2011 IEEE 17th International Conference on Parallel and Distributed Systems

StreamMR: An Optimized MapReduce Framework for AMD GPUs

Marwa Elteir*†, Heshan Lin†, Wu-chun Feng†, and Tom Scogland†
*City of Scientific Researches and Technology Applications, Egypt
†Department of Computer Science, Virginia Tech
Emails: {maelteir, hlin2, feng, njustn}@cs.vt.edu

***Abstract*—MapReduce is a programming model from Google that facilitates parallel processing on a cluster of thousands of commodity computers. The success of MapReduce in cluster environments has motivated several studies of implementing MapReduce on a graphics processing unit (GPU), but generally focusing on the NVIDIA GPU.**

Our investigation reveals that the design and mapping of the MapReduce framework needs to be revisited for AMD GPUs due to their notable architectural differences from NVIDIA GPUs. For instance, current state-of-the-art MapReduce implementations employ atomic operations to coordinate the execution of different threads. However, atomic operations can implicitly cause inefficient memory access, and in turn, severely impact performance. In this paper, we propose StreamMR, an OpenCL MapReduce framework optimized for AMD GPUs. With efficient atomic-free algorithms for output handling and intermediate result shuffling, StreamMR is superior to atomic-based MapReduce designs and can outperform existing atomic-free MapReduce implementations by nearly *five-fold* on an AMD Radeon HD 5870.

***Index Terms*—atomics, parallel computing, AMD GPU, GPGPU, MapReduce, Mars, MapCG, OpenCL**

I. Introduction

While graphics processing units (GPUs) were originally designed to accelerate data-parallel, graphics-based applications, the introduction of programming models such as CUDA [15], Brook+ [2], and OpenCL [10] has made general-purpose computing on the GPU (i.e., GPGPU) a reality. Although GPUs have the potential of delivering astounding raw performance via the above programming models, developing and optimizing programs on GPUs requires intimate knowledge of the architectural details, and thus, is nontrivial.

High-level programming models such as MapReduce play an essential role in hiding architectural details of parallel computing platforms from programmers. MapReduce, proposed by Google [7], seeks to simplify parallel programming on large-scale clusters of computers. With MapReduce, users only need to write a map function and a reduce function, and the parallel execution and fault tolerance is handled by the runtime framework. The success of MapReduce on cluster environments has also motivated studies of porting MapReduce on other parallel platforms including multicore systems [6], [16], Cell [12], and GPUs [16], [5], [9], [18].

However, existing MapReduce implementations on GPUs focus on NVIDIA GPUs. The design and optimization techniques in these implementations may not be applicable to AMD GPUs, which have a considerably different architecture than NVIDIA ones. For instance, state-of-the-art MapReduce implementations on NVIDIA GPUs [5], [9] rely on atomic operations to coordinate execution of different threads. But as the AMD OpenCL programming guide notes [3], including an atomic operation in a GPU kernel may cause all memory accesses to follow a much slower memory-access path, i.e., CompletePath, as opposed to the normal memory-access path, i.e., FastPath, even if the atomic operation is not executed.[1] Our results show that for certain applications, the atomic-based implementation of MapReduce can introduce severe performance degradation, e.g., a 28-fold slowdown.

Although Mars [4] is an atomic-free implementation of MapReduce on GPUs, it has several disadvantages. First, Mars incurs expensive preprocessing phases (i.e., redundant counting of output records and prefix summing) in order to coordinate result writing of different threads. Second, Mars sorts the keys to group intermediate results generated by the map function, which has been found inefficient [5].

In this paper, we propose StreamMR, an OpenCL MapReduce framework optimized for AMD GPUs. The design and mapping of StreamMR provides efficient atomic-free algorithms for coordinating output from different threads as well as storing and retrieving intermediate results via hash tables. StreamMR also includes efficient support of combiner functions, a feature widely used in cluster MapReduce implementations but not well explored in previous GPU MapReduce implementations. The performance results of three real-world applications show that StreamMR is superior to atomic-based MapReduce designs and can outperform an existing atomic-free MapReduce framework (i.e., Mars) by nearly *five-fold* on AMD GPUs.

II. Background

A. AMD GPU Architecture

An AMD GPU consists of multiple SIMD units named compute units, and each compute unit consists of several cores called stream cores. Each stream core is a VLIW processor containing five processing elements, with one of them capable of performing transcendental operations like sine, cosine, and logarithm. Each compute unit also contains a branch execution unit that handles branch instructions. All stream cores on a compute unit execute the same instruction sequence in a lock-step fashion.

There is a low-latency, on-chip memory region shared by all stream cores on a compute unit named LDS. Each LDS is connected to L1 cache as shown in Figure 1. Several compute units then share one L2 cache that is connected to the global memory through a memory controller. The global memory is a high-latency, off-chip memory that is shared by all compute units. Host CPU transfers the data to global memory through PCIe bus. In addition to the local and global memory, there are two special types of memories that are also shared by all compute units. Image memory is a high-bandwidth memory

[1]The details of both CompletePath and FastPath will be discussed in Section II.

1521-9097/11 $26.00 © 2011 IEEE
DOI 10.1109/ICPADS.2011.131

region whose reads are cached through L1 and L2 caches. Constant memory is a memory region storing data that are allocated/initialized by the host and not changed during the kernel execution. Access to constant memory is also cached.

B. Memory Paths

As shown in Figure 1, ATI Radeon HD 5000 series GPUs have two independent paths for memory access: FastPath and CompletePath [3]. The bandwidth of the FastPath is significantly higher than the CompletePath. Loads and stores of data whose size is multiple of 32 bits are executed through FastPath, whereas advanced operations like atomics and sub-32 bit data transfers are executed through the CompletePath.

Executing a memory load access through FastPath is performed by a single vertex fetch (vfetch) instruction. In contrast, a memory load through the CompletePath requires a multi-phase operation and thus can be several folds slower according to the AMD OpenCL programming guide [3]. On AMD GPUs, the selection of the memory path is done automatically by the compiler. The current OpenCL compiler maps all kernel data into a single unordered access view. Consequently, including a single atomic operation in a kernel may force all memory loads and stores to follow the CompletePath instead of the FastPath, which can in turn cause severe performance degradation of an application as discovered by our previous study [8]. Note that atomic operations on variables stored in the local memory does not impact the selection of memory path.

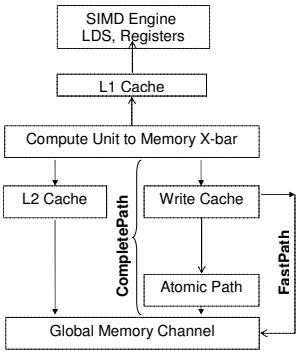

Fig. 1. AMD GPU memory hierarchy

C. GPGPU with AMD

We implemented our framework using OpenCL because currently OpenCL [10] is the main programming language on AMD GPUs. Another advantage of using OpenCL is its portability across GPUs from different vendors. In OpenCL terminology, each thread of a kernel is called a *workitem* and executed on a single stream core. Multiple workitems are organized into a *workgroup*. One or more workgroups can run concurrently in a compute unit. The resource scheduler executes each workgroup as several *wavefronts* (a wavefront is similar to the warp concept in CUDA). To hide memory latency, it switches between the wavefronts whenever any one is waiting for a memory transaction to complete.

D. MapReduce Programming Model

MapReduce is a high-level programming model aims at facilitating parallel programming by masking the details of the underling architecture. Programmers need only to write their applications as two functions: the *map* function and the *reduce* function. All of the input and outputs are represented as *key/value* pairs. Implementing a MapReduce framework involves implementing three phases: the *map* phase, the *group* phase, and the *reduce* phase. Specifically, the MapReduce framework first partitions the input dataset among the participating parties (e.g. threads). Each party then applies the map function to its assigned portion and writes the intermediate output (map phase). The framework groups all of the intermediate outputs by their keys (group phase). Finally, one or more keys of the grouped intermediate outputs are assigned to each partition party, which will carry out the reducing function and write out the result key/value pairs (reduce phase).

III. Related Work

Many research efforts have been done to enhance the MapReduce framework [11], [19], [17], [14], [13] in cluster environments. Valvag et al. developed a high-level declarative programming model and its underlying runtime, Oivos, which aims at handling applications that require running several MapReduce jobs [19]. Zahria et al. [14] on the other side proposed a speculative task scheduling named LATE (Longest Approximate Time to End) to cope with several limitations of the original Hadoop's scheduler in heterogeneous environments such as Amazon EC2[1]. Moreover, Elteir et al. [13] recently enhanced MapReduce framework to support asynchronous data processing. Instead of having barrier synchronization between map and reduce phases, they propose interleaving both phases, and start the reduce phase as soon as a specific number of map tasks are finished.

Mars [4] is the first MapReduce implementation on GPUs. One of the main challenges of implementing MapReduce on GPUs is to safely write the output to a global buffer without conflicting with output from other threads. Mars addresses this by calculating the exact write location of each thread. Specifically, it executes two preprocessing kernels before the map and reduce phases. The first kernel counts the size of the output from each map/reduce thread by executing the map/reduce function without writing the generated output to the global buffer. The second kernel is a prefix summing that determines the write location of each thread. Each thread then reapplies the map/reduce function and safely writes the intermediate/final output to the predetermined location in the global buffer. After the map phase, Mars groups the intermediate output by their keys using bitonic sort. After similar preprocessing kernels (counting and prefix summing), the reduce phase starts, where every thread reduces the values associated with certain key and finally writes the generated key/value pair to the final output. One main disadvantage of Mars' preprocessing design is that the map and reduce functions need to be executed twice. Such a design was arguably due to that atomic operations were not supported on the GPUs at the time Mars was developed.

Recently Hong et al. proposed MapCG [5], an implementation for MapReduce on both CPU and GPU. Its GPU implementation depends on using atomic operations to safely write the intermediate and final output. Also, MapCG designed a memory allocator to allocate buffers from the global memory for each warp. Moreover, MapCG uses hash

tables to group intermediate output from map function, which is shown to be more efficient than sorting used in Mars.

