Mars: A MapReduce Framework on Graphics Processors

Bingsheng He, Wenbin Fang, Qiong Luo
Hong Kong Univ. of Sci. and Tech.
{saven,wenbin, luo}@cs.ust.hk

Naga K. Govindaraju
Microsoft Corp.
nagag@microsoft.com

Tuyong Wang
Sina Corp.
tuyong@staff.sina.com.cn

Abstract

We design and implement Mars, a MapReduce framework, on graphics processors (GPUs). MapReduce is a distributed programming framework originally proposed by Google for the ease of development of web search applications on a large number of commodity CPUs. Compared with CPUs, GPUs have an order of magnitude higher computation power and memory bandwidth, but are harder to program since their architectures are designed as a special-purpose co-processor and their programming interfaces are typically for graphics applications. As the first attempt to harness GPU's power for MapReduce, we developed Mars on an NVIDIA G80 GPU, which contains over one hundred processors, and evaluated it in comparison with Phoenix, the state-of-the-art MapReduce framework on multi-core CPUs. Mars hides the programming complexity of the GPU behind the simple and familiar MapReduce interface. It is up to 16 times faster than its CPU-based counterpart for six common web applications on a quad-core machine. Additionally, we proposed a MapReduce framework with co-processing between the GPU and the CPU for further performance improvement.

1 Introduction

Search engines and other web server applications routinely perform data processing tasks, for example, indexing web pages and counting access frequency. Due to the time criticalness of these tasks in day-to-day operations and the vast amount of data, high performance is essential for these tasks [6]. For instance, www.sina.com.cn deploys tens of high-end Dell servers just to serve hourly updates of web stats such as the top ten hottest web pages. Furthermore, the complexity of these tasks and the heterogeneity of available computing resources makes it desirable to provide a generic framework for developers to implement these tasks correctly, efficiently, and easily.

The MapReduce framework is a successful model to support such data processing applications [6, 24]. It was originally proposed by Google for the ease of development of web search applications on a large number of machines. This framework provides two basic primitives (1) a map function to process input key/value pairs and to generate intermediate key/value pairs, and (2) a reduce function to merge all intermediate pairs associated with the same key. With this framework, developers can implement their application logic using these two primitives. The MapReduce runtime will automatically distribute and execute the task on multiple machines [6] or multiple processors in a single machine [20]. Thus, this framework reduces the complexity of parallel programming so that the developer can easily exploit the parallelism in the underlying computing resources for complex tasks. Encouraged by the success of the CPU-based MapReduce frameworks, we develop Mars, a MapReduce framework on graphics processors, or GPUs.

GPUs can be regarded as massively parallel processors with 10x faster computation and 10x higher memory bandwidth than CPUs [1]. Moreover, GPU performance is improving at a rate higher than Moore's law for CPUs. As the programmability of the GPU improves, several GPGPU (General-Purpose computing on GPUs) languages, such as AMD CTM [2] and NVIDIA CUDA [17], are introduced so that developers can write GPU programs without the knowledge of the graphics rendering pipeline. Nevertheless, since GPUs are traditionally designed as special-purpose co-processors for gaming applications, their languages lack support for some basic programming constructs, e.g., variable-length data types, as well as more complex functions such as dynamic memory allocation and recursion. Additionally, GPU architectural details are highly vendor-specific and programmers have limited access to these details. All these factors make the GPU programming a difficult task in general and more so for complex tasks such as web data analysis. Therefore, we propose to develop a MapReduce framework on the GPU so that programmers can easily harness the GPU computation power for their data processing tasks.

Compared with CPUs, the hardware architecture of GPUs differs significantly. For instance, current GPUs have over one hundred SIMD (Single Instruction Multiple Data) processors whereas current multi-core CPUs offer a much smaller number of cores. Moreover, most GPUs do not support atomic operations or locks. Due to the architectural differences, we identify the following three challenges in implementing the MapReduce framework on the GPU. First, the synchronization overhead in the runtime system of the framework must be low so that the system can scale to hundreds of processors. Second, a fine-grained load balancing scheme is required on the GPU to exploit its massive thread paral-

lelism. Third, the core tasks of MapReduce programs, including string processing, file manipulation and concurrent reads and writes, are unconventional to GPUs and must be handled efficiently.

With these challenges in mind, we design and implement Mars, our MapReduce framework on the GPU. Mars provides a small set of APIs that are similar to those of CPU-based MapReduce. Since GPUs currently do not have built-in support for strings, we have developed an efficient string library for our APIs. Our runtime system utilizes a large number of GPU threads for Map or Reduce tasks, and automatically assigns each thread a small number of key/value pairs to work on. As a result, the massive thread parallelism on the GPU is well utilized and the load is balanced across all threads. Additionally, we have developed a lock-free scheme to avoid any conflict between concurrent writes. This scheme guarantees the correctness of parallel execution with little synchronization overhead.

