Grex: An Efficient MapReduce Framework for Graphics Processing Units

Can Basaran, Kyoung-Don Kang

Department of Computer Science, Binghamton University

Abstract

In this paper, we present a new MapReduce framework, called Grex, designed to leverage general purpose graphics processing units (GPUs) for parallel data processing. Grex provides several new features. First, it supports a parallel split method to tokenize input data of variable sizes, such as words in e-books or URLs in web documents, in parallel using GPU threads. Second, Grex evenly distributes data to map/reduce tasks to avoid data partitioning skews. In addition, Grex provides a new memory management scheme to enhance the performance by exploiting the GPU memory hierarchy. Notably, all these capabilities are supported via careful system design without requiring any locks or atomic operations for thread synchronization. The experimental results show that our system is up to 12.4x and 4.1x faster than two state-of-the-art GPU-based MapReduce frameworks for the tested applications.

1. Introduction

As the modern GPUs are becoming more programmable and flexible, their use for general purpose computation is increasing. NVIDIA's CUDA [1] and ATI's OpenCL [2] support general purpose GPU programming. Generally, a GPU has a significantly larger number of processing elements than a modern multicore CPU does, providing massive hardware parallelism. For example, an NVIDIA GeForce 8800 GPU provides 128 cores and a maximum 768 resident threads. Also, a more advanced NVIDIA Fermi GPU provides 512 cores that can accommodate up to 1536 threads.[1] Especially, GPUs are designed to support immense data parallelism; that is, GPU hardware is optimized for threads to independently process different chunks of data in parallel.

MapReduce [3] is a popular programming model designed to process a large amount of data in parallel. A user developing a data parallel application via

Email addresses: cbasaran@cs.binghamton.edu (Can Basaran), kang@binghamton.edu (Kyoung-Don Kang)

[1] In this paper, we focus on NVIDIA GPUs; however, the principal ideas of our approach are generally applicable to the GPUs from the other vendors. Hereafter, a GPU refers to an NVIDIA GPU.

MapReduce is only required to write two primitive operations, namely map and reduce functions. A MapReduce application is called a job. Given a job, the MapReduce runtime partitions the job into smaller units called tasks. It assigns tasks and data chunks to workers, which have available computational resources, called slots, to process the data chunks by running the tasks.[2] The workers process different data chunks in parallel independently from each other. The workers running map tasks process input <*key, value*> pairs to generate intermediate <*key, value*> pairs. The reduce function merges all intermediate <*key, value*> pairs associated with the keys that are in the same logical group. As the runtime handles scheduling and fault tolerance in a transparent manner, it is easy for a user to write data parallel applications using MapReduce.

Given that both GPUs and MapReduce are designed to support vast data parallelism, it is natural to attempt to leverage the hardware parallelism provided by GPUs for MapReduce. Although doing this may seem simple on the surface, it faces a number of challenges especially to process variable size data for industrially relevant and highly data parallel applications, such as natural language processing and web document analysis:

- Most GPUs do not support dynamic memory management. Thus, it is challenging to process variable size data in GPUs. Furthermore, the maximum size of variable length data is often unknown *a priori* in natural language processing and web document analysis. Hence, existing techniques developed to deal with variable size data, such as padding, are not directly applicable.[3]

- While processing variable size data, MapReduce tasks may suffer from load imbalance. To process variable size textual data, the MapReduce runtime sequentially scans the entire input data and assigns the same number of lines to each map task in a phase called input split. Unfortunately, input split is not only slow due to the sequential execution but also subject to potential load imbalance among the map tasks, since the amount of data may vary from line to line.

- In the reduce phase of MapReduce, an entire group of intermediate data having the same key is hashed to a single reduce task, incurring severe load imbalance among the reduce tasks in the presence of data skews [5, 6]. For example, certain words or URLs tend to appear more frequently than the others in natural language or web documents. As a result, a reduce task assigned a popular key group is likely to have a significantly larger number of intermediate <*key, value*> pairs to process than the other reducers do. Load imbalance due to the input split and data skews may adversely affect

[2] In Hadoop [4], an open source implementation of MapReduce by Yahoo, a slot represents an available CPU core by default.

[3] To apply padding, the entire set of the variable size data has to be preprocessed to find the size of the biggest data item first. After that, actual padding needs to be performed. These extra steps may introduce a significant overhead when the data set is large.

the performance, since it is known that the slowest task determines the speed of a job that consists of multiple tasks executed in parallel.

- GPU memory hierarchy is significantly different from the host (i.e., CPU side) memory hierarchy. Since the number of the cycles consumed by the fastes and slowest memory in the hierarchy is different by several orders of magnitude, it is challenging but important to efficiently utilize the GPU memory hierarchy.

To tackle these challenges, we design, implement, and evaluate a new GPU-based MapReduce framework called Grex (meaning a herd in Greek). Our goal is to efficiently process variable size data for the entire MapReduce phases beginning from input split to reduction, while minimizing potential load imbalance in MapReduce. At the same time, we aim to efficiently utilize the GPU memory hierarchy, which is important for general MapReduce applications executed in GPUs to process either variable or fixed size data. A summary of the key contributions made by Grex towards achieving these goals follows:

- To efficiently handle variable size data, Grex replaces the sequential split phase in MapReduce with a new *parallel split* step. In parallel split, Grex *evenly distributes input data* in terms of the number of bytes to the threads that tokenize input data in parallel.[4] Notably, Grex does not need to know the size of any token in advance to do parallel split.

- After parallel split, each thread executes the user-specified map() function to process the tokens found as a result of parallel split. After finishing to execute the map() function, each thread that has found the beginning of a token computes a *unique address* to which an intermediate data item will be written without communicating with the other threads. Therefore, *no locks or atomics* for synchronization are needed. Similarly, the output of the reduce phase is written without requiring any locks or atomic operations.

- Grex assigns an *equal number of intermediate <key, value> pairs to the reduce tasks* to execute the user-specified reduce() function in a parallel, load-balanced manner even if the keys are skewed.

- Although MapReduce deals with *<key, value>* pairs, only keys are needed for a considerable part of data processing in a number of MapReduce applications. In word count, for example, keys can be processed by map tasks and sorted and grouped together with no values. After grouping the same keys together, the frequency of each unique key has to be computed.

[4]In this paper, a meaningful unit of variable size data, e.g., an English word or URL, is called a token. Converting raw data, e.g., natural language or web documents, into tokens is called tokenization. For fixed size input data such as integers or floats, Grex simply assigns the same number of data items to the map tasks and skips the parallel split step.

Thus, it is not necessary to generate values in an earlier stage in certain applications. Based on this observation, Grex supports a new feature called *lazy emit*. Grex provides an API to let a user specify the MapReduce phase of a specific application where Grex needs to create values and emit complete <*key, value*> pairs to the GPU DRAM, called global memory. Before the user-specified phase begins, Grex does not generate or store any values but only deals with keys. By doing this, Grex intends to decrease the overhead to handle intermediate data in terms of computation and memory usage.

- Through all the MapReduce phases, Grex takes advantage of the *GPU memory hierarchy* to decrease the delay for memory access. To this end, Grex provides an API by which a user (i.e., a MapReduce application developer) can specify the size and type (read-only or read/write) of a buffer that will be stored in global memory and cached by Grex using faster memory of the GPU in a transparent manner.

Although a number of GPU-based MapReduce systems have previously been developed [3, 4, 7, 8, 9, 10, 11, 12], we are not aware of any other GPU-based MapReduce framework that supports all these features.

We have implemented Grex and tested our implementation on a PC with an AMD quad-core CPU using two different generations of NVIDIA GPUs. We have compared the performance of Grex to that of Mars [7] and MapCG [8], which are open source GPU-based MapReduce frameworks. For performance evaluation, we have used six common applications with different needs in terms of MapReduce computations, i.e., string match, page view count, word count, inverted index, similarity score, and matrix multiplication. The first four applications deal with variable size data, while the last two applications process fixed size data. By considering a diverse set of applications, we thoroughly evaluate the performance of Grex in comparisons to Mars and MapCG. Our performance evaluation results indicate up to 12.4x speedup over Mars and up to 4.1x speedup over MapCG using the GeForce GTX 8800 GPU and GeForce GTX 460 GPU that supports atomic operations needed by MapCG, respectively.

The remainder of this paper is organized as follows. Related work is discussed in Section 2. A brief description of MapReduce and the general GPU architecture is given in Section 3. The design of Grex is described in Section 4. Grex code snippets are discussed in Section 5 to further illustrate the key concepts introduced in this paper. The performance evaluation results are discussed in Section 6. Finally, Section 7 concludes the paper and discusses future work.

2. Related Work

MapReduce is developed by Google [3]. Hadoop [4] is an open-source implementation of MapReduce developed by Yahoo. Both MapReduce and Hadoop are designed for commodity servers. Phoenix [9] and its extensions [10, 13]

are one of the first projects to implement a MapReduce framework for a multi-core architecture with shared memory. Ostrich [14] intends to improve Phoenix by implementing an iterative map phase. Similarly, Grex iteratively processes input data by applying the tiling technique [15]. There is also a MapReduce implementation for the IBM Cell architecture [16].

Property	Grex	Mars	MapCG	[11]	[17]	Mithra	GPMR
Single pass execution	yes	no	yes	yes	yes	yes	yes
Locks or atomics	no	no	yes	yes	yes	no	no
Parallel split	yes	no	no	no	no	no	no
Load balancing	yes	no	no	no	no	no	yes
Lazy emit	yes	no	no	no	no	no	no
GPU caching	yes	no	no	yes	yes	no	no
GPU cluster	no	no	no	no	no	yes	yes

Table 1: Comparisons of the GPU-based MapReduce frameworks

Several GPU-based MapReduce frameworks have been developed by different researchers. A GPU-based MapReduce framework developed by Catanzaro et al. [18] supports a constrained MapReduce model that assumes that the keys are known in advance and a map function produces only one output. Due to these assumptions, the applicability of [18] is limited. Especially, it is not directly applicable to process raw (i.e., not tokenized) variable size data. Table 1 summarizes the key features provided by several GPU-based MapReduce frameworks that support full MapReduce functionalities. Mars [7] is a GPU-based MapReduce framework. It shows up to 10x speedup over Phoenix [9]; however, it is not optimized for the GPU architecture. To overcome the lack of dynamic memory management in many GPUs, Mars computes the precise amount of GPU memory required for processing a MapReduce job in the first pass. It does actual data modification and generation in the second pass. Since the first pass performs every MapReduce task except for producing actual output, Mars essentially executes a MapReduce job twice. Neither does it utilize the GPU memory hierarchy.

MapCG [8] uses atomic operations to support dynamic memory management in global memory. In this way, MapCG avoids two pass execution of Mars. However, atomic operations require sequential processing of dynamic memory allocation or deallocation requests, incurring performance penalties. Ji and Ma [11] enhance the performance of MapReduce applications using shared memory in a GPU. However, in their approach, half the threads are used to support synchronization based on atomic operations, incurring potentially non-trivial overheads. Chen and Agrawal [17] efficiently utilize shared memory to support partial reduction of <*key, value*> pairs. However, their approach is only applicable to communicative and associate MapReduce operations. Different from these GPU-based MapReduce frameworks [18, 7, 8, 11], Grex 1) processes a MapReduce job in a single pass, 2) supports parallel split, 3) facilitates load-balanced data partitioning among the map and reduce tasks of a

job, 4) supports lazy emit, and 5) efficiently utilizes the GPU memory hierarchy for performance enhancement. Further, it is not limited to specific operators. It takes advantage of not only shared memory but also constant memory and texture cache, while requiring no atomic operations or locks.

Mithra [19] uses Hadoop as a backbone to cluster GPU nodes for MapReduce applications. By building a GPU cluster on top of Hadoop, Mithra takes advantage of powerful features of Hadoop such as fault tolerance, while leveraging a large number of GPU threads. As a result, it significantly outperforms a CPU-only Hadoop cluster. However, it does not support the key optimizations of Grex described before. GPMR [12] is an open source MapReduce framework designed for a GPU cluster. It relies on simple clustering of GPU nodes without supporting a distributed file system or fault tolerance. Instead, it focuses on providing novel features for partial data aggregation to decrease the I/O between the CPU and GPU in a node as well as network I/O between the map and reduce tasks. However, GPMR does not consider to support the main features of Grex. Also, the current GPMR implementation processes already tokenized data only. Thus, variable size data have to be tokenized using the sequential input split phase in Hadoop or by any other means before being processed by GPMR. Mithra [19] and GPMR [12] are complementary to Grex in the sense that the current version of Grex uses a single GPU, similar to Mars [7], MapCG [8], and [11]. Grex could be combined with Hadoop or GPMR to support MapReduce applications in a GPU cluster. A thorough investigation is reserved for future work.

Kwon et al. [6] investigate several causes of skews in Hadoop applications. Grex evenly distributes data to the map and reduce tasks. This approach will alleviate partitioning skews; however, we do not claim Grex completely solves skew problems. For example, a certain <*key, value*> pair can be computationally more expensive to process than the others [6]. Efficiently dealing with potential skews in a GPU-based MapReduce framework is an important yet rarely explored research problem. Other GPU-based MapReduce frameworks including [7, 8, 11, 12, 19] do not consider this problem.

There have been many recent applications of GPU computing to different problem domains such as Fast Fourier Transform [20], matrix operations [21], encryption and decryption [22], genetics [23], intrusion detection [24], and databases [25]. Using GPUs, these research efforts have achieved significant performance improvement [26, 27]. These results call for a general GPU-based parallel programming framework to make the architectural advantages of GPUs available with reduced complexity compared to manual architectural optimizations.

3. Preliminaries

In this section, backgrounds of MapReduce and GPUs needed for the rest of the paper are given. Through the paper, word count is used as a running example to explain key concepts of MapReduce and Grex especially in terms of dealing with variable size data. Although word count is conceptually simple,

it requires all the MapReduce phases and deals with variable-size textual data challenging to handle in a GPU.

3.1. MapReduce

A MapReduce framework executes an application as a set of sequential phases such as split, map, group, and reduce. Each phase is executed by a set of tasks that run in parallel. A task is sequentially processed by a single processing element called a worker. In the split phase, the input is split into smaller subproblems called records and distributed among the workers. For example, a text file such as a web document can be split at new lines in the word count application. In the map phase, the input data is mapped to intermediate <*key, value*> pairs by a user defined map() function. In word count, a map task can be designed to process a user-specified number of text lines to emit <*word, 1*> pairs for each word in the lines. In the group phase, intermediate <*key, value*> pairs are sorted and grouped based on their keys. Finally, the intermediate pairs are reduced to the final set of <*key, value*> pairs by a user defined reduce() function. As an example, for 10 <*'the', 1*> intermediate <*key, value*> pairs produced by the map tasks, one new pair, <*'the', 10*> is emitted as the final output. (Some applications, e.g., string match, do not need grouping or reduction.)

Generally, a user is required to implement the map() and reduce() functions and the grouping logic, which are often specific to applications. Since a map/reduce task or grouping is executed sequentially in MapReduce, a user is only required to specify *sequential* logic. The rest of the problem, such as data distribution to tasks, collection of results, and error recovery for parallel processing, is handled by the MapReduce runtime system. Thus, developing a parallel application via MapReduce does not impose undue complexity of parallel programming on users.

3.2. General Purpose Graphics Processing Units

Graphics processors have architectural advantages that can be harnessed to solve computational problems in domains beyond graphics [27]. Further, the introduction of general purpose programming languages such as CUDA [1], Stream [28], and OpenCL [2] has promoted their use as general purpose processors [27].

GPU code has an entry function called a *kernel*. A kernel can invoke other (non-kernel) device functions. Once a kernel is submitted for execution, the host has to wait for its completion. No interaction is possible with a running kernel. A GPU has a number of SIMT (Single Instruction Multiple Threads) multiprocessors as shown in Figure 1. Threads are temporarily bound to these SIMT elements. Thus, a GPU can accommodate thousands or even more live threads. The number of processing elements and maximum number of active threads vary among GPUs [27, 29]. Threads are grouped into blocks. A (thread) block is scheduled dynamically to run on one of the available multiprocessors. All the threads of one block execute the same code because the underlying

architecture is SIMT. When the threads in a block branch to different code segments, branches are executed sequentially. This phenomenon, called control divergence, can result in dramatic performance degradation. Thus, the threads in Grex are designed to execute the same logic at a time to avoid divergence.

To support efficient memory access, GPUs employ a sophisticated memory hierarchy [30]. Specifically, there are five types of memory as shown in Figure 1: 1) a set of general propose registers per thread; 2) shared memory that is shared by all the threads in a block. It can be used as a software managed cache to cache data in the off-chip global memory; 3) a read-only constant memory that can be used as software managed cache for the global memory or to store frequently accessed constant data that is small in size; 4) read-only hardware managed texture cache that caches global memory; and 5) global memory. The size and access delay generally increases from 1) to 5). General purpose registers are several orders of magnitude faster than global memory whose access latency is hundreds of cycles. Global memory is the largest memory in the GPU. It can be accessed by any thread and even from the host CPU. However, it is the slowest memory.

Shared memory is almost as fast as the registers. Every thread of one thread block shares the shared memory; therefore, the threads in one thread block can communicate easily with each other. However, a thread in one thread block cannot access the shared memory of a different thread block. Constant and texture memory have short access latency upon a cache hit. Upon a miss, they have the same latency as global memory. Constant and texture memory can be used to store read-only data. Any thread can access any data stored in constant or texture memory. Since constant memory allocation is static, the amount of constant memory to be used must be known at compile time. As a result, constant data have to be copied to the fixed address before it can be accessed. Hence, in Grex, constant memory is used to keep fixed-size read-only data, such as a translation table used to convert upper case characters to lower

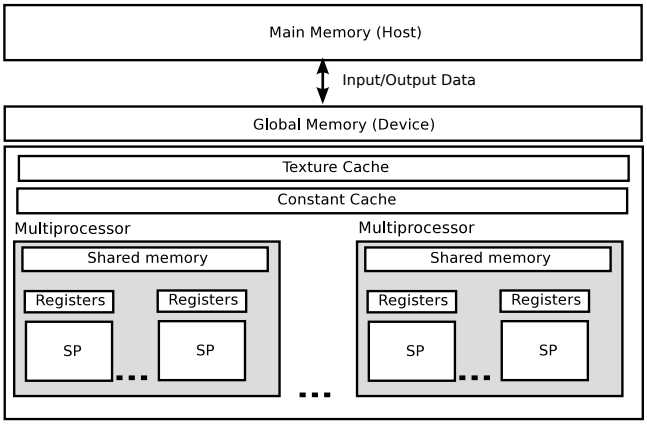

Figure 1: General Purpose GPU Architecture. (SP: Streaming Multi-Processor)

Algorithm 1 Control flow of Grex for processing variable size data

1. Read data blocks from global memory and cache them evenly in shared memory across the thread blocks. Execute the parallel split phase. (Skip this step if the data size is fixed.)
2. Execute the map phase. Compute a unique address for each intermediate key found after finishing the map phase. Using the unique addresses, write the keys to global memory in parallel without relying on locks or atomics.
3. Perform boundary merge in global memory to combine a key divided between two adjacent thread blocks, if any. (Skip this step if the data size is fixed.)
4. Group the intermediate keys via parallel sorting.
5. Read the intermediate keys from global memory and cache them evenly in shared memory across the thread blocks. Execute the reduce phase. Merge the results of the reduce phase performed by two adjacent thread blocks if necessary. (Skip this step if the data size is fixed.)
6. Perform lazy emit. Write the results to global memory. (Lazy emit is optional. If lazy emit is not used, the keys in the previous steps in this pseudo code should be replaced with <*key, value*> pairs.)

case ones in word count. If the size of a user specified data structure exceeds the amount of constant memory, a compile time error is issued by Grex. Unlike shared memory and constant memory, texture cache is managed by hardware. It does not require static allocation and provides good performance for random access of read-only data.

4. System Design

This section describes the MapReduce phases and buffer management scheme of Grex.

4.1. MapReduce Phases in Grex

Grex has five stages of execution: 1) parallel split, 2) map, 3) boundary merge, 4) group, and 5) reduce. A pseudo code that summarizes these stages is given in Algorithm 1. Note that the parallel split phase replaces the sequential input split phase of MapReduce [3, 4]. Also, lazy emit is not an additional phase but it postpones emitting values until the end of the entire MapReduce computation. Hence, only the boundary merge is added to the MapReduce control flow. Moreover, the parallel split and boundary merge phases are only needed to handle variable size data. If the size of data is fixed, Grex distributes the same number of data items, e.g., integers or floats, to each task for load

balancing. Therefore, no parallel split or boundary merge is needed for fixed size data. A detailed description of the MapReduce phases of Grex follows.

1. Parallel split phase: In the parallel-split step, Grex initially assigns one unit of data to each thread. For example, it can assign a single character in the input documents to a thread to compute the starting position and length of each string in the input as shown in Figure 2. For variable size data, a token is a key. For example, a word or URL in word counting or web document analysis is a key. Thus, in the rest of this paper, we use keys and tokens interchangeably.

The j^{th} thread in the i^{th} thread block, $T_{i,j}$, is associated with a unique thread identifier: $\text{TID}(i,j) = i \times BS + j$ where BS is the size of the data unit handled by a thread block, which is equal to the number of the threads in a block in the word count example given in this paper. $T_{i,j}$ writes 1 (or 0) to `vec[TID(`i,j`)]` in Step 1 of Figure 2, if the assigned byte, i.e., input[TID(i,j)], is a token (or non-token) byte using the user-specified `istoken()` function such as `ischar()` in word count. (Bit sting `vec` is a temporary data structure allocated in shared memory.)

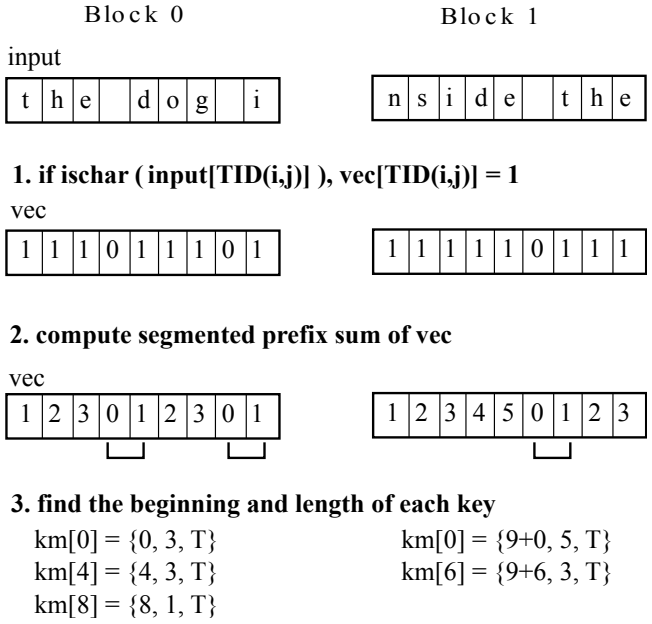

Figure 2: Kernel 1 - Parallel Split (Steps 1 and 2) and Map Phase (Step 3)

Grex applies the parallel prefix sum algorithm [31] to the bit string to compute the starting position and length of each key in parallel as shown in Step 2 of Figure 2. To give an illustrative example of parallel split, we introduce a data structure in Figure 3, which shows a temporary data structure, called `keyRecord`, used for the parallel split and map phases. The pointer field points to the beginning of a variable size key, e.g., a word or URL. The length field is

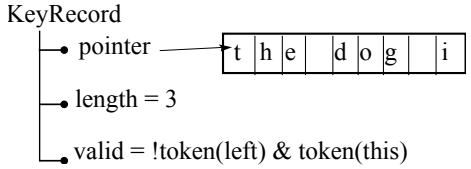

Figure 3: Key record data structure

used to store the length of a key. Also, the valid attribute is used to indicate the beginning of a key. (This data structure is not necessary for fixed size data.)

An instance of `keyRecord` is referred to `km` in Figure 2 and stored in a register by a thread. As shown in Step 2 of Figure 2, thread $T_{i,j}$ knows that it has the beginning of a token, if `vec[TID(`$i, j-1$`)]` $= 0$ and `vec[TID(`i, j`)]` $= 1$ where $i \geq 0$ and $j \geq 1$ and sets `km[TID(`i, j`)].valid` $=$ T(rue) in Step 3 of Figure 2. It also sets the `km[TID(`i, j`)].pointer` to the beginning of the key, while storing the length of the key to `km[TID(`i, j`)].length`.

In Grex, thread $T_{i,j}$ is assumed to have the beginning of a string, if $j = 0$ and `vec[TID(`i, j`)]` $= 1$. In fact, this is not always true, because a string can be divided between two adjacent blocks as shown in the figure. Such a case, if any, is handled by the boundary merge phase that will be discussed shortly. The boundary merge phase is needed, since two different GPU thread blocks cannot directly communicate with each other due to the current GPU hardware design.

The time complexity of the parallel prefix sum, which is the most expensive in the parallel split phase, is $\Theta((\frac{n}{N})lgn)$ where n is the length of the bit string indicating the input size in terms of the number of bytes and N is the total number of the GPU threads that can run at once. Thus, for large N, parallel split is considerably faster than a sequential algorithm used in MapReduce for tokenization, which is $\Theta(n)$. However, existing MapReduce frameworks including [3, 4, 7, 8, 9, 10, 11, 12] do not support parallel split.

Notably, a user only has to specify an `istoken()` function, such as the `ischar()` function in Figure 2. Parallel split is supported by the underlying Grex runtime system and hidden from the user. Since a user has to write a function similar to `istoken()` in any existing MapReduce framework for sequential input split, parallel split does not increase the complexity of developing a MapReduce application. Notably, all threads finish the parallel split phase simultaneously, because they process the same number of bytes using the same algorithm. Thus, no synchronization between them is required.

2. Map phase: A user specifies a `User::map()` function, similar to MapReduce [3] and Hadoop [4]. Grex further processes the keys in shared memory tokenized during the parallel split step via the map tasks, which execute the map() function in parallel. In word count, the parallel split and map phases are implemented as a single kernel, since the map function is only required to write the

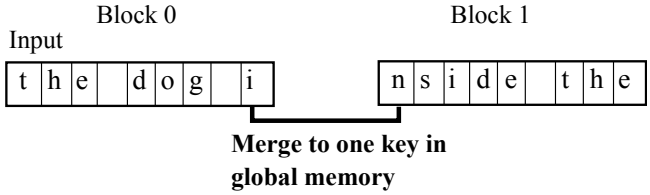

Figure 4: Kernel 2 - Boundary merge phase

keys found in the parallel split phase to global memory. If `km[TID(i,j)].valid == T` in Step 3 of Figure 2, thread T_{ij} writes `km[TID(i,j)].pointer` into `keys[`$\lceil$`TID(i,j)/2`$\rceil$`]` in global memory. Note that we divide the thread ID by 2, since a block cannot have more keys than half the bytes assigned to the block. A string, e.g., a word or URL, has at least one non-null character followed by a null character that indicates the end of the string. As the TID of a thread is unique, the threads that have valid pointers to the keys can write their pointers to the global memory location independently from each other. Thus, all the pointers to the beginnings of the keys are written to the global memory at the same time with no need to synchronize the threads.

The `input` data structure in Figure 2 is part of a Grex buffer, which keeps the input data and output of the map phase in global memory, respectively. Input is evenly divided among the thread blocks and cached in the shared memory of each block by Grex. Also, the `keys` data structure is the output buffer used to write the output of the map phase. A user can specify buffers needed to store input, intermediate, or output data in global memory. Grex caches them in shared memory or texture cache in a transparent manner. In addition, temporary data structures, such as `vec` and `km` in Figure 2, are created and managed by Grex. In the current version of Grex, `vec` and `km` are stored in shared memory and registers, respectively. After completing the execution of a kernel such as the one for parallel split and map, Grex deallocates the temporary data structures in shared memory.(Buffer management and caching is discussed in Section 4.2.)

In general, if the size of data is bigger than the shared memory size, the Grex steps are executed iteratively. Hence, the job size is not limited by the shared memory size. In addition, data upload to and download from the GPU in Grex is overlapped with the computation in the GPU to reduce the I/O overhead.

Grex supports an optional feature called **lazy-emit**, in which the value is neither created nor held in memory before the user-specified phase. For lazy-emit, a user only has to specify in which phase the value should be created and emitted to memory and what value should be emitted by defining a `User::emit(key)` function. For example, the emit() function for word count returns 1 for each word. Also, a user can optionally implement a `User::combine()` function to do partial reduction of intermediate <*key, value*> pairs produced in the map phase for an associative and commutative reduce() function, similar to MapReduce.

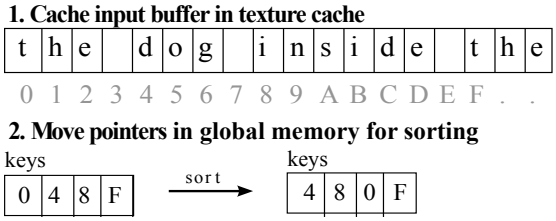

Figure 5: Kernel 3 - Sorting intermediate keys

3. Boundary merge phase: We have designed a simple yet effective parallel method, called boundary merge, to check whether any key is split between neighboring thread blocks and merge it to a single key. If there are $B > 1$ blocks, Grex executes a kernel using $B - 1$ threads to do boundary merge in global memory, since there are $B - 1$ block boundaries. When the map phase is completed, the result is written to global memory as discussed before. The $B-1$ threads for boundary merge can start immediately when the kernel for the parallel split and map phases completes. Also, they do not have to coordinate or communicate with each other, since they do not share any data or other resources with each other. Thus, they work independently in parallel.

To do boundary merge, thread i $(0 \leq i < B - 1)$ observes the end of the i^{th} block and the beginning of the $(i + 1)^{th}$ block to check whether or not both of them are valid (i.e., istoken() = true) as shown in Figure 4. If this condition is satisfied, thread i merges them to a single key and eliminates the partial key in the $(i + 1)^{th}$ block. Also, the thread decrements the number of the keys in the $(i + 1)^{th}$ block. At the end of the boundary merge step, the pointers to the variable size tokens are stored in `keys` buffer in global memory as shown at the bottom left in Figure 5.

4. Grouping phase: In this step, the intermediate <*key, value*> pairs are sorted, similar to MapReduce and Hadoop. In this paper, parallel bitonic sorting is used for grouping. For sorting, a user only has to specify how to compare keys through the `User::cmp()` function. For example, a user can use the `strcmp()` function in word count.

To decrease the memory access delay for parallel sorting, we bind the input buffer in global memory to the on-chip texture cache as shown in Step 1 of Figure 5. We use the texture cache rather than shared memory, since sorting generally involves data accesses across thread blocks. For grouping in Grex, data comparisons are done in the texture cache, which is read-only and managed by the hardware. Actual movements of the pointers to variable size data, if necessary for sorting, are performed by Grex in global memory as shown in Step 2 of Figure 5. As a result, the pointers in the figure are sorted in non-ascending lexicographic order of the keys, i.e., words in word count.

For grouping, Mars [7] and MapCG [8] create an index space to keep the

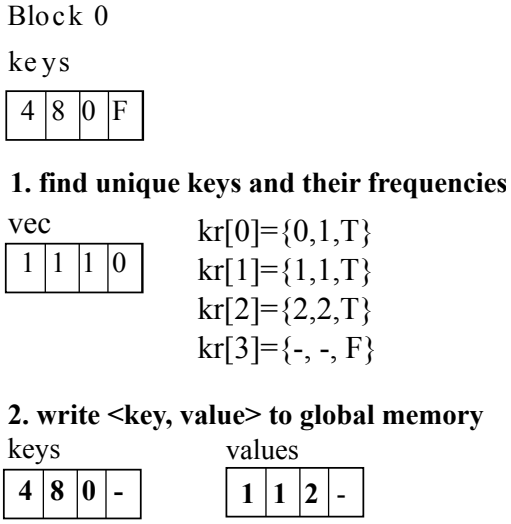

Figure 6: Kernel 4 - Reduction

<*key, value*> pairs and apply hashing to find <*key, value*> pairs in global memory. Our approach is more efficient than these methods, since it reduces the memory consumption as well as the delay and contention for global memory access by leveraging texture cache and lazy emit, while requiring no index space.

5. Reduce phase: Grex provides a `User::reduce()` API to let a user write a reduce function specific to the application of interest. To significantly enhance performance, Grex uses a large number of thread blocks to reduce <*key, value*> pairs in a parallel, load balanced manner. Grex evenly distributes intermediate <*key, value*> pairs to each reduce task to avoid load imbalance due to data skews as discussed before. (Only one thread block is used in Figure 6 due to the small data size.)

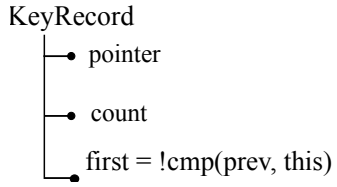

Figure 7: Key record data structure for reduction

In the reduce phase, Grex uses a temporary data structure in Figure 7. The pointer field points to the beginning of a variable size key. The count field is used to compute the frequency of a key's occurrences in natural language or web documents. Also, the first boolean attribute is used to indicate the first occurrence of a key. An instance of this data structure is called `kr`. It is stored

in a register by a thread. (This data structure is unnecessary for fixed size data, similar to the temporary data structures used for the parallel split and map phases.)

In Step 1 of Figure 6, each thread is assigned a word and it compares its word to the word of the next thread (if any). A thread sets its bit in the temporary bit string, `vec`, newly allocated in shared memory, if the two strings are different. Using the bit string, the threads compute the exclusive parallel prefix sum to calculate the total number of *unique* keys. In Step 2, thread T_{ij} sets `kr[TID(i,j)].valid = T(rue)`, if it has a pointer to the first appearance of a key or vice versa. Also, `kr[TID(i,j)].pointer` and `kr[TID(i,j)].length` are set to the beginning of a unique key and the number of the occurrences, i.e., word count, of the key, respectively.[5] Finally, each thread writes a pointer to a variable size key, e.g., a word, to the `keys` buffer and a value, e.g., word count, to the `values` buffer in global memory using the information in the temporary `kr` data structure in Figure 6. Thus, the final output is <dog, 1>, <inside, 1>, and <the, 2> in this example.

Grex also applies a variant of the boundary merge technique in the reduce phase to produce correct final <*key, value*> pairs in global memory, if one group of <*key, value*> pairs with the same key is divided over more than one thread blocks due to load balancing among the reducers. An example code snippet of boundary merge is given in Section 5.

4.2. Buffer Management

In Grex, a user can use buffers managed by the Grex runtime system for efficient memory access in the GPU. A buffer in Grex is a data container that can partially or entirely reside in global memory, texture cache, or shared memory depending on the size and type of a buffer and the availability of shared memory or texture cache space. Grex provides a number of buffering methods and user APIs to enhance the performance by leveraging the GPU memory hierarchy.

To use a buffer, a user needs to specify the access mode and size of a buffer. Grex provides two buffer types: 1) a *constant buffer* and 2) a *buffer*. A constant buffer is stored in the constant memory. Its size should be fixed and known at compile time as discussed before. We describe non-constant buffers in the remainder of this subsection. A (non-constant) buffer is either read-only or read-write. If the user specifies a buffer as read-only, Grex uses the texture cache to cache the buffer. As the texture cache is managed by the hardware, a user does not need to specify the size. Grex simply binds a read-only buffer to the texture cache. Thus, data are loaded from the texture cache to shared memory by the hardware, if the data are available in the texture cache. On the other hand, if a buffer is read-write, Grex directly loads data from global memory to shared memory instead of checking the texture cache first. In Grex, a user can also

[5] In the reduce phase, the `length` field is overloaded to indicate the word count. However, Grex is not tied to this overloading. Depending on a specific application of interest, a temporary data structure different from `keyRecord` can be used.

Algorithm 2 Grex buffer management

The following steps are executed in the CPU:

1. Invoke `User::init()` function. In this function the user enables/disables buffers for the current phase and modifies their properties as needed.
2. Before starting to execute the first phase of a job, upload input data to global memory.
3. Traverse the list of buffers to compute the total size of the shared memory cache needed by the enabled buffers. Also, bind buffers marked as read-only to the texture cache.
4. Launch the next MapReduce kernel with the calculated total cache size.

The following steps are executed in the GPU:

5. Initialize the shared memory cache assigned to each buffer in parallel.
6. Run the user-specified phase kernel.
7. Write the result of the kernel execution temporarily stored in in shared memory, if any, to the corresponding read-write buffer in global memory. (To do this, each thread that has data to write to global memory independently computes a unique address as discussed in Section 4.1.)

specify a specific buffer to become active during a specific MapReduce phase via the `User::init()` callback function invoked and executed by Grex before processing each phase.

To process I bytes of input data, a user needs to specify the size of the buffer-/block, P, and the number of bytes processed by each GPU thread, $D(\geq 1)$. Thus, to process I bytes, Grex uses $B = \frac{I}{PD}$ blocks and $T = \frac{P}{D}$ threads/block. By default, the size of the shared memory used to cache a fraction of a buffer in each block is equal to the size of the per-block buffer in global memory. For example, if the size of a per-block buffer is 256 bytes, 256 bytes of shared memory in each thread block (if available) is used to cache the buffer by default. Grex also provides an option by which a user can specify a different amount of shared memory to be used to cache a buffer in each thread block.

In Grex, data are fetched to shared memory in sequence and the oldest data is replaced, if necessary, to accommodate new data, since a considerable fraction of MapReduce applications linearly scan input data to group and aggregate them in parallel. The overall workflow of the buffer manager in Grex is given in Algorithm 2. Grex buffer manager first calculates the total cache size needed by each thread block by traversing the user defined buffers. After computing the total size of the (shared memory) caches needed per thread block, Grex launches the kernel to execute a MapReduce phase in the GPU.

At the beginning of a phase, contiguous regions of shared memory in a thread block is assigned to the buffers active in the phase and initialized as specified by the user. Grex provides parallel cache initialization policies a user can choose for a cache: 1) The default policy reads data from global memory into the cache; 2) An alternative policy initializes every cache element to a constant value specified by the user; 3) The no initialization policy leaves the cache (e.g., a cache associated with an output buffer) uninitialized; and 4) The user value policy initializes each cache element to the return value of user defined `User::cache()` function that can be used to convert the input data. The selected policy is enforced by Step 5 of Algorithm 2.

After initializing caches, the kernel for the corresponding MapReduce phase is executed in Step 6 of Algorithm 2. After finishing the execution of a phase kernel, Grex writes the resulting data in shared memory back to global memory and deallocates shared memory. In this way, we aim to ensure that the result of executing a phase is globally available to the following MapReduce phases, if any. This approach allows the next MapReduce phase to fully utilize shared memory too. Since most MapReduce applications performs group-by and aggregation functions in parallel, the amount of results to be written back to global memory is not likely to be bigger than the original input data size. Thus, the overhead of write-back to global memory in Step 7 is expected to be tolerable. Moreover, Grex supports lazy emit to further decrease the shared memory consumption and the amount of data to write back to global memory.

In summary, Grex is designed to support the parallel split and load-balanced map and reduce phases, while efficiently utilizing shared memory and texture cache. In addition, Grex supports lazy emit and it is free of locks and atomics different from the other GPU-based MapReduce frameworks [7, 8, 11, 12, 19].

In the future, GPUs may support a unified cache model. However, even in a single cache model, it is important to carefully manage the cache. Our buffer management scheme may become simpler and more efficient in such an architecture, because it is unnecessary to use different caches based on read/write or read-only nature of a buffer. Also, if future GPUs allow a thread to directly read data cached by a different thread block, boundary merge can be performed not in the slow GPU DRAM but in the cache. Thus, it is possible for the performance of Grex to further improve in such an environment. In addition, the algorithmic approaches of Grex such as parallel split, even distribution of data to the mappers and reducers, and late emit will be still applicable regardless of the cache model.

5. Sample Grex Code

This section illustrate an example Grex code for word count to provide details on handling variable size data. The basic structure of word count is applicable to various text processing applications that require word or token extraction. A variable data application example is preferred since a fixed size data processing implementation is relatively trivial.

In Code 1, Grex calls the user specified `User::main()` function to create buffers and do system initialization. In this example, three (non-constant) buffers and one constant buffer are created. A user can use the `new_buffer <data_type> (per_block_size)` function call to create a buffer with a certain data type and size per thread block. Grex creates a buffer in global memory whose size is equal to the product of the per-block buffer size and the number of the thread blocks as discussed in Section 4. Also, it caches the buffer in shared memory of each thread block when the buffer is accessed at runtime. To create a constant buffer, a user has to use the `new_const_buffer <data_type> (total_size)` API as shown in the code. As a result of the Grex system call, one system-wide constant buffer of `total_size` is created in constant memory.

```
1  int Grex::User::main () {
2    // Create a per-block buffer that contains 256 bytes of input data.
3    input  = new_buffer<char>(256);
4    // There are at most 256/2 words from 256 bytes of data due to the
5    // null char at the end of a string.
6    keys   = new_buffer<char*>(128);
7    // Max no. of values is equal to max no. of words.
8    values  = new_buffer<int>(128);
9    // Create a constant buffer to hold the upper-case to lower-case
10   // translation table
11   tr_map  = new_cbuffer<char>(tr_table, 256);
12   // Store boundary key sizes for merging. There can be at most 2
13   // boundary words in a thread block.
14   keysize = new_buffer<char>(2);
15   ...
16 }
```

Code Snippet 1: Buffer setup

When Grex completes executing the main function, it reads the input data from disk as shown in Code 2 before it starts running a MapReduce job in the GPU. Data read from the input file are downloaded to global memory over the PCIe bus that connects the host and GPU. The data is read and the job is iteratively executed until the `read_input()` function returns the End of File (EOF). In this way, the GPU memory can be allocated and the job can be executed in an incremental manner. Grex also overlaps the I/O across the PCIe bus with the computation in the GPU, if any, to hide the I/O overhead.

```
1 size_t Grex::User::data(
2   Buffer *b, char *d, off_t offset, size_t size){
3   if(buffer->id()==input->id()) {
4     input_file.seek(offset);
5     return input_file.read(d, size);
6   }
7   return 0;
8 }
```

Code Snippet 2: Reading Input Data

Code 3 shows an example user initialization routine for the word count application. Grex invokes the `User::init()` function before starting to execute each phase kernel. In the function, the user needs to declare buffers that will be

used in a specific phase and set the type of each buffer as either a read-write or read-only buffer. A buffer becomes usable once it is enabled and remains valid until it is explicitly disabled by the user.

When a buffer is enabled, it is set as a read-write buffer by default. A user needs to explicitly set a buffer as read only via the `read_only()` member function as shown in Code 3. A read-only buffer is cached in texture memory as discussed before.

Notably, there is no case statement for parallel split or boundary merge in Code 3, because the execution of the parallel-split and boundary merge phases plus the buffer management in those phases are handled by Grex. A user only has to define the boundary merge function (e.g., Code 5). Also, in Code 3, the `values` buffer declared in Code 1 is enabled in the reduce phase, because lazy emit is applied in our word count example.

```
1  int Grex::User::init(Job_t &j, Task_t& t) {
2    switch ( t.phase ) {
3      case MAP:
4        // Do not initialize the cache for keys
5        // Initialize the cache for keysize as zeros
6        buffers.enable(input, keys, keysize);
7        keys.cacheFunc(CACHE_NO_INIT);
8        keysize.cacheFunc(CACHE_VALUE, 0);
9        break;
10     case GROUP:
11       // Specify keys as the output buffer
12       // Declare input as read-only (texture cache)
13       set_targets(keys);
14       input.writeEnable(false);
15       break;
16     case REDUCE:
17       // Specify keys as the output buffer
18       // Enable values
19       set_targets(keys);
20       buffers.enable(values);
21       break;
22   }
23   return 1;
24 }
```

Code Snippet 3: User initialization

```
1  __device__ void Grex::User::map() {
2    // Initialize active buffers
3    IBUFFERS(input, keys, keysize);
4    input[workerId] = tr(input[workerId]);
5    // Parallel split of the input buffer into words
6    KeyRecord kr  = input.split(isWordChar());
7    // Emit words found in the input
8    keys.emit(kr.key, kr.valid);
9    // Emit key size only for boundary keys
10   keysize.emit(kr.size, input.is_boundary(kr));
11 }
```

Code Snippet 4: User map phase function

In Code 4, the map phase is implemented as a device function executed in the GPU. (Code 4 − Code 8 are all device functions.) In Code 4, the `tr()` function is executed by the threads to convert the input data to lower-case by referring to the translation table stored in the constant memory, while removing non-word characters in parallel. In Code 4, the parallel split is invoked via the `input.split()` function provided by Grex. The pointers to the keys found as a result of executing the parallel split step is written to global memory by the `keys.emit()` function. In Code 4, only the keys are generated due to lazy emit.

```
1  __device__ void Grex::User::mmerge() {
2    IBUFFERS(keys, keysize);
3    // Read two keys on the boundary of two adjacent thread blocks
4    char *k1 = keys.last_of_blk(workerId), *k2 = keys.first_of_blk(workerId+1);
5    int size1 = keysize.last_of_blk(workerId);
6
7    // If one word is spread across two adjacent blocks, merge parts
8    // into one key. This is the case if k1 ends where k2 begins
9    if ( k1 + size1 == k2 ) // Just delete the 2nd key to merge
10     keys.del_first_key(workerId+1);
11 }
```

Code Snippet 5: User function for boundary merge at the end of the map phase

The boundary merge function is shown in Code 5. The function is executed in parallel using $B-1$ threads where B is the total number of the thread blocks. Note that deleting the first key of the second block is enough for a thread to merge the pieces of a key divided between the adjacent thread blocks, because the last key in the first block already points to the beginning of the merged word.

```
1  __device__ void Grex::User::reduce(){
2    IBUFFERS(keys, values);
3    // Parallel split of keys into two groups at points where two
4    // consecutive keys are different
5    ReduceRecord rr = keys.split(CompareKeys());
6    keys.emit(rr.key, rr.valid);
7    values.emit(rr.size, rr.valid);
8  }
```

Code Snippet 6: User reduce function

The reduce function in Code 6 splits the keys into groups and combines each group with the same key into a single <*key, value*> pair. A thread executes the user defined `compareKeys()` function to compare the key assigned to itself with the key assigned to the next thread. It returns 0 if the two keys are equal. Otherwise, it returns 1. Grex creates a bit string in shared memory, in which 1 indicates the beginning of a new group and contiguous 0's following 1 indicates a group, similar to the bit string in Figure 6. By applying the exclusive parallel prefix sum, Grex computes the size of each key group as discussed in Section 4. By doing this, the reduce function in Code 6 groups the keys and aggregates each group with the same key into a single <*key, value*> pair, e.g., <word, count> in word count. At the end of the reduce phase, actual values are generated and written to global memory.

Code 7 merges a key group divided between two adjacent thread blocks due to load balancing. If there are B thread blocks, $B-1$ threads are used for boundary merge at the end of the reduce phase. Using the user defined compare function (e.g., strcmp() in word count), a thread compares the last key of a thread block and the first key of the next block. If they are same, the frequency of the word is changed to the sum of the sizes of the first and second key groups. This parallel process is executed iteratively until no group size increases any further.

```
__device__ void Grex::User::rmerge(){
  IBUFFERS(input,values);
  char *k1=last_of_blk(workerId), *k2=first_of_blk(workerId+1);
  int  *v1=plast_of_blk(workerId), *v2=pfirst_of_blk(workerId+1);
  if (compare(k1, k2)){
    *v1+=*v2;
    return 1;
  }
  else return 0;
}
```

Code Snippet 7: User function for boundary merge after the reduce phase

```
void Grex::User::end(){
  compact(keys, values);
  char *b_data = input.download();
  char **b_keys = keys.download();
  int  *b_values  = values.download();
  display(b_keys, b_values);
}
```

Code Snippet 8: User end function

Finally, Grex invokes the `User::end()` function when all phases are executed. The example `User::end()` function given in Code 8 downloads the keys and values from the global memory in the GPU into the main memory in the host.

6. Performance Evaluation

In this section, the experimental settings for performance comparisons among Mars [7], MapCG [8], and Grex are described. Also, the performance evaluation results are discussed.

We pick Mars and MapCG for performance evaluation, because they represent the state-of-the-art in terms of GPU-based MapReduce. In addition, they are open source. Ji and Ma [11] have also developed a novel GPU-based MapReduce framework. It uses atomic operations to avoid two pass execution of a MapReduce job, similar to MapCG. However, [11] is not publicly available; therefore, we cannot compare the performance of Grex to that of [11].

We have tested the performance of Grex using different generations of GPUs for Mars and MapCG. The experiments for performance comparisons between Grex and Mars were run on the NVIDIA GeForce 8800 GTX GPU, which was used for performance evaluation by the developers of Mars [7]. The benchmarks

comparing Grex against MapCG were executed on the NVIDIA GeForce GTX 460 GPU that supports atomic operations needed by MapCG. Using different platforms for Mars and MapCG provides fair comparisons, since Mars' design was influenced by relatively restrictive programming support of older generation GPUs, in which atomic operations are not available and coalescing memory accesses has a significant impact on the performance. The host machine in both cases has the AMD Athlon II X4 630 CPU, 4GB RAM, and 500GB hard disk. Linux 2.6.32.21 is installed on the host.

6.1. Benchmarks

In this section, the benchmarks used for performance evaluation are described. Table 2 summarizes the characteristics of the micro-benchmarks. They are popular MapReduce applications and used for performance evaluation in the Mars [7] and MapCG work [8]. By using the same benchmarks, we intend to provide the fairness of our performance comparisons with Mars and MapCG. Similar workloads are used for performance evaluation in [11] and [12] too.

1. Similarity Score (SS): Similarity score is a matrix application. The input is a list of document feature vectors $\vec{v_i}$ that represent the characteristics of given documents. The output of this application is pairwise similarity scores of the input documents $\frac{v_i \cdot v_j}{|v_i| \cdot |v_j|}$. Since the size of an input data is fixed in SS, no parallel split phase is needed. Similarly, the size of groups in reduce phase is known and fixed. Thus, the input split and boundary merge phases are not needed as shown in Table 2. To compute the similarity score, we implement tiled matrix operations, in which tiles (i.e., sub-matrices) are manipulated independently in the map phase and the intermediate results are aggregated in the reduce phase. The divisors and the dividends are calculated in the map phase. In the reduce phase, we calculate the final values by doing the divisions.

2. Matrix Multiplication (MM): Since the input to MM is fixed sized, it does not require the input split and boundary merge phases as summarized in Table 2. We implement tiled matrix multiplication in the map phase and add the partial sums of the tiles in the reduce stage.

3. String Match (SM): Given an input string, the string match application finds the offset of the first character for each occurrence of the string in the input file. Our implementation in Grex uses a constant buffer to store the search string. Neither sequential nor parallel input split is needed, because a match can be found without tokenizing the input data. In the map phase, parallel string matching is performed. In Grex, boundary merge is performed to find any matching string divided between the adjacent thread blocks. This application does not require sorting or reduction, since the output buffer already contains individual pointers to the matched strings.

4. Word Count (WC): This benchmark counts the frequency of each word's occurrences in the input file. The final output is <*word, frequency*> pairs sorted

Name	Input	Split	Map	B. Merge	Reduce	L. Emit	Data Size
SS	fixed	no	yes	no	yes	no	512, 1024, 2048 documents
MM	fixed	no	yes	no	yes	no	512, 1024, 2048 elements
SM	variable	no	yes	yes	no	no	32, 64, 128 MB
WC	variable	yes	yes	yes	yes	yes	32, 64, 128 MB
II	variable	yes	yes	yes	no	no	32, 64, 128 MB
PC	variable	yes	yes	yes	yes	no	32, 64, 96 MB

Table 2: Properties of the benchmarks

in non-ascending order of frequencies. This benchmark is required to run every phase of MapReduce as shown in Table 2. Grex does parallel split and boundary merge. In addition, Grex uses the lazy emit option to further decrease the memory consumption and access delay by storing only the keys until the end of the reduce phase.

5. Inverted Index (II): The inverted index application reads a group of web documents and outputs every encountered link and the names of the web documents that include the link. Since the URL size is variable, this application requires the parallel split, map, boundary merge, and group phases for Grex. However, it needs no reduce phase.

6. Page View Count (PC): In the page view count benchmark, the input is a group of web logs in the form of (URL, IP address, cookie) tuples and the output is the number of distinct visits to each URL sorted in non-ascending order. The input to this benchmark is variable sized; therefore, this application requires the parallel split, map, boundary merge, and group phases. In addition, it needs the reduce phase as summarized in Table 2.

We have tested three different sets of inputs for each application: small, medium, and large, similar to Mars and MapCG. For Word Count and String Match applications, the input data is created by merging various e-books. The input for Page View Count application is created by randomly combining different web logs. The input for Inverted Index is generated using a simple crawler. The input for Matrix Multiplication and Similarity Score are randomly generated using the input generator provided by Mars [32].

6.2. Experimental Results

The performance evaluation results in terms of the speedup achieved by Grex over Mars and MapCG are given in Figures 8 and 9, respectively. We also show the throughput achieved by Grex for the large data sets in Table 2. A more detailed discussion of the performance results follows.

1. Similarity Score (SS): For this benchmark, Grex achieves up to approximately 2.4x speedup over Mars for the largest data set as shown in Figure 8. It also provides roughly 2.5x speedup over MapCG for the medium data set

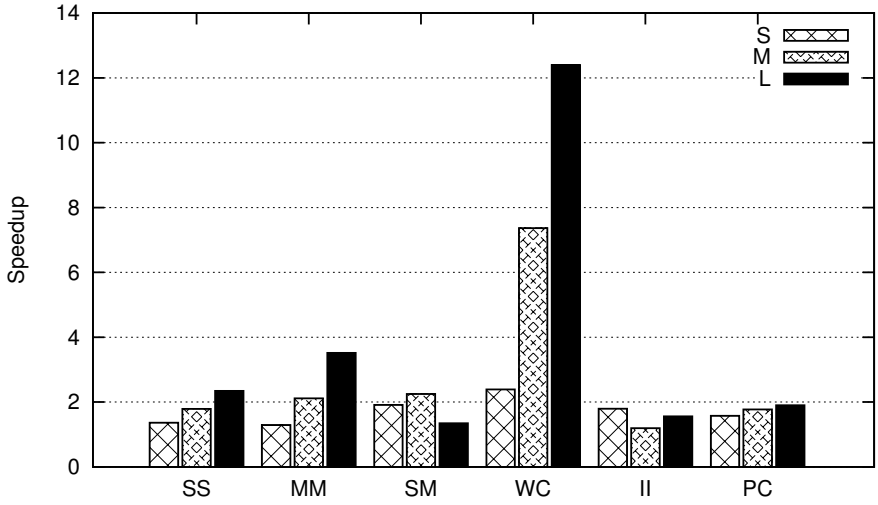

Figure 8: Speedup of Grex over Mars (Delay of Mars: 86ms - 10901ms)

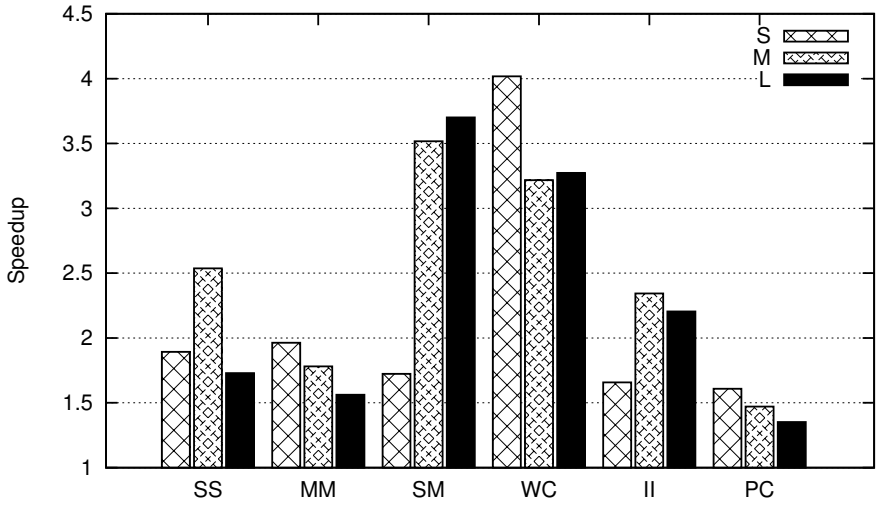

Figure 9: Speedup of Grex over MapCG (Delay of MapCG: 53ms - 2389ms)

as shown in Figure 9. The performance loss we observe for the large data set in Figure 9 is due to the increased input size that requires the reduce phase to combine more partial results. The observed speedups are due to better use of memory hierarchy. The throughput for the largest dataset with Grex is 56 GB/s from the task release to completion.

2. Matrix Multiplication (MM): For the largest data set, which requires to multiply two 2048 x 2048 matrices, Grex is 3.8x faster than Mars. Grex shows 2x speedup over MapCG for the small set; however, the speedup decreases down to 1.5x for the large data set, similar to SS. Although SS and MM do not need parallel split or lazy emit, Grex substantially enhances the performance by taking advantage of the GPU memory hierarchy and tiling technique. For the fairness of comparisons, tiling is used for the baselines as well as Grex. The throughput of Grex is 92 GB/s for the large dataset. Both SS and MM do matrix multiplication. However, SS requires vector multiplications and divisions, which are more computationally demanding than multiplying numbers and adding the partial results in MM. Hence, for SS and MM, Grex shows different performance in terms of the throughput and speedup over Mars and MapCG.

3. String Match (SM): Grex achieves maximum 2.2x speedup over Mars for the medium-size input and 3.7x speedup over MapCG for the largest data set. Grex achieves the performance improvement due to parallel split and efficient buffering using the shared memory and texture cache. The throughput of SM for the large dataset is 36 GB/s. By comparing the throughput achieved by Grex for SS, MM, and SM, we observe that processing variable size data is more challenging than dealing with fixed size data is. As a result, Grex supports significantly lower throughput than processing fixed size data.

4. Word Count (WC): Grex achieves up to 12.4x speedup over Mars. Also, it achieves up to 4x speedup over MapCG. Among the tested benchmark applications, Grex achieves the highest performance improvement for word count. This is because word count requires all the MapReduce phases and the amount of data to process can be decreased only after the reduce phase begins. Since we use English e-books for WC, data skews impair the performance of Mars and MapCG. In contrast, Grex evenly divides intermediate <*key, value*> pairs to the reduce tasks to avoid performance penalties due to skewed data distributions. Further, Grex applies the parallel split method, while applying lazy emit as well as caching to decrease the frequency of global memory access. The throughput for WC in Grex is 14 GB/s.

5. Inverted Index (II): This benchmark requires the map and grouping phases of MapReduce to invert links and group them together; however, it needs no reduce phase. Due to parallel split, even data partitioning among map/reduce tasks, and efficient buffering, Grex is up to 1.7x and 2.3x faster than Mars and MapCG, respectively. The measured throughput of II with Grex for the large dataset is 11 GB/s. II deals with variable sized data. In addition, it has no means to reduce the amount of data during the computation, because it only inverts links without aggregating them at all. Thus, Grex achieves the lower throughput than it did for WC.

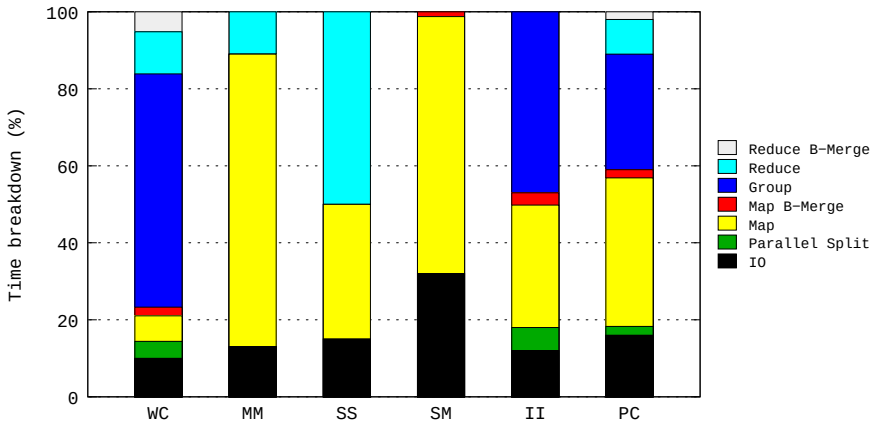

Figure 10: The time breakdown of Grex for the large data set

6. Page View Count (PC): For the large data set, Grex is 2x faster than Mars and 1.3x faster than MapCG. This benchmark needs all the MapReduce phases, similar to WC. However, Grex achieves relatively small performance gains over Mars and MapCG, because dealing with variable-size URLs requires more complex data processing than handling natural language words in WC. For the large dataset, the throughput of Grex is 8 GB/s. The throughput is lower than that of II, because PC is more compute-intensive than II is. In PC, intermediate <*key, value*> pairs have to be sorted and aggregated in the reduce phase.

Finally, Figure 10 shows the time breakdown of the Grex phases in terms of percentage. I/O for moving input and output data between the CPU and GPU takes approximately 10% of the total job execution delay on average with the only exception of SM. In SM, however, the percentage of time spent for I/O is exaggerated due to the artifact of a computationally simple map phase. Also, SM needs neither a grouping phase nor a reduce phase. Although both MM and SS are matrix benchmarks, the relatively complex reduce phase of SS takes about 50% of the job execution time. Notably, textual data applications that need the grouping phase, i.e., word count, inverted index, and page view count, spend approximately $30\% - 60\%$ of the job execution time for sorting. However, parallel split and boundary merge phases together take less then 10% of total execution time. From these results, we claim that the overhead of parallel split and boundary merge is acceptable considering the fact that (either sequential or parallel) input split is required to process variable size data. Moreover, we observe that further performance gains can be yielded by accelerating the sort phase, while decreasing the I/O overhead. A thorough investigation is reserved for future work.

Overall, Grex has achieved substantial performance improvements compared to Mars and MapCG, because it avoids executing the map and reduce phases twice and effectively utilizes the GPU memory hierarchy, while supporting par-

allel split and even load distribution among map/reduce tasks. Also, Grex outperforms MapCG by avoiding serialized memory management and by supporting parallel split, even data partitioning among map/reduce workers, lazy emit, and efficient memory management.

7. Conclusion and Future Work

In this paper, we present a new GPU-based MapReduce framework called Grex. We exploit the GPU hardware parallelism and memory hierarchy to significantly enhance the performance especially in terms of the delay. We strive to avoid creating possible performance bottlenecks or load imbalance due to inefficient data distribution or data skews. At the same time, we avoid using atomics and locks, which incur potentially high contention for memory access and complexities for conflict resolution. To achieve these goals, we have developed a number of new methods for GPU-based MapReduce summarized as follows:

- Grex supports parallel split that replaces the sequential split step provided by most existing MapReduce frameworks;
- Grex partitions data to map and reduce tasks in a parallel, load-balanced fashion;
- Grex supports efficient memory access by leveraging shared memory, texture cache, and constant memory; and
- It supports lazy emit to further decrease the frequency of global memory access and the resulting delay.

We have actually implemented and experimentally evaluated Grex. The results of the performance evaluation indicate up to 12.4x and 4.1x speedup over Mars [7] and MapCG [8] that represent the current state of the art in terms of GPU-based MapReduce computing.

Given the promising initial results, we will incorporate Grex with Hadoop to efficiently process large amounts of data with minimal delays. We will also investigate the portability of Grex to AMD GPUs.

References

[1] J. Nickolls, I. Buck, M. Garland, K. Skadron, Scalable Parallel Programming with CUDA, ACM Queue 6 (2) (2008) 40–53.

[2] A. Munshi, OpenCL: Parallel Computing on the GPU and CPU, Presentation at Computer Graphics and Interactive Techniques (2008).

[3] J. Dean, S. Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, Communications of ACM 51 (2008) 107–113.

[4] Hadoop Project, http://hadoop.apache.org.

[5] J. Lin, The Curse of Zipf and Limits to Parallelization: A Look at the Stragglers Problem in MapReduce, in: Workshop on Large-Scale Distributed Systems for Information Retrieval, 2009.

[6] Y. C. Kwon, M. Balazinska, B. Howe, J. Rolia, A Study of Skew in MapReduce Applications, in: Open Cirrus Summit, 2011.

[7] B. He, W. Fang, Q. Luo, N. K. Govindaraju, T. Wang, Mars: a MapReduce Framework on Graphics Processors, in: International Conference on Parallel Architectures and Compilation Techniques, 2008.

[8] C. Hong, D. Chen, W. Chen, W. Zheng, H. Lin, MapCG: Writing Parallel Program Portable between CPU and GPU, in: International Conference on Parallel Architectures and Compilation Techniques, 2010.

[9] C. Ranger, R. Raghuraman, A. Penmetsa, G. R. Bradski, C. Kozyrakis, Evaluating MapReduce for Multi-core and Multiprocessor Systems, in: IEEE International Symposium on High Performance Computer Architecture, 2007.

[10] R. M. Yoo, A. Romano, C. Kozyrakis., Phoenix Rebirth: Scalable MapReduce on a Large-Scale Shared-Memory System, in: IEEE International Symposium on Workload Characterization, 2009.

[11] F. Ji, X. Ma, Using Shared Memory to Accelerate MapReduce on Graphics Processing Units, in: IEEE International Parallel and Distributed Processing Symposium, 2011.

[12] J. A. Stuart, J. D. Owens, Multi-GPU MapReduce on GPU Clusters, in: IEEE International Parallel and Distributed Processing Symposium, 2011.

[13] J. Talbot, R. M. Yoo, C. Kozyrakis, Phoenix++: Modular MapReduce for Shared-Memory Systems, in: International Workshop on MapReduce and its Applications, 2011.

[14] R. Chen, H. Chen, B. Zang, Tiled-MapReduce: Optimizing Resource Usages of Data-Parallel Applications on Multicore with Tiling, in: International Conference on Parallel Architectures and Compilation Techniques, 2010.

[15] S. Coleman, K. S. McKinley, Tile Size Selection Using Cache Organization and Data Layout, ACM Conference on Programming Language Design and Implementation 30 (1995) 279–290.

[16] M. de Kruijf, K. Sankaralingam, MapReduce for the Cell B.E. Architecture, Tech. rep., Department of Computer Sciences, The University of Wisconsin-Madison (2007).

[17] L. Chen, G. Agrawal, Optimizing mapreduce for gpus with effective shared memory usage, in: Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing, ACM, 2012.

[18] B. Catanzaro, N. Sundaram, K. Keutzer, A Map Reduce Framework for Programming Graphics Processors, in: In Workshop on Software Tools for MultiCore Systems, 2008.

[19] R. Farivar, A. Verma, E. M. Chan, R. Campbell, MITHRA: Multiple data Independent Tasks on a Heterogeneous Resource Architecture, in: IEEE Conference on Cluster Computing, 2009.

[20] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, J. Manferdelli, High Performance Discrete Fourier Transforms on Graphics Processors, in: International Conference for High Performance Computing, Networking, Storage and Analysis, 2008.

[21] J. Bolz, I. Farmer, E. Grinspun, P. Schröoder, Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid, ACM Transactions on Graphics 22 (2003) 917–924.

[22] R. Szerwinski, T. Güneysu, Exploiting the Power of GPUs for Asymmetric Cryptography, in: Workshop on Cryptographic Hardware and Embedded Systems, 2008.

[23] S. Manavski, G. Valle, CUDA Compatible GPU Cards as Efficient Hardware Accelerators for Smith-Waterman Sequence Alignment, BMC Bioinformatics 9 (2008) 1–9.

[24] Q. Qian, H. Che, R. Zhang, M. Xin, The Comparison of the Relative Entropy for Intrusion Detection on CPU and GPU, Australasian Conference on Information Systems (2010) 141–146.

[25] P. Bakkum, K. Skadron, Accelerating SQL Database Operations on a GPU with CUDA, in: Workshop on General-Purpose Computation on Graphics Processing Units, 2010.

[26] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, J. C. Phillips, GPU Computing, IEEE Proceedings 96 (5) (2008) 879–899.

[27] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, T. Purcell, A Survey of General-Purpose Computation on Graphics Hardware, Computer Graphics Forum 26 (2007) 80–113.

[28] A. Bayoumi, M. Chu, Y. Hanafy, P. Harrell, G. Refai-Ahmed, Scientific and Engineering Computing Using ATI Stream Technology, Computing in Science and Engineering 11 (2009) 92–97.

[29] NVIDIA, CUDA Programming Guide 3.2, NVIDIA, 2009.

[30] CUDA Zone, http://www.nvidia.com/object/cuda_home_new.html.

[31] GPU Gems 3 - Chapter 39. Parallel Prefix Sum (Scan) with CUDA, http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html.

[32] Mars: A MapReduce Framework on Graphics Processors, http://www.cse.ust.hk/gpuqp/Mars.html.