As we will discuss in Section IV, our investigation shows that using global atomic operations can cause severe performance degradation in MapReduce implementation on AMD GPUs. Consequently, StreamMR does not use global atomic operations. Instead, StreamMR introduces several novel techniques that address disadvantages of Mars, including an efficient output procedure that greatly reduces the preprocessing overhead as well as an atomic-free algorithm that groups intermediate results using hash tables. More details of StreamMR design will be discussed in Section V.

There are two other studies on accelerating MapReduce on GPUs [9], [18] that are orthogonal to our study in this paper. In [9], Ji et al. proposed several techniques to improve the input/output performance by using shared memory as a staging area. These techniques can be applied to StreamMR to further improve its performance. GPMR [18] is a MapReduce implementation for a cluster of GPUs. GPMR is mainly designed to minimize the communication cost between different GPUs. GPMR also introduces several application-specific optimizations to improve the program performance.

IV. Performance Impacts of Atomic Operations on MapReduce Frameworks

In this section, we seek to quantify the performance impacts of using atomic operations in MapReduce implementations on an AMD Radeon HD 5870 GPU. We first implement a basic OpenCL MapReduce framework based on Mars. In its original design, Mars uses preprocessing kernels, i.e., counting and prefix summing kernels, to calculate the locations of output records in global memory for each thread. We add a feature that allows threads in different wavefronts to use atomic operations (instead of using preprocessing kernels) to compute the output locations.

We compare the performance of the basic OpenCL MapReduce implementation (named Mars) and the atomic-based implementation (named AtomicMR), focusing on the execution time of two MapReduce applications: Matrix Multiplication (MM) and KMeans (KM). Specifically, we run MM for matrix sizes of 256 X 256, 512 X 512, and 1024 X 1024, and KM for number of points 4K, 16K, 64K. As shown in Figure 2 and Figure 3, the performance of atomic-based MapReduce framework is significantly worse than Mars. More specifically, the average slowdown is 28-fold and 11.3-fold for Matrix Multiplication and KMeans, respectively. These results suggest that atomic-based MapReduce implementations are not suitable for AMD/ATI Radeon HD 5000 series.

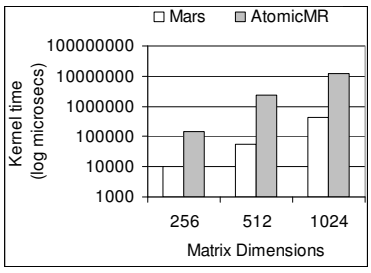

Fig. 2. Performance of atomic-based MapReduce vs. Mars using Matrix Multiplication

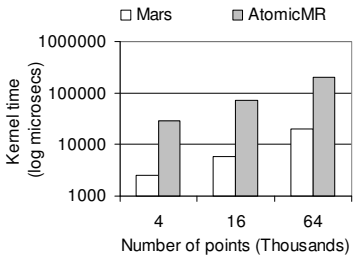

Fig. 3. Performance of atomic-based MapReduce vs. Mars using KMeans

It is worth noting that, our atomic-based implementation uses atomic operations at the granularity of a wavefront, i.e., one master thread in the wavefront is responsible for allocating more buffer for all threads in this wavefront. In KMeans and Matrix Multiplication, each map thread writes to the global buffer once, so atomic operation is called once per wavefront by a master thread. This implementation using atomics at the wavefront level fairly mimics the map phase of the MapCG[5] implementation.

V. StreamMR: Proposed MapReduce Framework

A. Design Overview

There are two major design issues in a MapReduce runtime framework on GPUs: 1) how to efficiently and correctly write output from the large number of threads to the global memory and 2) how to efficiently group intermediate results generated by the map function according to their keys.

1) Writing output with opportunistic preprocessing: As discussed in Section IV, using global atomic operations in the MapReduce framework can incur severe performance penalties on AMD GPUs. While Mars implementation does not employ atomic operations, it requires expensive preprocessing kernels to coordinate output from different threads to the global memory. In particular, the computation in the counting kernel is repeated in the actual compute (map or reduce) kernel; this redundant computation results in wasted compute resources.

StreamMR introduces a two-pass atomic-free algorithm that enables different threads to efficiently write their output to the global memory on AMD GPUs. Specifically, each work-group maintains a separate output buffer in global memory. In the first pass, these output buffers are preallocated according to a user-defined size. Each work-group independently writes the output to its own buffer without synchronizing with other work-groups. When the preallocated buffer is full, the compute kernel (map or reduce) switches to a counting procedure that only counts the sizes of different output records (without actually writing them), similar to the Mars design. In the second pass, an overflow buffer is allocated for the work-groups that use up their preallocated buffer in the first pass, using the sizes computed in the counting procedure. A separate kernel is then launched to handle the unwritten output of the first pass.

The StreamMR output design eliminates the need for global atomic operations. It can also greatly save the preprocessing overhead compared to Mars. For applications with output sizes that can be easily estimated, e.g., Matrix Multiplication and KMeans, the counting procedure and the second pass can be skipped altogether, yielding the most efficient execution. That is, the preprocessing only happens opportunistically.

For applications with output sizes that are hard to predict, StreamMR saves the counting computation corresponding to preallocated buffers during the first pass, whereas Mars performs the redundant counting computation for all output. In addition, in StreamMR, we record the output size per work-group as opposed to recording output size per thread in Mars, thus improving the prefix summing performance (as fewer size records need to be dealt with in the prefix summing).

2) Grouping intermediate results with atomic-free hash tables: Like MapCG, StreamMR organizes the intermediate output generated by the map phase using hash tables. However, MapCG uses atomic operations on global variables, e.g., compare-and-swap, to implement the hash table, which will incur performance penalty caused by the slow CompletePath on AMD GPUs. To address this issue, StreamMR maintains one hash table per wavefront, thus removing the need of using global atomics to coordinate updates from different workgroups to the hash table. Also, as explained in the next section, StreamMR leverages the lock-step execution of threads in a wavefront as well as atomic operations on local variables (i.e., variables stored in the local memory) to implement safe concurrent updates to the hash table of each wavefront. During the reduce phase, a reduce thread reduces the intermediate output associated with a specific entry in all hash tables, i.e., hash tables of all wavefronts.

B. Implementation Details

In StreamMR, each work-group maintains four global buffers as shown in Figure 4. Among these buffers, $Keys_i$ and $Values_i$ store keys and values of intermediate results. HT_i is the hash table of wavefront i. Figure 5 depicts the details of the hash table design. Each entry in the hash table contains two pointers to the head and tail of a linked list (hash bucket) stored in $KVList_i$. The head pointer is used to explore the elements stored in a hash bucket, and the tail pointer is used when appending a new element. Each element in $KVList_i$ associates every key to its value, and it contains a pointer to the next element in the linked list.

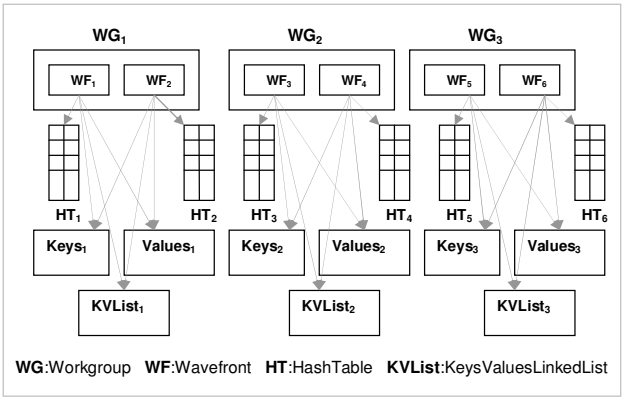

Fig. 4. Main data structures used in the map phase of StreamMR

1) Map Phase: Initially, every map thread executes the map function on its assigned input key/value pair. A map thread then collaborates with other threads on the same work-group *i* to determine its write location on the global buffers, i.e., $Keys_i$, $Values_i$, and $KVList_i$ without conflicting with other threads in the same work-group. This can be efficiently done using the system-provided atomic operations on *local* variables, leveraging the fact that atomic operations on local variables does not force memory access to follow the CompletePath.

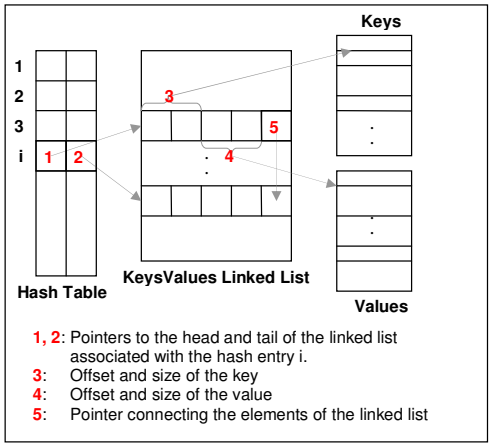

Fig. 5. Details of the hash table

To safely update the hash table HT_i, a single entry of the hash table should be updated by only one thread in the workgroup, this thread is named *master thread*. Before the *master thread* updates the hash table, all threads in the workgroup should be synchronized. However, since the threads of the workgroup may diverge based on the input characteristics, deadlock can occur during the synchronization. To address this issue, we decide to use one hash table per wavefront, for all threads in a wavefront are synchronized by the lock-step execution.

All threads of a wavefront use three auxiliary arrays stored in shared memory arrays to coordinate concurrent updates to the hash table of this wavefront. The first array is *HashedKeys*. Thread *i* writes the hash of its key to its corresponding entry *HashedKeys[i]*. The second array is *Slaves*, which is used to identify the *master thread* of each hash entry. The third array *KeyValListId* is used by the master thread to update the links on the linked list associated with the hash entry. In updating the hash table, all threads in the wavefront go through three steps as shown in Figure 6. First, all active threads in the wavefront write the hash of their keys to the *HashedKeys* array and the index of the inserted record to $KVList_i$ to the *KeyValListId* array. Second, every thread reads the hash keys of all other threads, and the first thread with a certain hash key is considered as a *master thread*. For example, if thread t_1, t_3, and t_5 all has the same key, then t_1 will be marked as the *master thread*. Finally, the *master thread* t_1 reads the indices of its slave threads, i.e., *KeyValListId[3]*, and *KeyValListId[5]*, and then it updates the tail of the hash entry *HashedKeys[1]* to refer to slave records, in addition to updating the links of these records to form the extended linked list.

2) Reduce Phase: Reducing the key/value pairs from different hash tables is not a trivial task. Since keys are different for in different hash tables, to insure all keys are handled, we run the reduce kernel with number of threads equals to the total number of entries of all hash tables i.e., number of entries per hash table $\times$ number of hash tables. In

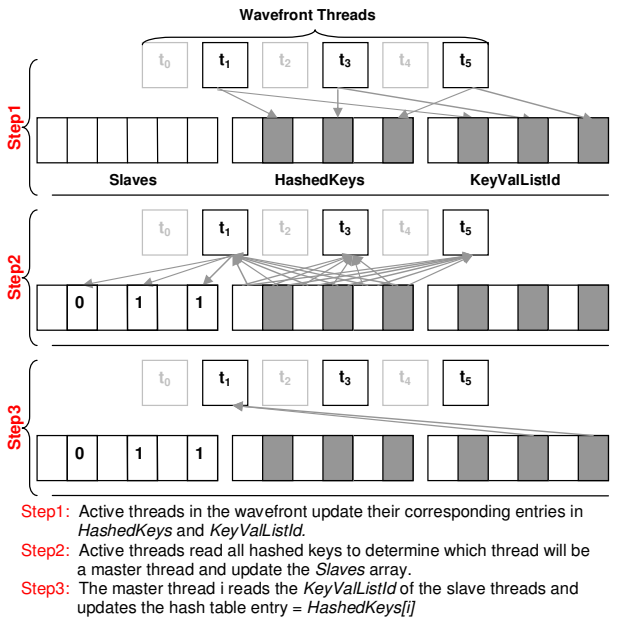

Fig. 6. Steps for updating the hash table assuming wavefront of 6 threads, and t_1, t_3, and t_5 are the active threads

particular, each reduce thread reduces the values associated to a specific hash entry beginning from certain hash table (ranging from hash table 0 to hash table $n-1$) passing through all subsequent hash tables. We then run another cleanup kernel to remove any redundant outputs. Note that, for applications generating the same keys for every workgroup, it is enough to invoke the reduce kernel with number of threads equals the number of hash entries in one hash table. Every thread reduces the values of a specific hash entry in all hash tables, and hence the cleanup kernel can be skipped.

Similar to the map phase, threads in the same wavefront collaborate using the system-provided atomic operations on local variables to write their final key/value pairs to the global buffers.

C. Optimizations

StreamMR provides several optimizations in addition to the basic design.

1) Map with Combiner: If the combiner function is available, the map phase can be modified so that instead of writing the map output directly to the global buffer, only one combined value is written per key. Specifically, the *master thread* generates the combined value of slave threads, and update the hash table accordingly. Since the map outputs are combined before being written to the global buffer, the number of global memory transactions can be significantly reduced.

In StreamMR, values produced by the slave threads are written to the shared memory to improve the combining performance. For values with variable sizes, the available shared memory may not be sufficient to hold values from all threads in the memory. Upon such an overflow, the number of active threads per wavefront is reduced from 64 threads (in case of AMD Radeon HD 5870 GPU) to 32 threads. Threads from 0 to 31 continue their processing and threads from 32 to 64 remains idle. When the first half of threads complete their processing, the other half starts. While processing the active threads, the used sizes are compared to the allocated sizes. If the overflow occurs again, the number of active threads is reduced to 16 threads, and so on until the used sizes fit the available shared memory. The overhead of this mechanism will be evaluated in section VI.

2) Reduce with Combiner: To improve the scalability of the reduce phase with regard to the number of wavefronts of the map kernel, more than one kernels can be launched. For instance, instead of having one kernel where one thread reduces the values of a certain hash entry from all hash tables, multiple kernels can be launched to reduce entires in a tree-like manner. Such a design allows more parallelism to be exploited during the reduction because reducing of a single hash entry is parallelized.

3) Image memory input: This optimization aims at improving memory access performance. When the input dataset is bound to the texture memory, the L1 and L2 texture caches can help reduce access to the global memory. When the input dataset is heavily reused by the kernel, we have found that this optimization can significantly improve performance on AMD GPUs.

D. Discussion

One limitation of using a separate buffer for each wavefront can cause inefficient memory utilization when the size of the initial buffer is too large. This limitation can be alleviate for applications with relatively predictable output sizes. The multi-buffer design may also cause inefficiency when the final output is copied back to the host memory. Assuming the allocated output buffers for all workgroups are stored in contiguous memory locations in the global memory, there are two options for transferring the final output back to the host memory. The first option is to copy only the used buffer from each workgroup. This requires multiple transfers i.e., one per workgroup. The second option is to copy all allocated buffers using only one transfer. In this case other unneeded buffers will be copied as well. Experiments have shown that the second option is more efficient, since it requires communicating with the host only once. However, the second option is still less perfect. We plan to investigate more efficient solution for this problem in the future work.

VI. Experimental Analysis

All of the experiments presented in this section are conducted on a 64-bit server with an Intel Xeon E5405 x2 CPU (2.00GHz) and 3GB of RAM. The equipped GPU is ATI Radeon HD 5870 (Cypress) with 512MB of device memory. The server is running the GNU/Linux operating system with kernel version 2.6.28-19. StreamMR and the testing applications are implemented with OpenCL 1.1 and compiled with AMD APP SDK v2.4.

We use three test applications that are commonly used in other MapReduce studies such as Mars and MapCG. These applications include:

- **Matrix Multiplication (MM)**. MM accepts two matrices X and Y as input and outputs matrix Z. Each element $z_{i,j}$ in Z is produced by multiplying every element in row i of X with the corresponding element in column j of Y and summing these products. The MapReduce implementation of MM includes only the map phase, where each map task is responsible for calculating one element of the output matrix.

- **KMeans (KM)**: KM is an iterative clustering algorithm. Each iteration takes a set of input points and a set of clusters, assigns each point to a closest cluster based on the distance between the point and the centroid of the cluster, and recalculates the clusters after. The iteration is repeated until clustering results converge (In our results we run only one iteration). The MapReduce implementation of KM include both map and reduce phases. The map function attaches the assigned points to their closest clusters, and the reduce function calculates the new coordinates of a cluster based on the attached points. Note that the combiner function is enabled for both map and reduce phases in StreamMR in our experiments.
- **String Match (SM)** SM searches an input keyword in a given document and outputs all matching locations. The MapReduce implementation of SM includes only the map phases. Each map task reads a chunk of the input document, character by character, and outputs the locations of any found matching words.

For each of the testing application, we use three input data sets, i.e., Small (S), Medium (M) and Large (L) whose size are given in Table I.

Applications	Dataset Size
$MatrixMultiplication(MM)$	S: 256, M: 512, L:1024
$KMeans(KM)$	S: 4096 points, M: 16384, L: 65536
$StringMatch(SM)$	S: 16MB, M: 64MB, L: 100MB

TABLE I
DATASET SIZES PER APPLICATION

A. Applications Performance

1) Comparison to Mars: We first evaluate the performance of StreamMR against Mars with three test applications. In order to execute Mars, which is originally implemented in CUDA, on AMD GPUs, we have reimplemented Mars with OpenCL. The bitonic sort and scan algorithms available in the AMD Stream SDK are used to implement the sorting and scanning phases of Mars.

As shown in Figure 7, StreamMR outperforms Mars for almost all testing applications with speedups between 0.96 to 4.7. For applications with the map phase only, i.e. MM and SM, the advantage of StreamMR comes from the reduced preprocessing overhead (counting and prefix summing phases as detailed in Section V). To better understand the performance gain of StreamMR over Mars, we break down the execution time of the large input dataset into five phases, i.e., preprocessing, map, group, reduce, and copy result (from GPU to CPU), as shown in Figure 8. To get normalized times, the execution times of each phase is divided by the total execution time of the corresponding Mars run. For MM, the Mars preprocessing overhead is 9.7% of the total execution time in Mars. Since the output size is fixed, the preprocessing time of MM is negligible in StreamMR. As a consequence, StreamMR outperforms Mars by 1.1 times on the average. On the other side, in SM, since the size of the output is variable, Mars preprocessing phases, especially the counting phase consumes significant portion of the map time. Specifically, the counting phase passes through the whole file and searches for matches to accurately determine the size of the output of each map task. These preprocessing phases represent 30.54% of the total map time. So our framework better improves the performance by 1.3-fold speedup on the average.

For KM, as shown in Figure 8, although the overhead of Mars preprocessing kernels is small i.e., 3.54% of the total time, the speedup of our framework over Mars is the highest among all applications i.e., 3.9-fold speedup on the average. This returns to two reasons; first, the efficiency of the hashing-based grouping over sorting-based one which results in reducing the number of accesses to the global memory. Second, the larger number of threads contributing in the reduce phase through the combiner function which result in improving the reduce time.

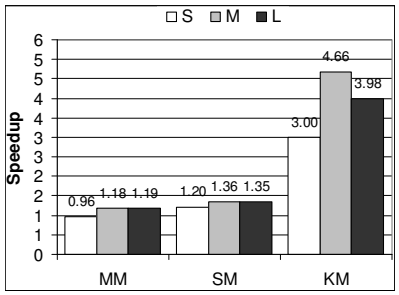

Fig. 7. Speedup of StreamMR over Mars using small, medium, and large datasets

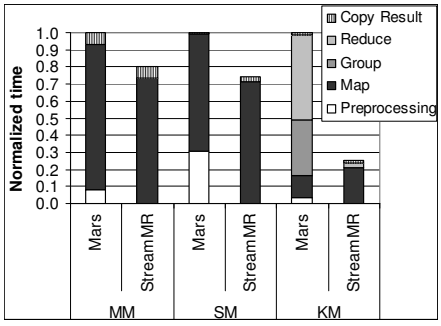

Fig. 8. Execution time breakdown of Mars and StreamMR using Large dataset

2) Comparison to Atomic-based MapReduce: As we discussed earlier, state-of-the-arts MapReduce frameworks in CUDA use atomic operations to coordinate the output from different threads. To evaluate atomic-based MapReduce designs on AMD GPU, we modified Mars by removing the preprocessing kernels and using atomic operations to determine write locations of each map and reduce thread in global memory. For MM and KM, the atomic operation is used in a wavefront granularity, where only one thread per wavefront executes the atomic operation. For SM, since threads in a wavefront may execute divergent paths, the atomic operation is issued per thread. We named this modified version of MapReduce AtomicMR.

As we discussed in Section II, atomic operations on AMD GPUs can force all memory access to use a slow CompletePath instead of the normal FastPath, can thus can result in severe performance degradation for memory-bound applications. StreamMR addresses this issue with an atomic-free design. As shown in Figure 9 and Figure 10, for MM,

StreamMR significantly outperforms AtomicMR, i.e., with an average speedup of 30.3-fold. It turns out that the ALU:Fetch ratio (measured by AMD APP Kernel Analyzer v1.8) of MM is 0.4. Such a low ALU:Fetch ratio suggests that MM is indeed a memory-bound application. On the other hand, the ALU:Fetch ratio of SM is very high, i.e. 9.89, suggesting that SM is more compute-bounded. Consequently, SteamMR does not show performance improvements over AtomicMR for SM.

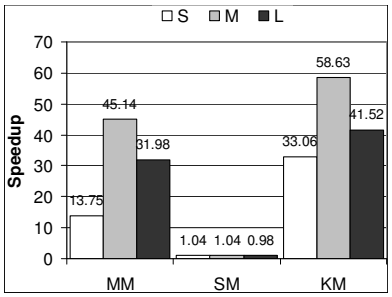

Fig. 9. Speedup of StreamMR over Atomic-based MapReduce using small, medium, and large datasets

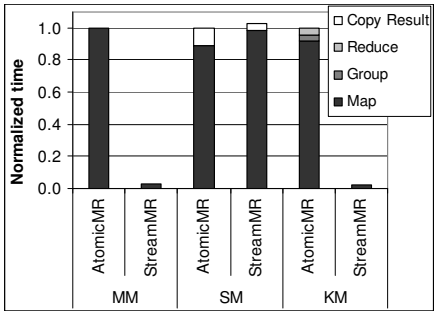

Fig. 10. Execution time breakdown of AtomicMR and StreamMR using large dataset

For KM, the average speedup of SteamMR over AtomicMR is 44.4-fold. Again, one of the reason is that KM is also memory-bounded, as indicated by an ALU:Fetch ratio of 0.9. In addition, the map phase of KM contributes to more than 90% of the total execution time as shown in Figure 10.

B. Overflow Handling Overhead

In this experiment, we aim at quantifying the overhead of the overflow handling mechanisms i.e., global and local buffers overflow. For SM, there is a high probability for the global overflow to occur since the size of the output is nondeterministic and depends on the input file and the keyword. For KM, if the local buffer is not set appropriately, a local overflow may be encountered. For MM, since the size of the output is deterministic, then the overflow can be avoided.

We run SM using medium-size dataset and varied the global buffer size to study the effect of global overflow on the performance. we reduce the size of the preallocated output buffer, so overflow occurs, and another map kernel is executed. The overflow percentage is the ratio between the number of matches emitted by the second map kernel and the total number of matches. As shown in Figure 11, the speedup of StreamMR over Mars slightly decreases from 1.37 to 1.29 when the percentage of overflow reaches 18%. As the overflow percentage increases to 88%, the speedup drops further to 0.94. This is because StreamMR will incur more and more counting overhead as the overflow percentage increases. However, the above performance results also suggest the overhead of global overflow is tolerable.

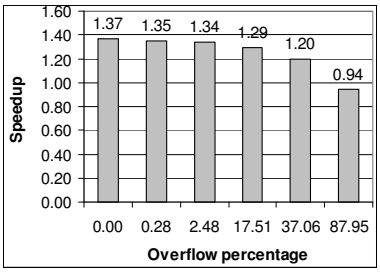

Fig. 11. Effect of global overflow on the speedup over Mars using StringMatch

For KM, we varied the allocated local buffer, so instead of running all threads per wavefront concurrently, they run on two and four consecutive iterations. As a result, the map kernel execution time increases as shown in Figure 12. Specifically, the speedup compared to overflow-free case is 0.87 and 0.78 for two and four consecutive iterations respectively.

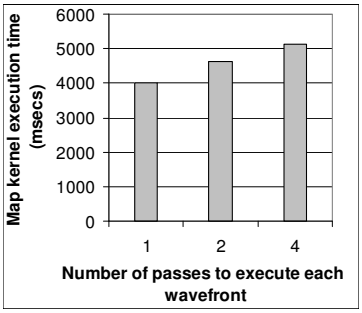

Fig. 12. Effect of local overflow on the Map kernel execution time of KMeans

C. Impact of Using Image Memory

In this experiment, we evaluate the effect of using texture memory instead of global memory to store the input dataset. Since the data retrieved from the texture memory are cached, we expect applications with data locality to benefit from this feature. MM is an example of such applications since a single row is accessed by several map tasks. For SM and KM, since each thread works in a different piece of input data, texture caching may not be beneficial.

For MM, we have found that using texture memory to store the input matrices, improves the performance of the map kernel significantly. More specifically, the speedup of the map kernel over non-texture map kernel is 9.77 and 3.84 for 256 X 256 and 512 X 512 matrices respectively. Moreover, the overall application speedup is 3.92 and 2.63 for 256 X 256 and 512 X 512 matrices, respectively.

VII. Conclusions and Future Work

In this paper, we revisit the design of MapReduce framework on AMD GPUs. We found that the atomic operations

used in state-of-the-arts GPU MapReduce frameworks, e.g., MapCG, can cause severe performance degradation on AMD GPUs. Existing atomic-free implementation of MapReduce, i.e. Mars, has several disadvantages i.e., preprocessing kernels, and the time-consuming sorting in grouping intermediate results. Consequently, we designed StreamMR, an atomic-free implementation of MapReduce optimized for AMD GPUs. StreamMR uses opportunistic preprocessing and groups intermediate results with global-atomic-free hash tables. Experiments have shown that our implementation provides significant improvement compared to Mars and atomic-based MapReduce frameworks. For future work, we plan to study the performance using more applications and evaluate the performance of StreamMR on NVIDIA GPUs.

ACKNOWLEDGMENT

This work is supported in part by 1) the VTMENA program, 2) AMD Research Faculty Fellowship, and 3) NSF grants IIP-0804155 for NSF I/UCRC CHREC and CNS-0916719. The authors would like to thank Feng Ji for his comments and feedback that helped develop this work.

REFERENCES

[1] Amazon.com. Amazon Elastic Compute Cloud. http://www.amazon.com/gp/browse.html?node=201590011.

[2] AMD. Stream Computing User Guide. http://www.ele.uri.edu/courses/ele408/StreamGPU.pdf, December 2008.

[3] AMD. OpenCL Programming Guide rev1.03. http://developer.amd.com/gpu_assets/ATI_Stream_SDK_OpenCL_Programming_Guide.pdf, June 2010.

[4] Bingsheng He, Wenbin Fang, Naga K. Govindaraju, Qiong Luo, and Tuyong Wang. Mars: a MapReduce Framework on Graphics Processors. In *17th International Conference on Parallel Architectures and Compilation Techniques*, pages 260–269. ACM, 2008.

[5] Chuntao Hong, Dehao Chen, Wenguang Chen, Weimin Zheng ,and Haibo Lin. MapCG: Writing Parallel Program Portable Between CPU and GPU. In *19th International Conference on Parallel Architectures and Compilation Techniques*, pages 217–226. ACM, 2010.

[6] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24, 2007.

[7] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *6th Symposium on Operating Systems, Design, and Implementation*, 2004.

[8] Marwa Elteir, Heshan Lin, and Wu-chun Feng. Performance Characterization and Optimization of Atomic Operations on AMD GPUs. In *IEEE Cluster 2011*, Austin, TX, USA, September 2011.

[9] Feng Ji and Xiaosong Ma. Using Shared Memory to Accelerate MapReduce on Graphics Processing Units. In *IEEE 25th International Parallel and Distributed Processing Symposium*, 2011.

[10] Khronos Group. The Khronos Group Releases OpenCL 1.0 Specification. *http://www.khronos.org/news/press/releases*, 2008.

[11] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters. In *ACM SIGMOD International Conference on Management of Data*, pages 1029–1040, New York, NY, USA, 2007. ACM.

[12] Marc de Kruijf and Karthikeyan Sankaralingam. Mapreduce for the Cell Broadband Engine Architecture. *IBM Journal of Research and Development*, 53(5):10–1, 2009.

[13] Marwa Elteir, Heshan Lin, and Wu-chun Feng. Enhancing MapReduce via Asynchronous Data Processing. In *IEEE 16th International Conference on Parallel and Distributed Systems*, pages 397–405. IEEE, 2010.

[14] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *USENIX Symposium on Operating Systems Design and Implementation*, 2008.

[15] NVIDIA. NVIDIA CUDA Programming Guide-2.2. http://developer.download.nvidia.com/compute/cuda/2.2/toolkit/docs/, 2009.

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

[17] Steven Y. Ko, Imranul Hoque, Brian Cho, and Indranil Gupta. On Availability of Intermediate Data in Cloud Computations. In *12th Workshop on Hot Topics in Operating Systems*, 2009.

[18] Jeff A. Stuart and John D. Owens. Multi-GPU MapReduce on GPU Clusters. In *IEEE 25th International Parallel and Distributed Processing Symposium*, 2011.

[19] Steffen Viken Valvag and Dag Johansen. Oivos: Simple and Efficient Distributed Data Processing. In *IEEE 10th International Conference on High Performance Computing and Communications*, pages 113–122, Sept. 2008.