We have implemented Mars on a PC with an NVIDIA GeForce 8800 GPU (G80) and an Intel quad-core CPU. To evaluate Mars, we pick six common tasks in web server applications including web search, web document processing, and web log analysis. These tasks are String Match (SM), Inverted Index (II), Similarity Score (SS), Matrix Multiplication (MM), Page View Count (PVC), and Page View Rank (PVR). We implemented these tasks using Mars as well as using Phoenix [20], the state-of-the-art MapReduce framework for multi-core CPUs. For these six tasks, the amount of source code written by the developer using Mars is comparable to that using Phoenix. Moreover, our results show that Mars achieves a 1.5-16 times performance improvement over Phoenix. Additionally, we demonstrate that adding co-processing between the CPU and the GPU can further improve performance over Mars.

Organization: The remainder of the paper is organized as follows. We give a brief overview of the GPU, prior work on GPGPU and MapReduce in Section 2. We present our design and implementation of Mars and MapReduce with co-processing between the CPU and the GPU in Section 3. In Section 4, we present our experimental results. Finally, we conclude in Section 5.

2 Preliminaries and Overview

In this section, we introduce the GPU architecture and discuss the related work on GPGPU, and review the MapReduce framework.

2.1 Graphics Processors (GPUs)

Due to the high computation power and rapidly increasing programmability, GPUs have become an attractive co-processor for general purpose computing [1]. For more details on the GPU and its programming techniques, we refer the reader to a recent book edited by Nguyen [15].

Following the previous work [12, 21], we model the GPU as a many-core processor, as shown in Figure 1. The GPU consists of many SIMD multi-processors, and supports thousands of concurrent threads. GPU threads have low context-switch and low creation time as compared to CPU threads. The threads on each multiprocessor are organized into *thread groups*. Threads within a thread group share the computation resources such as registers on a multiprocessor. A thread group is divided into multiple schedule units that are dynamically scheduled on the multiprocessor. We adopt the metric *occupancy* to measure the resource utilization on the GPU [16]. The occupancy is the ratio of active schedule units to the maximum number of schedule units supported on the GPU. A low occupancy indicates that the computation resources of the GPU are under-utilized.

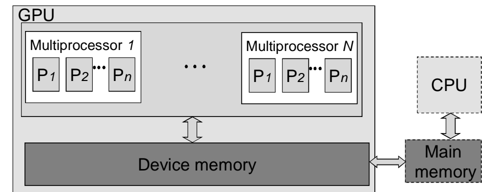

Figure 1. The many-core architecture of GPU.

The device memory of current GPUs is of both high bandwidth and high access latency. For example, the device memory of the NVIDIA G80 has a bandwidth of 86 GB/sec and latency of around 200 cycles. Due to the SIMD property, the GPU can apply *coalesced access* to exploit the spatial locality of memory accesses among threads: When the threads in a thread group access consecutive memory addresses, these memory accesses are grouped into one.

2.2 GPGPU

GPGPU has recently emerged in various applications, such as matrix operations [13], embedded system design [7], bioinformatics [14], databases [9, 12], and distributed computing projects including Folding@home [8] and Seti@home [22]. Previously, GPGPU developers used graphics APIs such as OpenGL [18] and DirectX [4] to map applications to the graphics rendering mechanism. They need the knowledge of graphics rending pipelines. Recently, several GPGPU languages including AMD CTM [2] and NVIDIA CUDA [17] have been proposed by GPU vendors. They usually expose a general-purpose, massively multi-threaded parallel computing architecture and provide a programming environment similar to multi-threaded C/C++. High-level programming frameworks such as Accelerator [23] are also developed to better facilitate GPGPU programming. Along this line, we further propose to develop a MapReduce framework on the GPU to ease the development of a more complex class of data processing tasks. For efficiency, we used CUDA to implement our framework.

We now briefly survey the work that developed GPGPU primitives as building blocks for various applications. For

additional information on the state-of-the-art GPGPU techniques, we refer the reader to a recent survey by Owens et al. [19]. Govindaraju et al. [9] presented novel GPU-based algorithms for the bitonic sort. Sengupta et al. [21] proposed the segmented scan primitive. He et al. [11] proposed a multi-pass scheme to optimize the scatter and the gather operations. He et al. [12] further developed a small set of primitives such as prefix sum and split for relational databases. Additionally, CUDPP [10], a CUDA library of data parallel primitives, was released for GPGPU computing. These GPU-based primitives reduce the complexity of GPU programming. However, even with the primitives, developers need to write complex GPU code for data processing tasks such as calculating similarity scores. In comparison, our work further simplifies GPU programming for MapReduce programmers by providing them a higher level and more familiar interface than the primitives.

2.3 MapReduce

The MapReduce framework [6] is based on two primitives, Map and Reduce, from functional programming. They are defined as follows:

Map: $(k_1, v_1) \rightarrow (k_2, v_2)*$.

Reduce: $(k_2, v_2*) \rightarrow v_3*$.

The Map function takes an input key/value pair (k_1,v_1) and outputs a list of intermediate key/value pairs (k_2,v_2). The Reduce function takes all values associated with the same key and produces a list of output values. Programmers implement the application logic using these two primitives. The parallel execution of each primitive is managed by the system runtime.

The following pseudo code illustrates a program written using MapReduce. This program counts the number of occurrences of each word in a collection of documents [6]. In this program, *Map* and *Reduce* are implemented using two other system-provided APIs, *EmitIntermediate* and *Emit*, respectively.

```
Map(void *doc) {
1:  for each word w in doc
2:      EmitIntermediate(w, 1);
}
Reduce(void* word, Iterator values) {
1:  int result = 0;
2:  for each v in values
3:      result += v;
4:  Emit(word, result); //output word and its count
}
```

MapReduce has been applied to various domains such as data mining, machine learning, and bioinformatics. Additionally, several implementations and extensions have been proposed. Hadoop [3] is an open-source MapReduce implementation similar to Google's. Phoenix [20] is an efficient implementation on multi-core CPUs. Chu et al. [5] applied MapReduce to ten machine learning algorithms on a multi-core CPU. Yang et al. [24] added the merge operation to MapReduce for relational databases. The existing work focuses on the parallelism of multiple CPUs or multiple cores within a single CPU. In contrast, our work is the first implementation of the MapReduce framework on the GPU. Since the GPU is a co-processor to the CPU, our framework complements these existing frameworks. Different from Phoenix, our GPU-based framework is lock-free, and is scalable to hundreds of processors on GPUs or future multi-core CPUs. Additionally, we handle co-processing between the CPU and the GPU.

3 Design and Implementation

In this section, we present our design and implementation for Mars. Our design is guided by the following two goals.

1) Ease of programming. Ease of programming encourages developers to use the GPU for their tasks.

2) Performance. The overall performance of our GPU-based MapReduce should be comparable to or better than that of the state-of-the-art CPU counterparts.

3.1 APIs

Mars provides a small set of APIs. Similar to the existing MapReduce frameworks, Mars has two kinds of APIs, the user-implemented APIs, which the users implement, and the system-provided APIs, which the users can use as library calls. Mars has the following user-defined APIs. These APIs are implemented with C/C++. We use the *void** type so that the developer can manipulate strings and other complex data types conveniently.

```
//MAP_COUNT counts result size of the map function.
void MAP_COUNT(void *key, void *val, int keySize, int valSize);
//The map function.
void MAP(void *key, void* val, int keySize, int valSize);
//REDUCE_COUNT counts result size of the reduce function.
void REDUCE_COUNT(void* key, void* vals, int keySize, int valCount);
//The reduce function.
void REDUCE(void* key, void* vals, int keySize, int valCount);
```

Mars has the following four system-provided APIs. The emit functions are used in user-implemented map and reduce functions to output the intermediate/final results.

```
//Emit the key size and the value size in MAP_COUNT.
void EMIT_INTERMEDIATE_COUNT(int keySize, int valSize);
//Emit an intermediate result in MAP.
void EMIT_INTERMEDIATE(void* key, void* val, int keySize, int valSize);
//Emit the key size and the value size in REDUCE_COUNT.
void EMIT_COUNT(int keySize, int valSize);
```

```
//Emit a final result in REDUCE.
void EMIT(void *key, void* val, int keySize, int valSize);
```

Overall, the APIs in Mars are similar to those in the existing MapReduce frameworks such as Hadoop [3] and Phoenix [20]. The major difference is that we need two APIs in Mars to implement the functionality of each CPU-based API. One is to count the size of results, and the other one is to output the results. This is because the GPU does not support atomic operations, and the Mars runtime uses a two-step design for the result output. We describe this two-step design in Section 3.3.

The following pseudo code illustrates the map for implementing the word count application using Mars. Instead of processing a document, each map task on the GPU processes one line in a document. For each word, the MAP_COUNT function emits a pair of the length of the word string and the size of an integer. Correspondingly, the MAP function emits the intermediate results of the word and the count of one.

```
//key is a line in the document.
//val is the line ID.
MAP_COUNT(key, val, keySize, valSize){
1:  for each word w in key
2:      EMIT_INTERMIDIATE_COUNT(w.length,
sizeof(int));
}
MAP (key, val, keySize, valSize) {
1:  for each word w in key
2:      EMIT_INTERMIDIATE (w, 1);
}
```

3.2 System Workflow and Configuration

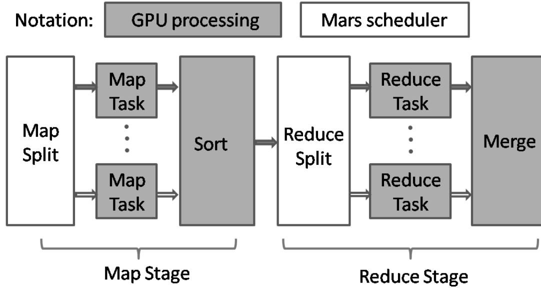

Figure 2. The work flow of Mars on the GPU.

Figure 2 illustrates the work flow of Mars. Similar to the CPU-based MapReduce framework, Mars has two stages, Map and Reduce. Before starting each stage, Mars initializes thread configuration including the number of thread groups and the number of threads per thread group on the GPU.

In the map stage, the split operator divides the input key/value pairs into multiple chunks such that the number of chunks is equal to the number of threads. A GPU thread is responsible for only one chunk. Thus, the threads in the map stage are load-balanced. After the map stage is finished, we sort the intermediate key/value pairs so that the pairs with the same key are stored consecutively.

In the reduce stage, the split operator divides the sorted intermediate key/value pairs into multiple chunks of similar sizes. The key/value pairs with the same key belong to the same chunk. The number of chunks is equal to the number of threads. The thread with a larger thread ID is responsible for a chunk with larger keys. This ensures that the output of the reduce stage is sorted by the key. The threads in the reduce stage are usually load-balanced unless there are skews in the lengths of the value lists. Dynamic load balancing scheme is not applicable, because current GPUs does not support dynamic thread scheduling. As the last step, the final output from all threads is stored into a single buffer.

Algorithm 1 describes the work flow of Mars in more detail. The operations in Lines 1–3 and 7 are done by the *scheduler*. The scheduler is responsible for preparing the data inputs, invoking the map and the reduce stages on the GPU and returning the results to the user.

Algorithm 1 Mars, the MapReduce framework on the GPU

1: Prepare input key/value pairs in the main memory and store them into input arrays.
2: Initialize the parameters in the run-time configuration (Table 1).
3: Copy the input arrays from the main memory to the GPU device memory.
4: Start the map stage on the GPU and store the intermediate key/value pairs into arrays.
5: If $noSort$ is F, sort the intermediate result.
6: If $noReduce$ is F, start the reduce stage on the GPU and generate the final results. Otherwise, the intermediate results are the final results.
7: Copy the final results from the GPU device memory to the main memory.

Compared with the existing CPU-based MapReduce frameworks [6, 20], Mars has two major simplifications due to the limitation of the GPU. First, Mars specifies the thread configuration and assigns the tasks to the GPU threads statically, because current GPUs do not support dynamic thread scheduling. Second, for efficiency, Mars relies on the CPU to preprocess raw input data into key/value pairs, because the GPU has no direct access to disk files.

Table 1. Configuration parameters of Mars.

Parameters	*Description*
$noReduce$	Whether a reduce stage is required (If it is required, $noReduce$=F; otherwise, $noReduce$=T).
$noSort$	Whether a sort stage is required (If it is required, $noSort$=F; otherwise, $noSort$=T).
$tgMap$	Number of thread groups in the map stage.
$tMap$	Number of threads per thread group in the map stage.
$tgReduce$	Number of thread groups in the reduce stage.
$tReduce$	Number of threads per thread group in the reduce stage.

Table 1 summarizes the configuration parameters of Mars. We provide these parameters to allow the programmers to tune the system for efficiency if they choose to. For instance, if the programmer knows that the application requires no Reduce, he/she can set $noReduce$ to true. All parameters have default values. The first two parameters are set to false by de-

fault. We discuss the default thread configuration in Section 3.4.

3.3 Implementation Details

Since the GPU does not support dynamic memory allocation on the device memory, we use arrays as the main data structure in Mars. The input data, the intermediate result and the final result are stored in three arrays, i.e., the key array, the value array and the *directory index*. The directory index consists of <key offset, key size, value offset, value size> for each input key/value pair. The key or the value at a certain index is accessed according to the offset and the length in the directory index.

With the array structure, we need to allocate the space on the device memory for the input data as well as the result output before executing the GPU program. However, the sizes of the output from the map and the reduce stages are unknown. Moreover, write conflicts occur when multiple threads write results to the shared output array, because most GPUs do not provide hardware-supported atomic operations or locks. To address these two problems, we adopt the previous lock-free output scheme of relational joins [12] and implement a two-step output scheme for the map and the reduce stages. Since the output scheme for the map stage is similar to that for the reduce stage, we present the scheme for the map stage only.

First, each map task outputs two counts, the total size of keys (in bytes) and the total size of values (in bytes) generated by the map task. Based on the counts of all map tasks, the runtime system computes a prefix sum on these counts and produces an array of write locations, each of which is the start location in the output array for the corresponding map to write. Through the prefix sum, we also know the array size for the intermediate result. Thus, the runtime allocates arrays in the device memory with the exact size for storing the intermediate results.

Second, each map task outputs the intermediate key/value pairs to the output array. The start write location for the map task is determined in the first step. Since each map has its deterministic and non-overlapping positions to write to, the write conflicts are avoided.

This two-step scheme does not require the hardware support of atomic functions. It is suitable for the massive thread parallelism on the GPU. However, it requires extra computation in counting the result size before output. Fortunately, with the GPU's high computation power, the extra computation poses little overhead.

3.4 Optimization Techniques

3.4.1 Memory Optimizations

We use two memory optimizations reducing the number of memory requests in order to improve the memory bandwidth utilization.

Coalesced accesses. We utilize the GPU feature of coalesced accesses to improve the memory performance. We design the memory accesses of each thread to the data arrays according to the coalesced access pattern when applicable. Suppose there are T threads in total and the number of key/value pairs is N in the map stage. Thread i processes the ($i + t \cdot k$)th (k=0,..,N/T) key/value pair. Due to the SIMD property of the GPU, the memory addresses from the threads in a thread group are consecutive and these accesses are coalesced into one. Figure 3 illustrates the map stage with and without the coalesced access optimization. In this example, three threads form a thread group. With the coalesced memory access, every three memory requests are coalesced into one. Thus, the coalesced access optimization reduces the number of memory requests by $2/3$.

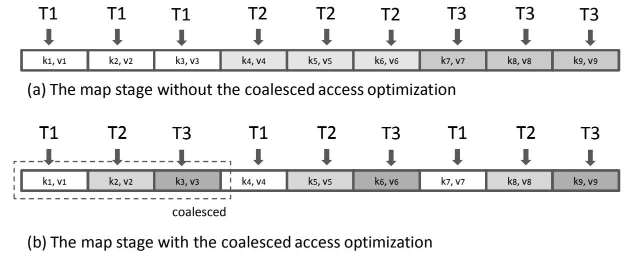

Figure 3. The map stage with and without coalesced access optimization.

Accesses using built-in vector types. Accessing the values in the device memory can be costly, because the data values are often of different sizes and the accesses are hardly coalesced. Fortunately, GPUs such as G80 support built-in vector types such as *char4* and *int4*. Reading built-in vectors fetches the entire vector in a single memory request. Compared with reading *char* or *int*, the number of memory requests is greatly reduced and the memory performance is improved.

3.4.2 Other Optimizations in Mars

Thread parallelism. The thread configuration, i.e., the number of thread groups and the number of threads per thread group, is related to multiple factors including, (1) the hardware configuration such as the number of multiprocessors and the on-chip computation resources such as the number of registers on each multiprocessor, (2) the computation characteristics of the map and the reduce tasks, e.g., they are memory- or computation-intensive.

Since the map and the reduce functions are implemented by the developer, and their costs are unknown to the runtime system, it is difficult to find the optimal setting for the thread configuration at run time. Fortunately, CUDA provides an off-line calculator [16] for computing the multiprocessor occupancy given a CUDA program. For the program (either the map task or the reduce task), the calculator takes the number of threads per thread group and the number of registers used per thread as input, and outputs the occupancy and the

number of active thread groups per multiprocessor. The number of registers used per thread is obtained using the NVCC compiler of CUDA.

With the calculator, we iterate the number of threads per group in multiples of 32 (the schedule unit size) ranging from 32 to 512 (the maximum number of threads per thread group), until the occupancy is higher than a predefined threshold. Thus, we get the number of threads per thread group and the number of thread groups. In practice, we set the occupancy threshold to be $2/3$ so that the GPU is sufficiently busy, and each thread group receives adequate computation resources.

Handling variable-sized types. The variable-sized types are supported with the directory index. If two key/value pairs need to be swapped, we swap their corresponding entries in the directory index without modifying the key and the value arrays. To further improve the GPU programmability, we develop a GPU-based string manipulation library. The operations in the library are consistent with those in C/C++ library on the CPU. Moreover, strings are accessed using *char4* to optimize the memory performance.

Hashing. Hashing is used in the sort algorithm to store the results with the same key value consecutively. In that case, we do not need the results with the key in the strict ascending/decreasing order. We use the hashing technique that hashes a key into a 32-bit integer, and we sort the records according to their hash values. When we compare two records, we first compare their hash values. Only when their hash values are the same, we need to fetch their keys and perform comparison on the key values. Given a good hash function, the probability of comparing the key values is low.

File manipulation. Currently, the GPU cannot directly access the data in the hard disk. Thus, we perform the file manipulation with the assistance of the CPU in three phases. First, we perform the file I/O on the CPU and load the file data into a buffer in the main memory. To reduce the I/O stall, we use multiple threads to perform the I/O task. Second, we perform the preprocessing on the buffered data and obtain the input key/value pairs. Finally, the input key/value pairs are copied from the main memory to the device memory.

3.5 Mars+: Co-Processing on CPUs and GPUs

Since both the CPU and the GPU are integrated components on a commodity machine, we develop MapReduce with co-processing on these two kinds of processors to fully utilize their computation power. We denote this MapReduce framework with co-processing as **Mars+**. We model a CPU and a GPU as two independent processors in Mars+. We partition the workload between the CPU and the GPU statically, because current GPUs do not support multitasking. We present the design of Mars+ on a GPU and a multi-core CPU, which is a typical configuration for the commodity machine. Our design can be easily extended to multiple GPUs and CPUs.

Mars+ consists of four main components, the CPU worker, the GPU worker, a code translator and the scheduler. The CPU and the GPU workers are responsible for processing the tasks on the CPU and the GPU, respectively. The code translator takes the APIs of the GPU worker as input and generates the APIs for the CPU worker. The scheduler partitions the tasks among the CPU and the GPU, and combines the results from the two processors.

The work flow of Mars+ is shown in Figure 4. Mars+ mainly have two stages, the map and the reduce. In the map stage, the scheduler divides the input data into multiple chunks. The number of chunks is equal to the total number of CPUs and GPUs in the machine. The chunk sizes are determined based on the performance comparison between the CPU and the GPU. Suppose the speedup of the GPU worker over the CPU worker is p, where the *speedup* is defined to be the ratio of the execution time on the CPU to that on the GPU for the same amount of input data. Given the total input size of I bytes, we assign data chunks of $\frac{pI}{1+p}$ and $\frac{I}{1+p}$ bytes to the GPU and the CPU workers, respectively.

During the map process, the runtime maintains the intermediate results. When a processor finishes a task and returns the intermediate result, the runtime merges the new intermediate result into the existing ones. When all the processors finish their tasks, the map stage ends.

The reduce stage takes the intermediate results from the map stage as input. Similar to the map stage, Mars+ statically assigns the data chunks to the processors. When all the processors finish their tasks, the reduce stage ends and Mars+ outputs the final results.

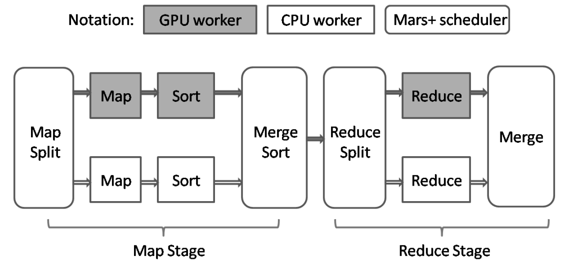

Figure 4. The work flow of Mars+.

Since we use Mars as the GPU worker, we now describe the design of the CPU worker in more detail. We design the CPU worker in a similar way to Mars other than adopting the existing implementation such as Phoenix [20]. The main reason is to preserve the programming simplicity of our MapReduce framework. Adopting Phoenix introduces the extra programming work of implementing Phoenix's APIs that are different from Mars'. In contrast, to use Mars+, the developer requires to implement the four user-implemented APIs for Mars. The code translator in Mars+ automatically generates the source code for the four user-implemented APIs in the CPU worker, because the four user-implemented APIs in the CPU worker is similar to those in the GPU worker. Thus, the developer requires little extra effort to utilize the co-processing capability of Mars+.

Another reason of adopting the design of Mars to the CPU worker is that the CPU worker can inherit the lock-free design of Mars and is scalable to hundreds of processors. Mars+ re-

quires little modification to run on the future multi-core CPUs with many cores.

Compared with the existing MapReduce, Mars+ exploits the chip-level parallelism within a processor (either a GPU or a multi-core CPU) and the inter-chip parallelism between the CPU and the GPU. Moreover, it preserves the programming simplicity of Mars.

4 Experimental Evaluation

In this section, we evaluate our GPU-based MapReduce framework in comparison with its CPU-based counterpart.

4.1 Experimental Setup

Our experiments were performed on a PC with a G80 GPU and an Intel Core2Duo Quad-Core processor running Linux Fedora 7.0. The hard drive is a 160G SATA magnetic hard disk. The GPU consists of 16 SIMD multi-processors, each of which has 8 processors running at 1.35GHz. In contrast, the CPU has four cores running at 2.4GHz. The GPU uses a PCI-E bus to transfer data between the main memory and the device memory with a theoretical bandwidth of 4 GB/sec. Based on our measurements, the device memory of G80 achieves a bandwidth of around 69.2 GB/sec and 34.5 GB/sec with and without coalesced access optimization, respectively, whereas the quad-core CPU has 5.6 GB/sec.

To evaluate the efficiency of our framework, we compared Mars with Phoenix [20], the state-of-the-art MapReduce framework on multi-core CPUs. Phoenix uses Pthreads to implement its runtime system. In our experiment, the number of cores used in Phoenix is set to four, i.e., the number of cores in the CPU.

For Mars, the sort algorithm was the bitonic sort on the GPU [12]. We used the prefix sum implementation from the CUDA library [17].

We run each experiment five times and report the average value.

Applications. We have implemented six applications for web data analysis as benchmarks for the MapReduce framework. They represent core computations for different kinds of web data analysis such as web document searching (String Match and Inverted Index), web document processing (Similarity Score and Matrix Multiplication) and web log analysis (Page View Count and Page View Rank). The first two and the fourth applications are those used in the experiments of Phoenix [20]. The third and the fourth applications are common tasks in the web search [25]. The last two are the routines for analyzing the statistics on the web page accesses in www.sina.com.cn.

Table 2 shows the description and the data sets used in our experiment for each application. We used three data sets for each application (S, M and L) to evaluate the scalability of the MapReduce framework. The values in the matrix or the vector are randomly generated float numbers. The web log entries are randomly generated. All these input data are stored as files in the hard disk.

Table 2. Application description

App.	Description	Data sets
String Match	Find the position of a string in a file.	S: 32MB, M: 64 MB, L: 128 MB
Inverted Index	Build inverted index for links in HTML files.	S: 16MB, M: 32 MB, L: 64MB
Similarity Score	Compute the pair-wise similarity score for a set of documents.	#doc: S: 512, M: 1024, L: 2048. #feature dimension: 128.
Matrix Multiplication	Multiply two matrices	#dimension: S: 512, M: 1024, L: 2048
Page View Count	Count the number of distinct page views from web logs.	S: 32MB, M: 64 MB, L: 96 MB
Page View Rank	Find the top-10 hot pages in the web log.	S: 32MB, M: 64 MB, L: 96 MB

We briefly describe how these applications are implemented using the MapReduce framework.

String Match (SM): Each Map searches one line in the input file to check whether the target string is in the line. Neither sort or the reduce stage is required.

Inverted Index (II): Each Map processes one line of HTML files. For each link it finds, it outputs an intermediate pair with the link as the key and the position as the value. Sort is required and no reduce stage is required.

Similarity Score (SS): It is used in web document clustering. The characteristics of a document are represented using a feature vector. Given two document features, $\vec{a}$ and $\vec{b}$, the similarity score between these two documents is defined to be $\frac{\vec{a} \cdot \vec{b}}{|\vec{a}| \cdot |\vec{b}|}$. SS computes the pair-wise similarity score for a set of documents. Each Map computes the similarity score for two documents. It outputs the intermediate pair of the score as the key and the pair of the two document IDs as the value. Sort is required and no reduce stage is required.

Matrix Multiplication (MM): Matrix multiplication is widely applicable to analyze the relationship of two documents. Given two matrices M and N, each Map computes multiplication for a row from M and a column from N. It outputs the pair of the row ID and the column ID as the key and the corresponding result as the value. Neither sort or the reduce stage is required.

Page View Count (PVC): A log entry is a 3-ary tuple $<URL, IP, Cookie>$, where URL is the URL of the accessed page; IP is the IP address that accesses the page; $Cookie$ is the cookie information generated when the page is accessed. This application has two executions of MapReduce. The first one removes the duplicate entries in the web logs. The second one counts the number of page views for each page. In the first MapReduce, each Map takes the pair of a log entry as the key and the size of the entry as value. The sort is to eliminate the redundancy in the web log. Specifically, if more than one log entries have the same information, we keep only one of them. The first MapReduce outputs the result pair of the log entry as key and the size of the line as value. The second MapReduce processes the key/value pairs generated from the first MapReduce. The Map outputs the

URL as the key and the IP as the value. The Reduce computes the number of IPs for each URL.

Page View Rank (PVR): It finds the top ten URLs that are most frequently accessed. The Map takes the pair of the page access count as the key and the URL as the value, which is also the output of PVC. Sort is required and no reduce stage is required.

In summary, these applications have different characteristics. MM and SS are more computation intensive than other applications. PVC has two MapReduces whereas other applications have one. PVC has the reduce stage, while others do not.

Finally, we show the size of the source code written by the developer using Mars and Phoenix in Table 3. The code size is measured in number of source code lines. Programming with Mars uses our own string manipulation library while programming with Phoenix uses the standard string library in C/C++. In general, the application with Mars has a similar code size as that with Phoenix. The Map and Reduce functions in Mars are simpler than those in Phoenix. Thus, the code size by the developer with Mars may be shorter than that with Phoenix.

Table 3. The size of the source code written by the developer using Mars and Phoenix.

	II	SM	SS	MM	PVC	PVR
Phoenix	365	250	196	317	292	166
Mars	375	173	258	235	276	152

4.2 Results on String Library

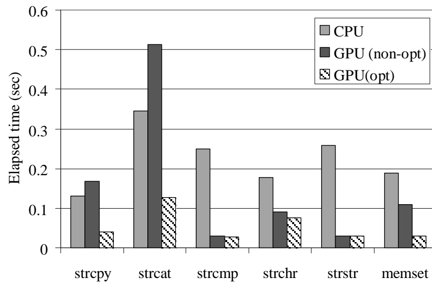

Figure 5. Performance comparison of the string libraries in C/C++ and Mars.

Figure 5 shows the performance comparison of the string libraries in C/C++ and Mars, denoted as "CPU" and "GPU", respectively. The number of string operations is eight million. The average string length is 16. The string operations on the CPU are performed using multiple concurrent threads to exploit the multi-core computation capability. The measurements on the GPU with accessing data using *char* and *char4* are denoted as "non-opt" and "opt", respectively. The optimized GPU implementation achieves 2-9x speedup over the CPU implementation.

Different string manipulations have different performance comparison between the CPU and the GPU. For the memory intensive operations such as *strcpy*, *strcat* and *memset*, the non-optimized GPU implementation can be slower than the CPU implementation. In contrast, the optimized GPU implementation is much faster than the CPU implementation with a speedup of 2.8-6.8x. For other three comparison-based operations, i.e., *strcmp*, *strchr* and *strstr*, the difference between the optimized and the non-optimized GPU implementation is small, because the memory optimization has little performance impact on these comparison-based operations.

4.3 Results on Mars

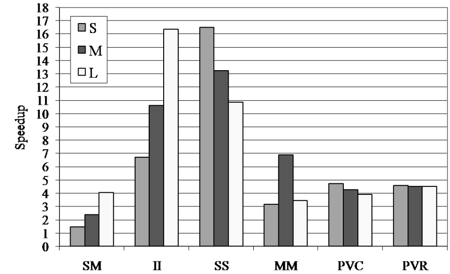

Figure 6. Performance speedup of Mars over Phoenix.

Figure 6 shows the performance speedup of the optimized Mars over Phoenix with the data set size varied. Overall, Mars is around 1.5-16x faster than Phoenix when the data set is large. The speedup varies for the applications with different computation characteristics. For computation-intensive applications such as SS and MM, Mars is over 4x faster than Phoenix. For other simpler applications such as SM, Mars is slightly faster than Phoenix.

We next investigate the time breakdown of each application. In Figure 7, we divide the total execution time of a GPU-based application into four components including the time for file I/O, the map stage excluding the sort, the sort after the map and the reduce stage. Note, the measurement of the map stage includes the time for copying the input data into the device memory, and the measurement of the Reduce includes the time for copying the result back to the main memory. The I/O time is dominant for SM and II, and the computation time is insignificant compared with the I/O time. Advanced I/O mechanisms such as using a disk array may greatly improve the overall performance for these two applications. In contrast, the total time of GPU processing including Map, Sort and Reduce is dominant for the other four applications. When the Sort step is required for the applications such as SS, PVC and PVR, the Sort time is a large part of the total execution time. Improving the sorting performance will greatly improve the overall performance of these applications.

In the following, we present the results for the performance impact of the hashing, the coalesced accesses and using built-in vector type. Since we obtain similar results for different data sizes, we present the results for the large data sets only.

We study the three optimization techniques in order. We first study the performance impact of the coalesced access

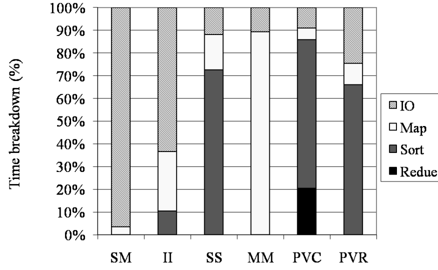

Figure 7. The time breakdown of Mars on the six applications with the large data set.

optimization, because it is orthogonal to the other two optimization techniques. The coalesced access optimization is mainly for the key array and the directory index, where tuples are of a fixed size. In contrast, the built-in vector type and hashing are for the value array with long or variable-sized tuples. Since hashing reduces the number of accesses to the value array, we evaluate the hashing technique before evaluating the built-in vector type optimization. Specifically, we evaluate the built-in vector type optimization at the presence of hashing and coalesced access to examine how much further improvement it makes.

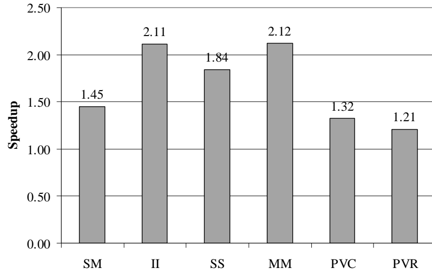

Figure 8. The performance speedup of coalesced accesses on the GPU.

Figure 8 shows the performance speedup of coalesced accesses on the six applications. We define the performance speedup of an optimization technique to be the ratio of the elapsed time without the optimization technique to that with the optimization technique. The measurements were obtained without using built-in vector types or hashing. The coalesced access improves the memory bandwidth utilization, which yields a performance speedup of 1.2-2.1x.

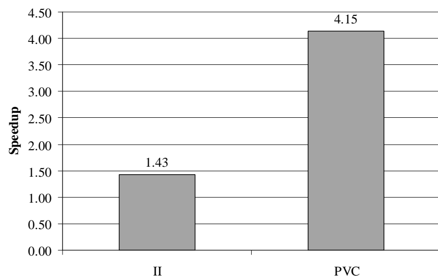

Figure 9. Performance speedup with hashing.

Figure 9 shows performance speedup of the hashing technique for II and PVC on the GPU, where hashing is applicable. We measured these numbers with coalesced access and without using built-in vector. The hashing technique improves the overall performance by 1.4-4.1x. The performance improvement of the hashing technique on PVC is larger than that on II, because PVC has two occurrences of MapReduce.

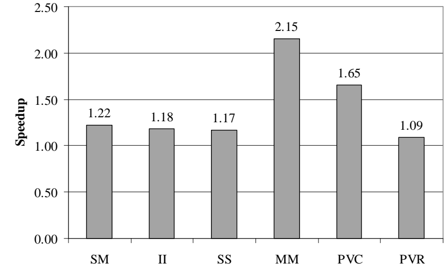

Figure 10. