15]. They copy the input from the device memory in a coalesced pattern into the shared memory. During the processing from a particular stage, when the shared memory is full, they copy the output from shared memory to device memory, again using coalesced writes. In this way, they achieve a speedup over the implementation that uses device memory only. However, the key-value pairs from the *map* stage still need to be stored in device memory, and shuffling is also required.

In our framework, we use a distinct approach for exploiting shared memory. We implement a MapReduce framework in a *reduction-based* manner, which is an alternative to the traditional implementation methods. Our focus is on exploiting shared memory effectively for *reduction-intensive* applications. By reduction-intensive application, we imply an application that generates a large number of key-value pairs that share the same key. Thus, there is a significant amount of work performed during the *reduce* phase, Moreover, the reduction operation is associative and commutative, which gives us flexibility in executing the reduction function.

Before describing our system in details, we first discuss the idea of the reduction-based MapReduce.

3.1 Basic Idea: Reduction Based Implementation

In reduction-based MapReduce, the *map* function, which is defined by the users, works in the same way as in the traditional MapReduce. It takes one or more input units and generates one or more key-value pairs. However, the key-value pairs are handled in a different way. In the traditional MapReduce, the key-value pairs are first stored, and after all the input units have been processed, the MapReduce library groups together all of the intermediate values associated with the same key and passes them to the *reduce* function. The *reduce* function each time accepts a key and a set of values associated with that key. It merges these values in the way defined by the users. In some cases, a user may define a *combine* function, which can combine key-value pairs generated on the same node, and reduce interprocess communication. However, the memory requirement on each node is typically not reduced by the combine function.

The reduction-based method, which can only be used if the reduction function is associative and commutative, inserts each key-value pair to the reduction object at the end of each *map* operation. Thus, every key-value pair is merged to the *output results* immediately after it is generated. A data structure storing these intermediate values of output is referred to as the *reduction object*. Every time a key-value pair arrives, the reduction object locates the index corresponding to the key, and reduces this key-value pair to that index in the reduction object, exploiting the associative and commutative property. The reduction object is transparent to the users, who write the same *map* and *reduce* functions as in the original specification.

We use k-means clustering as a running example to illustrate our concepts. K-means clustering is one of most popular data mining algorithms [14]. The clustering problem is as follows. We consider data instances as representing points in a high-dimensional space. Proximity within this space is used as the criterion for classifying the points into clusters. Four steps in the sequential version of k-means clustering algorithm are as follows: 1) start with k given centers for clusters; 2) scan the data instances, for each data instance (point), find the center closest to it and assign this point to the corresponding cluster; 3) determine the k centroids from the points assigned to the corresponding centers, and 4) repeat this process until the assignment of points to clusters does not change.

Algorithm 1: map($input, offset$)

```
$point \leftarrow$ get_point($input, offset$);
for $i \leftarrow 0$ to $K$ do
    $dis \leftarrow$ distance ($point, clusters[i]$);
    if $dis < min$ then
        $min \leftarrow dis$;
        $min\_idx \leftarrow i$;
$Kmeans\_value\ value$;
$value.num\_points \leftarrow 1$;
$value.dim0 \leftarrow point[0]$;
$value.dim1 \leftarrow point[1]$;
$value.dim2 \leftarrow point[2]$;
$value.dist \leftarrow min$;
$reduction\_object$.insert(&$min\_idx$, sizeof($min\_idx$), &$value$,
sizeof($value$));
```

Algorithm 2: reduce($value1, value1\_size, value2, value2\_size$)

```
$km\_value1 \leftarrow *(Kmeans\_value*)value1$;
$km\_value2 \leftarrow *(Kmeans\_value*)value2$;
$Kmeans\_value\ tmp$;
$tmp.num\_points \leftarrow$
$km\_value1.num\_points + km\_value2.num\_points$;
$tmp.dim0 \leftarrow km\_value1.dim0 + km\_value2.dim0$;
$tmp.dim1 \leftarrow km\_value1.dim1 + km\_value2.dim1$;
$tmp.dim2 \leftarrow km\_value1.dim2 + km\_value2.dim2$;
$tmp.dist \leftarrow km\_value1.dist + km\_value2.dist$;
copy($value1$, &$tmp$, sizeof($tmp$));
```

Algorithm 1 and Algorithm 2 show the code of *map* and *reduce* functions in k-means. The *map* function processes one input point each time. It uses the identifier of the closest cluster center as the key, and the coordinates of this point, together with the number of points (1) and the distance to the cluster center are combined together to form a key-value pair. As part of runtime processing, this key-value pair is *inserted* in the *reduction object*. A *reduction object* is a data structure that maintains the current value of the final output results from the *reduce* stage of the application. For k-means clustering, the output of the *reduce* stage is the number of points associated with each cluster center, together with the total aggregated coordinates and distance of these points from the cluster center. At any point in the processing of the application, the reduction object contains the same information, but only for the set of points that have been processed so far.

The *insert* function works as follows. If the *key* already exists in the reduction object, the *reduce* function is invoked to merge the new key-value pair to the existing key. If it is the first time that this key is inserted, a new space for this key-value pair is created. Thus, elements within the reduction object are dynamically allocated. The *reduce* function in this example computes the new value for the existing key, i.e., it accumulates the number of points, the coordinates, and the distance of the new value to the existing value associated with that key. We will discuss the *insert* function in details in Section 3.4.

As we stated earlier, the main advantage of this approach is that the memory overhead of a large number of key-value pairs is avoided. This, in turn, is likely to allow us to utilize the shared memory on a GPU. However, we still need to address several chal-

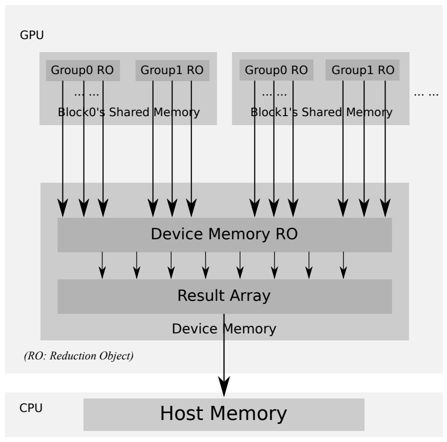

Figure 1: Memory Hierarchy

lenges to have an efficient implementation for MapReduce processing on a GPU.

To complete our design, we have the following features, which will be explained in the rest of this section.

- A memory hierarchy for maintaining the reduction object.
- A *multi-group* scheme in shared memory to trade off space requirements and locking overheads.
- A general and efficient data structure for the reduction object.
- An efficient overflow handling and swapping mechanism.

3.2 Memory Hierarchy

The processing structure we support reduces the memory requirements, by replacing the need for managing key-value pairs by a reduction object. For many applications, the reduction object tends to be small and can be kept in shared memory throughout the computation. However, for some other applications, it is possible that the number of distinct keys is very large, and shared memory may not be sufficient to hold the reduction object. To effectively use shared memory and yet have a general implementation, a memory hierarchy is maintained in our framework. As shown in Figure 1, reduction objects exist in both shared memory and device memory.

Elements of the reduction object are kept in shared memory as long as there is space. If an element corresponding to a key value that did not exist in the reduction is to be added, and there is no space, our framework swaps the data out to the device memory reduction object. The swapping mechanism will be discussed in details in Section 3.5.

There is only one copy of the reduction object in the device memory, which is used to collect the reduction result from shared memory reduction objects. The number of shared memory reduction objects depends upon the number of thread blocks and a design decision related to the number of *thread groups* that are created within each block. Each thread block is divided into one or more groups, and each group has one copy of the reduction object. Further details of the reasons for creating these thread groups will be discussed in the next subsection.

After all the threads in one thread block finish processing, they merge the data from the shared memory reduction object(s) into the device memory reduction object. To improve efficiency, this step is also performed in parallel. Merging threads read data as key-value pairs from shared memory reduction object(s), and insert them into the device memory reduction object. After all threads on GPU finish processing, they cooperate to copy data from the device memory reduction object to the result array, which is finally copied to host memory at last.

3.3 Multi-group Scheme in Shared Memory

A large number of threads (typically up to 512) are used in each thread block to exploit available parallelism in a streaming multiprocessor. Suppose there is one copy of the reduction object for all threads in the block. If all threads try to update the object, race conditions can arise. To avoid race conditions, we use synchronization mechanisms, such as the atomic operations that are supported on GPUs. However, when the number of distinct keys is small, the contention between the threads for updating the values associated with one key can be large, leading to significant waiting times.

One method for avoiding these costs could be to use *full replication*, i.e., create one copy of the reduction object for each thread. However, this method is not feasible because the number of threads is large, and if every thread owns one copy of the reduction object, the memory space needed will be large, leading to poor utilization of shared memory. In addition, such a large number of copies of the shared memory reduction object can lead to a very high overhead when they are merged to the device memory reduction object.

In order to address this problem, we introduce a *multi-group scheme*. Here, each thread block is partitioned into multiple groups, and each group owns its own copy of the reduction object. Let the number of threads in one block be N, and the number of groups in one block be M. Then the number of threads in one group is N/M and these threads share one copy of the reduction object. Locking operations are still required to update the copy of the reduction object, but the contention among the threads is reduced, since the number of threads competing for the same copy of the reduction object is now smaller. At the same time, the number of copies of the reduction object is still reasonable, which keeps the memory requirements and the overhead of combination modest.

The number of groups of threads that are used is a parameter in the system, and its optimal value can depend upon the application. Up to a certain level, increasing the number of thread groups improves performance, since the contention among threads is reduced. However, after a certain point, the performance can be reduced. One reason is that the size of shared memory is limited, so if the total number of keys is large, then the shared memory may not be able to hold many copies of the reduction object. This, in turn, increases the frequency with which swap mechanism needs to be invoked. Also, when the computation finishes, reduction objects in shared memory need to be merged into the device memory reduction object, and the data copying and synchronization to device memory can be time consuming. Clearly, the optimal choice can depend upon the size of one copy of the reduction object. When the reduction object is small and/or involves a smaller number of distinct keys, larger number of groups (or a smaller number of threads in each group) will be preferable. Similarly, when the reduction object is large and/or involves a large number of distinct keys, a smaller number of copies of the reduction object should be used.

An important parameter in our approach is the number of groups, g. The optimal choice for this parameter can be determined as follows. The overall execution time T for a specific application can be represented as:

$$T = C + \frac{T_{con}}{g} + T_{com} \times g \quad (1)$$

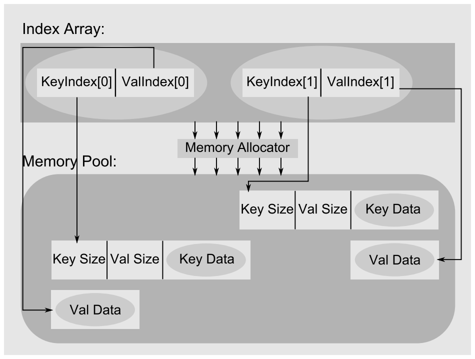

Figure 2: Reduction Object

where, C is the total time spent on initialization and computation, which is a constant for a given application and dataset. T_{con} is the contention overhead if only one group is used, and T_{com} is the combination overhead for one reduction object.

The minimum value of T is achieved when g equals $\sqrt{\frac{T_{con}}{T_{com}}}$, and the resulting optimal value of T is $C + 2\sqrt{T_{con} \times T_{com}}$.

3.4 Reduction Object Implementation

The structure of the reduction object is shown in Figure 2. It consists of two main parts: an *index array* and a *memory pool*. The index array is implemented as a hash table. Each bucket in the index array is composed of two indices: a *key index* and a *value index*. The *key index* refers to the position where the *key* data is stored in the memory pool. The *value index* is the position where the *value* data is stored.

Every bucket corresponds to a unique key, and the bucket is initialized when the first key-value pair with that particular key is inserted. The key data area contains not only the key data, but also the size information of key and value. In order to save space, only two bytes are used for storing the size information: one byte for the key size and the other for the value size. The memory pool is also an array, which is allocated before the computation starts. A memory allocator is used to allocate space for new keys and values.

Besides the index array and memory pool, one lock array is used in every copy of the reduction object. For each bucket in the index array, one lock is used to synchronize the updates to the key and value associated with that bucket.

Updating the Reduction Object: Algorithm 3 shows the logic of updating a reduction object. When a key-value pair is to be inserted into the reduction object, the hash value of the key is first calculated by the hash function, and then an index is calculated by using the hash value. The framework uses this index to find the bucket corresponding to the key.

The update involves the following two steps:

1. The thread which is performing the computation acquires the corresponding lock on the bucket. If the bucket is empty, new space for the key data and the value data is allocated by the memory allocator. After that, the size information and the key data is stored into the newly allocated key data space, and the value data is stored into the value data space. Finally, the index information of the key and the value is stored in the bucket. The computing thread releases the lock.

2. If the bucket already contains data, the key size and key address associated with this bucket are first retrieved, and then used to compare with the new key. If the two keys are the same, the user-defined *reduce* operation is performed, and the value associated with the key is updated. After the *reduce* operation, the computing thread releases the lock on the current bucket. If the keys are not the same, the computing thread releases the lock and calculates a new index, which is then used to repeat the steps starting from the Step 1.

Algorithm 3: insert($key, key\_size, val, val\_size$)

```
$index$ ←hash($key$, $key\_size$)%NUM_BUCKETS;
while $finish \neq 1$ do
    $DoWork \leftarrow 1$;
    /*wait on the lock*/
    while $DoWork$ do
        if get_lock($index$) then
            if $buckets[index] = 0$ then
                $k$ ←alloc($2 + key\_size$);
                $v$ ←alloc($val\_size$);
                /*store size information*/
                $key\_size\_addr$ ←get_addr ($k$);
                $val\_size\_addr \leftarrow key\_size\_addr + 1$;
                $*key\_size\_addr \leftarrow key\_size$;
                $*val\_size\_addr \leftarrow val\_size$;
                /*store key and value data*/
                $key\_data\_addr \leftarrow key\_size\_addr + 2$;
                $val\_data\_addr$ ← get_addr($v$);
                copy($key\_data\_addr, key, key\_size$);
                copy($val\_data\_addr, val, val\_size$);
                /*store key and value indices*/
                $buckets[index][0] \leftarrow k$;
                $buckets[index][1] \leftarrow v$;
                release_lock($index$);
                $DoWork \leftarrow 0$;
                $finish \leftarrow 1$;
            else
                $key\_r$ ← get_key_addr($index$);
                $key\_size\_r$ ← get_key_size($index$);
                if equal($key\_r, key\_size\_r, key, key\_size$) then
                    $val\_r$ ←get_val_addr($index$);
                    $val\_size\_r$ ←get_val_size($index$);
                    reduce($val\_r, val\_size\_r, val, val\_size$);
                    release_lock($index$);
                    $DoWork \leftarrow 0$;
                    $finish \leftarrow 1$;
                else
                    release_lock($index$);
                    $DoWork \leftarrow 0$;
                    $index$ ←new_index($index$);
```

Memory Allocation in Reduction Object: Beginning with CUDA 3.2, dynamic device memory allocation has been supported on NVIDIA GPUs. However, the default mechanism tends to be expensive. Hong *et al.* developed a light-weight dynamic device memory allocator as part of the MapCG implementation [13]. Our method for memory allocation works in a similar way as MapCG, but has its own characteristics.

In MapCG, a large memory pool in the device memory is reserved before the computation starts. Every time additional space is needed, the light-weight memory allocator allocates a memory space and returns the memory index. We associate a memory allocator with both the device memory reduction object and the shared memory reduction object. These two memory allocators are somewhat different from each other. In every shared memory reduction object, only one offset is used to indicate the next available memory space. However, in the device memory reduction object, multiple offsets are used. We divide the threads in each block into many groups, which is similar to the *multi-group* scheme in the

Algorithm 4: Overflow Handling and Swapping Mechanism

```
/*variables shared in block*/
$shared\_objects[NUM\_GROUPS]$;
$full$;
$finish$;
/*flag used to guarantee that only update $finish$ once*/
$first \leftarrow 1$;
/*$gid$ is group id*/
$shared\_objects[gid]$.init();
if $tid = 0$ then
    $full \leftarrow 0$;
    $finish \leftarrow 0$;
barrier()(all threads in one block);
$i \leftarrow blockIdx.x * blockDim.x + threadIdx.x$;
/*end the loop if all threads in a block finish computation*/
while $finish \neq blockDim.x$ do
    barrier();
    while $i < task\_number$ do
        if $full$ then
            break;
        $success \leftarrow$map($i$, &$shared\_objects[gid]$);
        if $success \neq 1$ then
            $full \leftarrow 1$;
            break;
        $i \leftarrow i + gridDim.x * blockDim.x$;
    /*current thread finishes computation*/
    if $first \&\& i \geq task\_number$ then
        atomicAdd($finish, 1$);
        $first \leftarrow 0$;
    barrier();
    /*merge to device memory reduction object*/
    merge($device\_object$, &$shared\_objects[gid]$);
    barrier();
    /*reinitialize the shared memory reduction objects*/
    $shared\_objects[gid]$.init();
    if $tid = 0$ then
        $full \leftarrow 0$;
```

shared memory. The reason why we use multiple offsets for the device memory reduction object is that all the threads across all the blocks can be updating the same device memory reduction object. To avoid race conditions, the memory offset is updated by using atomic operations. However, the contention of the lock can be high with a single copy, which is why separate offsets are used. Also, we store the offsets in the shared memory, which further lowers the overhead of memory allocation.

3.5 Handling Shared Memory Overflow

For many MapReduce applications, it is possible that during the process of computation, the shared memory reduction object gets full. More specifically, there can be two situations in which a reduction object is full. The first is when the buckets in the index array are all used up, and the second situation is when the memory pool does not have any more available space to allocate.

In such cases, we need to perform *overflow handling*. Our framework supports two ways of overflow handling. The first way is to swap the data out to device memory, and the second is to sort the reduction object and delete unnecessary data. For most applications, the swapping mechanism is used when the reduction object is full. Data from the shared memory reduction object is retrieved and inserted into the device memory reduction object. After such a swap is completed, the shared memory reduction object in one block is re-initialized, which is necessary for the remaining computations.

The reduction object also supports *in-object parallel sorting*. For applications like kNN (k-nearest neighbor search), when the reduction object in shared memory is full, bitonic sort [10] is performed on the reduction object. In such application, newly inserted data can make earlier data redundant. After sorting, only a small part of the data is kept in the reduction object, and the unnecessary data is deleted, which makes room for the remaining computation.

Algorithm 4 shows the logic of the swapping mechanism in overflow handling. In-object sorting logic is similar and is not shown here. CUDA provides a synchronization barrier routine for the threads within each block. A full flag shared among one block is used to indicate whether overflow handling is needed. Every time before a thread performs the *map* operation, it checks this flag. If this flag is not set, it carries out the regular update. Whenever a thread encounters an insertion failure, it sets this flag. All other threads within this block will see that the flag has been set, and know that they need to cooperate to conduct overflow handling. All threads in the block stop computations and merge the shared memory reduction object into the device memory reduction object. This operation is performed in parallel by all threads. If multiple thread groups, i.e., multiple copies of the reduction object are being used, the following logic is used. Irrespective of which group's reduction object is full, all threads from all groups within the block need to suspend all computations and perform overflow handling. Every thread only participates in the work associated with its own group's reduction object. We require all groups to perform overflow handling even when only one group's reduction object is full because the synchronization barrier, *__syncthreads()*, is used for all the threads in one block. There is no synchronization mechanism for a subset of threads in each block, except for the automatic synchronization within a warp. Although other groups' reduction objects may not yet be completely full, the rate of space utilization within the reduction objects is almost the same, because the load has been evenly distributed.

3.6 Application Examples

In this section, we use two representative examples to show how our framework supports different parallel applications. These two applications are k-means clustering (k-means) and k-nearest neighbor search (kNN). We use k-means to demonstrate the use of swapping mechanism, and use kNN to show how the in-object sorting mechanism helps.

K-means Clustering: The steps in the k-means clustering algorithm, along with a MapReduce implementation, were described earlier in Section 3.1. Now we show how our framework implements this algorithm.

As the implementation of the *map* function shows, we use the cluster center identifier as the key, and a structure containing the coordinates, the number of points and the distance from the cluster center to the current point as the value. Thus, at most k distinct keys are generated.

Immediately after a key-value pair is generated, it is inserted into the reduction object. As described in Section 3.4, when a key-value pair is inserted, the *reduce* function is used to update the newly generated value to the existing value corresponding to the same key. If k is small, for example, when k is 10 or 20, then the number of distinct keys is also small. We can use more groups in each block to reduce the contention among threads. If k is larger, such as 50 or 100, then we should use fewer groups to ensure that the shared memory can hold all the copies of the reduction object. For even larger k, we can choose the number of groups to be 1, though even in this case, shared memory may overflow and the swapping mechanism will be required.

K-nearest Neighbor Search: The k-nearest neighbor search (kNN) is a type of instance-based learning method [1]. This method is used to classify an object in an n-dimensional space based on the closest training samples. Given n training samples and one unknown sample, the k-nearest neighbor classifier searches the training samples by euclidean distance and finds the k nearest points to the unknown sample. Then it classifies the unknown sample according to the majority vote of the k nearest samples.

When implementing this algorithm by using our framework, we can take advantage of the in-object sorting mechanism we have supported, though it requires an extra function to be provided by the users. For every training sample, we use its point identifier as the key, and the distance from the unknown sample as the value. The framework keeps generating key-value pairs. When the reduction object corresponding to one group is full, the framework does not swap the reduction objects in the shared memory, but instead it conducts bitonic sort on the reduction objects. Users only need to define a single additional function, which will be comparing two buckets. This function is invoked by the sorting algorithm. After sorting, the framework deletes unnecessary data and only keeps k points that are the closest to the unknown sample. These points are kept in the shared memory reduction object, whereas the other space is made available for the remaining computations. When all threads finish processing data points and have combined the shared memory reduction objects to the device memory reduction object, an in-object sort on the device memory reduction object is performed.

3.7 Discussion

As we mentioned before, our framework is very suitable for *reduction-intensive* applications. These applications generate a large number of key-value pairs for any given key value, and the key-value pairs are reduced in an associative and commutative way. For these applications, we can reduce the memory overhead, and effectively utilize the shared memory.

Consider an application with no reduction, i.e. where only one key-value pair is generated for a given key, or the reduction operation is non-associative-and-commutative. Matrix Multiplication, if written using a MapReduce framework, has this property. In such cases, our method has no advantage over other GPU based MapReduce frameworks. In fact, it may be better not to try and use shared memory, since what is written in shared memory is never updated, and then has to be copied to device memory.

While using our framework, it is desirable to investigate the specific properties of the application. Our implementation is advantageous (over other GPU map-reduce implementations) only for applications with associative and commutative reduction operations, and within those, applications that are reduction-intensive. Also, certain parameters in our system, including the number of buckets, the memory pool size in each reduction object, as well as the number of groups in each thread block, currently need to be determined by the users. The users can make these decisions based on the potential number of key-value pairs that can be generated and the estimated memory requirements of the key-value pairs. This is similar to the more popular map-reduce implementations like Hadoop, where parameters like chunk size, replication factors, and others need to be explicitly chosen by users. The next step in our research will be developing techniques to automatically choose the values of some or all of these parameters.

The architecture of modern GPUs are changing rapidly. Over time, GPUs have seen an increase in shared memory size and device memory access speed. However, memory hierarchy still exists, and will continue to exist. Our method of utilizing shared memory for MapReduce on a GPU is, therefore, applicable to any of the existing or upcoming architectures. In fact, because larger shared memory can hold larger reduction objects, and thus reduce the frequency of swapping operations, our approach may be even more benefitial.

3.8 Extension for Multi-GPU Systems

Even though a single GPU is a highly parallel architecture, high throughput processing may require use of a multi-GPU node, and/or a cluster with GPU(s) on each node. We now argue why our reduction object based approach is suitable for multi-GPU environments.

For applications that involve associative and commutative reduction operations and that are reduction-intensive, the following approach can be used. First, the data to be processed can be evenly divided between the nodes or GPUs. Second, processing as described in this paper can be directly applied on each GPU. The result of this processing will be the reduction object. Next, the reduction objects from each node or GPU can be combined using the associative and commutative reduction function to obtain the final results. The approach here will have two advantages. First, the processing on every GPU will be faster than other approaches. Second, the communication overheads will be lower than other approaches that involve shuffling of key-value pairs. If the reduction object is small, the final global reduction step will be very inexpensive.

For applications that do not involve associative and commutative reduction operation or are not reduction-intensive, we can default to the approaches used by other multi-GPU implementations, for example, GPMR [21].

4. EXPERIMENTAL RESULTS

We have evaluated the performance of our framework with many different applications, covering various types of applications that can be implemented using the MapReduce framework. We compare the performance of the applications against sequential (1 CPU thread) implementations, which were chosen or implemented to have the same logic/algorithm as the corressponding MapReduce implementations. We also compare the performance against an earlier MapReduce framework for GPU, MapCG [13], which does not use shared memory intensively. Finally, we also compare the performance of our system against Ji *et al.*'s work [15], a recent implementation of MapReduce that does use shared memory.

We have also conducted a number of experiments to evaluate some of the design decisions we made. For the reduction-intensive applications we have used, we consider different configurations of thread groups in each block and test the impact of the *multi-group* scheme. To investigate the performance of the swapping mechanism, we evaluate two applications with specific datasets.

4.1 Experimental Setup and Applications

Our experiments were conducted using an NVIDIA Quadro FX 5800 GPU with 30 multiprocessors. Each multiprocessor has 8 processor cores working at a clock rate of 1.3 GHz. Thus the GPU we used has 240 processor cores in total. The total device memory size is 4 GB and the shared memory size on each multiprocessor is 16 KB. The GPU is connected to a machine with 8 AMD 2.6 GHz dual-core Opteron CPUs. The total main memory on this machine is 24 GB.

We selected seven commonly used MapReduce applications to evaluate the performance of our framework. These examples cover both *reduction-intensive* and *map computation-intensive applications*. As we stated earlier, the former represents applications which have a large number of key-value pairs for each unique key, and the reduction operation is associative and commutative, and thus, reduction computation time is significant. In contrast, the latter represents applications that spend almost all their time in map stage, i.e., they have one or a very small number of key-value pairs for each unique key.

The applications we selected were as follows. K-means clustering (k-means) is one of the most popular data mining algorithms, which has been described previously. An important parameter here is the number of cluster (centers), k, which impacts memory requirements, and thus the *multi-group* scheme. K-nearest neighbor classifier (kNN) is a simple but popular machine learning algorithm, which was also described earlier.

Word Count (WC) is a very commonly used application for evaluating MapReduce frameworks. It calculates the total number of occurrences for each distinct word. In the key-value pairs generated by this application, the key is the character sequence of a word, and the value is the integer 1. WC can be reduction-intensive if the number of distinct words is small, and then the reduction objects can be kept in shared memory until the end of the computation. If the number of distinct words is very large, the reduction object cannot be held in shared memory, and then the swapping mechanism is needed.

Naive Bayes Classifier (NBC) is a simple classification algorithm based on observed probabilities. Given two sets of observations, one with classifications, one without, the algorithm classifies the second set. NBC selects the most likely classification V_{nb} given the attribute values $a_1, a_2, \ldots a_n$. V_{nb} is computed as

$$V_{nb} = argmax_{v_j \in V} P(v_j) \prod P(a_i|v_j) \qquad (2)$$

The particular invocation of this application in our experiments is given a set of data which includes the attributes color, type, origin, transmission type, and whether or not stolen. It judges the age of this vehicle based on the given information. When implemented using MapReduce, the total number of distinct keys is 30. The value generated by each *map* function is 1.

Our next application, Page View Count (PVC), obtains the number of distinct page views from web logs. Each entry in the web log is represented as ⟨*URL, IP*⟩, where URL is the URL of the accessed page and IP is the IP address that accesses the page. This application involves two passes in MapReduce. The first one removes the duplicate entries in the web logs, and the second one counts the number of views for each URL. In the first pass, the pair of the entry is used as the key and the size of the entry is used as the value. The second pass uses the URL as the key, and integer 1 as the value. Typically, the datasets have a lot of repeated entries, so the application is reduction-intensive in nature.

Another application we used is Matrix Multiplication (MM), which is a map computation-intensive application. Given two input matrices A and B, the *map* stage computes multiplication for a row i from A and a column j from B. It outputs the pair $\langle \{i, j\} \rangle$ as the key and the corresponding result as the value. If A is an $m_a \times n_a$ matrix, and B is an $m_b \times n_b$ matrix, then the total number of distinct keys is $m_a \times n_b$, which is a very large number if the dimensions of the matrices are large. Thus for this application, we use the device memory reduction object only. The *reduce* stage is not needed in this application, so there is no advantage to using shared memory reduction objects.

Our last application is Principle Component Analysis (PCA), which is also a map computation-intensive application. It aims to find a new (smaller) set of dimensions (attributes) that better captures the variability of data. Because it includes several steps that are not compute-intensive, we only perform the calculation of the correlation matrix on the GPU. In this step, given an $n \times m$ data matrix D with n rows and m columns, we compute the matrix S, which is an $m \times m$ *covariance matrix* of D. The reduction object for this application stores the elements of the covariance matrix. This application also does not have the *reduce* phase, and it uses device memory only.

Application	*Dataset Size (Small, Medium, Large)*
K-means, K = 20 (KM20)	10K points, 100K points, 1M points
K-means, K = 40 (KM40)	10K points, 100K points, 1M points
K-means, K = 100 (KM100)	10K points, 100K points, 1M points
KNN, K = 20 (KNN)	10K points, 100K points, 1M points
Word Count, 90 Distinct Words (WC)	5.46MB, 10.94MB, 87.5MB
Naive Bayes Classifier (NBC)	26MB, 52MB, 104MB
Page View Count (PVC)	44MB, 88MB, 176MB
Matrix Multiplication (MM)	512*512, 1024*1024, 2048*2048
Principle Component Analysis (PCA)	1048576*8, 1048576*16, 1048576*32

Table 1: Applications and Datasets Used in our Experiments

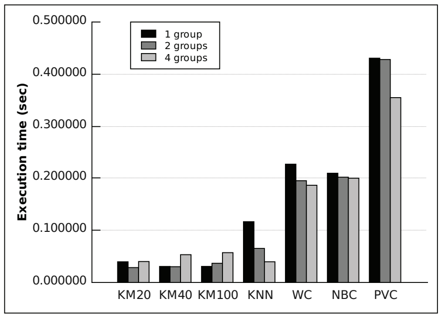

Figure 3: Performance Under Different Group Configurations

For every application example in our experiment, we use datasets of different sizes (small, medium, and large). The information about these datasets is shown in Table 1.

4.2 Evaluation of the Multi-group Scheme

To understand the behavior of our *multi-group* scheme, We executed the five reduction-intensive applications from our set of applications, varying the group numbers. Three different numbers of groups for each block were considered: 1, 2, and 4. Among our set of seven applications, MM and PCA are not included in these experiments, because they do not require reductions, and do not involve a shared memory reduction. kNN is not strictly a reduction-intensive application, but uses an in-object sort, and thus, the *multi-group* scheme is suitable for it. To focus on the impact of the number of groups, the results are reported from executions where the shared memory reduction objects are small and no overflow occurs during the entire computation. Finally, for k-means clustering, three different versions, KM20, KM40, and KM100 were created, by varying the number of clusters (centers), k. Recall that the size of the reduction object increases with increasing value of k.

Figure 3 shows the comparison of performance using different number of groups. These applications were tested with the large datasets. For KM20, the highest performance is achieved when the number of groups is 2. The total number of distinct keys in KM20 is 20, and thus the size of reduction object is very small. This implies a low combination overhead at the end of the computation. However, if the number of groups is 1, all the threads in one thread block update the same copy of the reduction object, and the contention can be very high. When the number of groups increases, the combination overhead increases, but the contention for each reduction object decreases. When we change the group number from 2 to 4, the increase of combination overhead is more significant as compared to the decrease in the contention overhead, so that the execution time increases.

KM40 has a larger reduction object. The combination overhead is now higher, but the contention during the updates to the reduction object is less severe. When we change the number of groups from 1 to 2, the benefit gained from the lower contention is almost the same with the performance loss due to the combination overhead, thus the execution time stays unchanged. When we increase the number of groups to 4, the performance gets worse due to the high combination overhead. The performance of KM100 follows a similar pattern with KM40 since it also has a large reduction object. The best performance is achieved with the group number of 1.

As we have discussed earlier, kNN is implemented by using the in-object sorting mechanism. Our experiments use 20 as the value of k. The reduction objects in this application fill up very frequently and the threads in each block need to perform overflow handling frequently, which, in turn, requires a bitonic sort. The bitonic sort incurs a large amount of data movement and is very time consuming. The combination overhead is relatively small compared with the sorting overhead. It turns out that in our implementation, as the number of groups increases, more threads take part in sorting. Thus the performance keeps improving when we increase the number of groups.

In the case of WC, the dataset used contains 90 distinct words, which makes it a reduction-intensive application. The best performance is achieved with the number of groups being 4. Although WC has a larger number of distinct keys than KM20, the contention on its reduction objects is more severe. This is because the time used to compare the newly generated keys with the keys in the reduction objects is long, and the hashing process and locating the key in the reduction object also take a substantial time. Thus, when the number of groups is increased, the contention decrease outweighs the combination overhead increase.

Although NBC has a small number of distinct keys (30), its execution time remains relatively unchanged with an increase in the number of groups. There are two reasons for this. On one hand, this application spends a relatively long time on processing each entry and generating key-value pairs, which makes the relative frequency of updating the reduction objects low, and the contention on the reduction objects also low. On the other hand, the hash function is able to locate the right buckets directly for this application, which makes the insertion to each bucket very fast.

PVC performs the best when the number of groups is 4. The PVC dataset we used involves 103 distinct entries, and thus, the total number of distinct keys in the first pass (which takes almost all the execution time) is also 103. The *map* function only computes the hash value of each entry and then emits it to the reduction object. Thus, this application has less computation, and the relative frequency of updates to each reduction object is very high. In addition, in this application, the hashing process and locating the key in the reduction object also take a relatively long time. Thus the contention on the reduction objects is quite high. Increasing the number of groups reduces such contention, resulting in better performance.

4.3 Comparison with Sequential Implementations

To show the overall benefits from our GPU-based framework, we compare the performance achieved on the GPU against the performance on a single thread application executing on 1 core of a CPU.

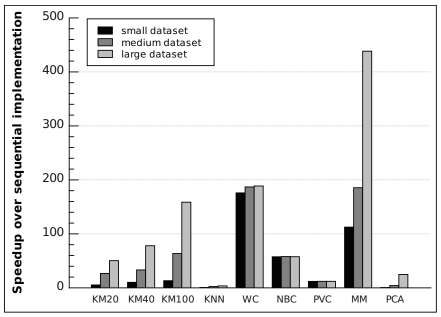

Figure 4: Speedup over Sequential Implementations

The results for the seven applications, and using three datasets for each application, are shown in Figure 4. The speedups vary considerably, depending upon the nature of the computation and the ratio between data transfer costs and execution times.

MM achieves the best speedup among all the applications. The reason is that MM does not involve any reduction. Moreover, the ratio between computation and amount of data to be processed is very high ($O(N^3)$ computation on $O(N^2)$ data). Thus, the GPU execution is not bound by data transfer costs or synchronization overheads.

The three versions of k-means, KM20, KM40, and KM100, along with WC, NBC, and PVC are reduction-intensive applications. Our framework is effective in speeding up these applications, especially for large datasets, but not to the extent of MM. This is because of a less favorable ratio between computation and data transfers, and because the reduction stage takes a considerable amount of time. Still, the speedup of KM100 on the large dataset is nearly 180, the speedups of WC on all datasets are around 200, and the speedups of NBC on all datasets are close to 50. In the case of k-means, the speedups increase with increasing value of k, because the amount of computation performed on each record also increases.

kNN only achieves a small speedup of 3.6x with the large dataset. The reason is the negligible amount of computation in the map phase and the high data movement cost. PCA also has a relatively small speedup, though it does become 24x on the large dataset. The reason is that the total number of data units to be processed is small, and thus the degree of parallelism is low.

All the applications achieve higher performance as the datasets become larger. The reason is that when the datasets are small, the initialization and synchronization overheads are dominant compared to the processing time.

4.4 Comparison with MapCG

We now compare the performance of our framework against one of recent implementations of MapReduce on GPUs [13]. This framework had been shown to outperform one of the earlier implementations of MapReduce for GPUs, Mars [12].

MapCG does not take advantage of shared memory intensively. Instead, it stores a lot of intermediate key-value pairs to the hash table in the device memory. This not only leads to significant overheads of device memory access and synchronization, but also high costs of execution of the reduction phase. So, we will expect that its performance will not be good for reduction-intensive applications, where our framework can exploit shared memory very effectively.

Figure 5 shows the speedup of our implementations over the

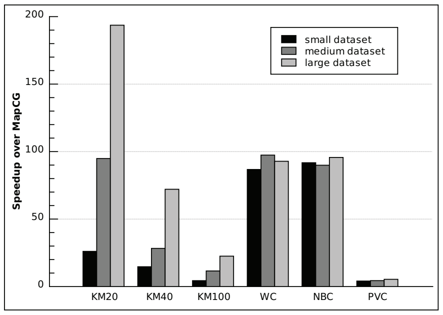

Figure 5: Speedup over MapCG for Reduction-intensive Applications

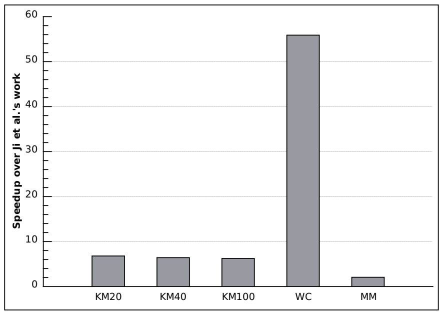

Figure 7: Speedup over Ji *et al.*'s Work

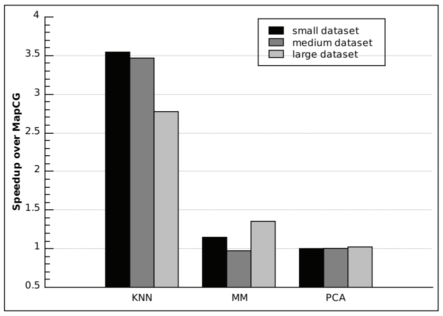

Figure 6: Speedup over MapCG for Other Applications

MapCG framework for four applications that we consider as the reduction-intensive applications. Each application is tested with small, medium, and large datasets, and three different values of k for k-means are used. We can see that we achieve significant speedups over MapCG for all datasets, and all applications in this set. KM20 achieves the highest speedup over MapCG. This is because it has a very small number of distinct keys, which leads to very high synchronization overheads (device memory based atomic operations) in the hash table used by MapCG. As the number of distinct keys becomes larger, MapCG performs somewhat better, though the relative speedups for large datasets are still very high. Similarly, WC and NBC both achieve very high relative speedups (about 90x on all datasets) due to the significant synchronization overheads in the map phase of MapCG's implementations. PVC has a speedup of 5x, which is smaller compared with the other 5 applications. One reason is that PVC has a large number of distinct keys, which makes the synchronization overhead with MapCG's hash table relatively low.

Figure 6 shows the speedup over MapCG for other three applications, which either spend a lot of time on map computations, or involve sorting. MM and PCA both do not use shared memory reduction objects. So the performance of our framework is almost the same as MapCG. For kNN, MapCG needs to sort the intermediate key-value pairs in device memory, which is time consuming. In comparison, our framework does most of the sorting in shared memory, although it also needs to sort the device memory reduction object before the computation finishes. Thus, our framework is more efficient, and as we can see from Figure 6, we have a factor of 3 speedup on the average over MapCG.

Overall, we can see that our method for exploiting shared memory leads to much better performance as compared with MapCG, for reduction-intensive applications. By comparing our speedups over MapCG and the speedups over sequential versions, we can see that for KM20 and NBC, MapCG even has a slowdown over sequential implementations, defeating the purpose in using a GPU. Thus, effective utilization of shared memory is crucial in benefiting from GPUs for MapReduce applications.

4.5 Comparison with Another Approach for Using Shared Memory

As we stated earlier in the paper, one recent implementation of MapReduce for GPUs does use shared memory [15]. This system, from Ji *et al.*, is based on a more traditional approach for executing MapReduce, i.e., it has a *shuffle* or *sort* phase between *map* and *reduce* phases. As described earlier, their system uses shared memory as a buffer to stage input and output.

We obtained the implementation from Ji *et al.* and have compared it against our implementation. Among the applications we have used, implementations of KM, WC, and MM were available as part of their distribution. Because their implementation is based on a somewhat different API, it was not possible to execute other four applications on their framework. Thus, we have used these three applications.

The results from KM20, KM40, KM100, WC, and MM with large datasets are shown in Figure 7. Our system is always faster, though the exact speedup varies. Our framework achieves a very high speedup (56x) for WC. The reason is the large proportion of time spent on shuffling the key-value pairs in Ji *et al.*'s implementation. Also, for all versions of k-means, our framework achieves a speedup of more than 6, as we get an effective use of shared memory and avoid shuffling overhead. MM only has the map phase, i.e., neither framework needs shuffling. We still achieve about a factor of 2 speedup for this application.

Overall, we can see that our approach for implementing MapReduce, including the way we use shared memory and avoid shuffling is significantly more efficient, and clearly outperforms the previous implementation.

4.6 Evaluation of the Swapping Mechanism

In some reduction-intensive applications, the number of distinct keys is large and the reduction objects in the shared memory can become full. The swapping mechanism implemented as part of our framework is invoked in such cases. Thus, while our framework can clearly correctly handle such applications, it is useful to explore the performance of our framework for these applications.

For this purpose, we experiment with WC and PVC, where the number of keys can vary (and can be quite large), depending upon

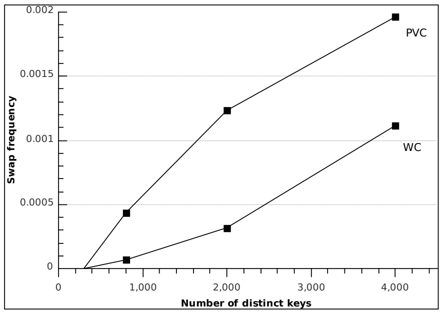

Figure 8: Swap Frequency under Different Number of Distinct Keys

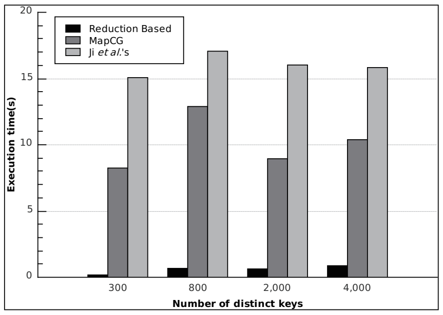

Figure 9: Comparison with Other Implementations for WC

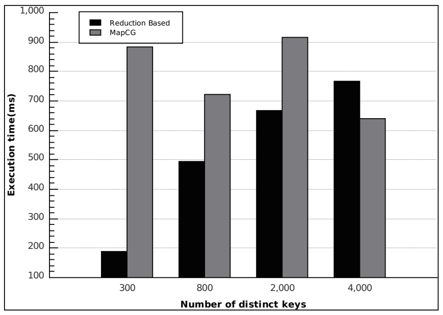

Figure 10: Comparison with MapCG for PVC

the dataset. Thus, we examine the performance of these two applications by increasing the number of distinct keys, which leads to the increase of swaps. For each application, we use 5 datasets to evaluate the swapping mechanism. The numbers of distinct keys in the datasets vary from 300 to 4000. The size of each dataset is 100 MB. The number of buckets in each shared memory reduction object is 600. Thus for datasets with 300 distinct keys, no swap is involved.

We first investigate the change of *swap frequency* with the number of distinct keys increasing. The swap frequency f is defined as:

$$f = \frac{N_{swap}}{N_{task}} \qquad (3)$$

where N_{swap} represents the total number of swaps in one execution, and N_{task} represents the total number of tasks in the workload. Larger values of f imply larger percentages of time spent on swapping.

The relationship between the number of distinct keys and the swap frequency is illustrated in Figure 8. We can see that as the number of distinct keys increases, the swap frequency increases, as we would expect. However, for both WC and PVC, the overall swap frequencies are quite low, although the swap frequency of PVC reaches 0.2% with the number of distinct keys being 4000. The results imply that in most cases, our system still performs much of the reduction work in shared memory.

We have also compared our system with MapCG and Ji *et al.*'s work when swaps are involved. Figure 9 shows the execution times of different frameworks for WC. Although our system involves swaps for number of distinct keys of 800, 2000, and 4000, it still achieves better performance than other two systems. MapCG has very high overhead because of the frequent accesses to global memory. Ji *et al.*'s framework spends considerable time shuffling. Figure 10 shows the comparison between our framework and MapCG for PVC (As was mentioned in Section 4.5, PVC is not included in Ji *et al.*'s distribution, and doesn't seem straight-forward to implement because of their specialized API). Our system performs better than MapCG for most cases, though MapCG has a shorter execution time as the distinct key number increases to 4000.

Thus, we can see from the results that even when the frequency of swaps increases, our framework still performs well. This is because that our system keeps a large proportion of the reduction object in shared memory, which results in lower device memory access frequency.

5. RELATED WORK

Recent years have seen a large number of efforts on MapReduce and its variants. We primarily focus on efforts specific to multi-core CPUs, GPUs, and heterogeneous systems.

Ranger *et al.* [19] have implemented a shared-memory MapReduce library named Phoenix in multi-core systems, and Yoo *et al.* [22] optimized Phoenix specifically for large-scale multi-core systems. CellMR [18] was a MapReduce framework implemented on asymmetric Cell multi-core processors. Streaming approach was adopted to support MapReduce. Mars [12] was the first attempt to harness GPU's power for MapReduce applications. MapCG [13] was a subsequent implementation which was shown to outperform Mars. We have extensively compared our work against MapCG. Catanzaro *et al.* [4] also built a framework around the MapReduce abstraction to support vector machine training as well as classification on GPUs. StreamMR [7] was a MapReduce framework implemented on AMD GPUs.

MITHRA [8] was introduced by Farivar *et al.* as an architecture to integrate the Hadoop MapReduce with the power of GPGPUs in the heterogeneous environments, specific for Monte-Carlo simulations. Shirahata *et al.* [20] have extended Hadoop on GPU-based heterogeneous clusters. They enabled *map* tasks to be scheduled onto both CPUs cores and GPUs devices. GPMR [21] was a recent project to leverage the power of GPU clusters for large-scale computing by modifying the MapReduce paradigm.

There has been a large volume of research on porting and tuning applications and developing programming systems on GPUs. Optimizing shared memory has been an important factor in many of these studies. Some of the prominent applications where shared memory use made a very substantial performance difference include N-Body [16] and FFT [11]. The methods used in these and other similar efforts were specific to the applications, and cannot be used as part of a high-level GPU programming system.

Aggressive compiler optimizations have also been used to exploit shared memory. Baskaran *et al.* have provided an approach for automatically arranging shared memory on NVIDIA GPUs by using the polyhedral model for affine loops [2]. They also focus on enabling more data reuse on shared memory, while reducing data movement. Moazeni *et al.* have adapted approaches for register allocation to manage shared memory on GPUs [17]. They used graph coloring to solve the problem.

6. CONCLUSIONS AND FUTURE WORK

This paper focuses on utilizing shared memory to accelerate MapReduce implementations on GPUs. In order to reduce the high overhead of data copying and synchronization on the device memory, we use a reduction-based approach, where we avoid storing the intermediate data (key-value pairs) in device memory. Instead, we perform reduction on the shared memory. We have also designed an approach for storing the reduction object on the memory hierarchy of the GPU. Finally, to further improve the performance, we have developed a *multi-group* scheme which allows us to balance the memory overhead and locking costs for the reduction objects.

We have evaluated the performance of our framework using seven applications. We have shown that our implementation delivers significantly higher performance than MapCG, a MapReduce framework which does not utilize shared memory intensively, and Ji *et al.*'s implementation, which takes a different approach for using shared memory.

Possibilities for future work include developing auto-tuning techniques to determine values of certain system parameters, which will make it easier for users to obtain high performance. We will also extend our framework to support new architectures, such as the emerging coupled CPU-GPU architectures from Intel, AMD, and NVIDIA.

7. REFERENCES

[1] David W. Aha, Dennis F. Kibler, and Marc K. Albert. Instance-based Learning Algorithms. *Machine Learning*, pages 6:37–66, 1991.

[2] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Automatic Data Movement and Computation Mapping for Multi-level Parallel Architectures with Explicitly Managed Memories. In *PPoPP '08*, pages 1–10, New York, NY, USA, 2008. ACM.

[3] Randal E. Bryant. Data-Intensive Supercomputing: The Case for DISC. Technical Report CMU-CS-07-128, School of Computer Science, Carnegie Mellon University, 2007.

[4] Bryan Catanzaro, Narayanan Sundaram, and Kurt Keutzer. A Map Reduce Framework for Programming Graphics Processors. In *Third Workshop on Software Tools for MultiCore Systems (STMCS)*, 2008.

[5] Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary R. Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-Reduce for Machine Learning on Multicore. In *NIPS*, pages 281–288, 2006.

[6] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, 2004.

[7] Marwa Elteir, Heshan Lin, and Wu-chun Feng. StreamMR: An Optimized MapReduce Framework for AMD GPUs. In *ICPADS '11*, Tainan, Taiwan, December 2011.

[8] Reza Farivar, Abhishek Verma, Ellick Chan, and Roy Campbell. MITHRA: Multiple Data Independent Tasks on a Heterogeneous Resource Architecture. In *CLUSTER*, pages 1–10. IEEE, 2009.

[9] Dan Gillick, Arlo Faria, and John Denero. MapReduce: Distributed Computing for Machine Learning. 2008.

[10] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. Gputerasort: High Performance Graphics Co-processor Sorting for Large Database Management. In *SIGMOD '06*, pages 325–336.

[11] Eladio Gutierrez, Sergio Romero, Maria A. Trenas, and Emilio L. Zapata. High Performance Computing for Computational Science - VECPAR 2008. pages 430–443. Springer-Verlag, 2008.

[12] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: A MapReduce Framework on Graphics Processors. In *PACT*, pages 260–269, 2008.

[13] Chuntao Hong, Dehao Chen, Wenguang Chen, Weimin Zheng, and Haibo Lin. MapCG: Writing Parallel Program Portable between CPU and GPU. In *PACT*, pages 217–226, 2010.

[14] Anil K. Jain and Richard C. Dubes. *Algorithms for Clustering Data*. Prentice-Hall, Inc., 1988.

[15] F. Ji and X. Ma. Using Shared Memory to Accelerate Mapreduce on Graphics Processing Units. In *IPDPS '11*, pages 805–816, 2011.

[16] Mark Harris Lars Nyland and Jan Prins. Chapter 31 Fast N-Body Simulation with CUDA, 2007.

[17] Maryam Moazeni, Alex Bui, and Majid Sarrafzadeh. A Memory Optimization Technique for Software-managed Scratchpad Memory in GPUs. *Application Specific Processors, Symposium on*, 0:43–49, 2009.

[18] M. Mustafa Rafique, Benjamin Rose, Ali Raza Butt, and Dimitrios S. Nikolopoulos. CellMR: A Framework for Supporting Mapreduce on Asymmetric Cell-Based Clusters. In *IPDPS*, pages 1–12, 2009.

[19] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary R. Bradski, and Christos Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *Proceedings of 13th HPCA*, pages 13–24, 2007.

[20] Koichi Shirahata, Hitoshi Sato, and Satoshi Matsuoka. Hybrid Map Task Scheduling for GPU-Based Heterogeneous Clusters. In *CloudCom '10*, pages 733–740, 2010.

[21] Jeff A. Stuart and John D. Owens. Multi-GPU MapReduce on GPU Clusters. In *IPDPS*, 2011.

[22] Richard M. Yoo, Anthony Romano, and Christos Kozyrakis. Phoenix Rebirth: Scalable MapReduce on a Large-Scale Shared-Memory System. In *IISWC*, pages 198–207, 2009.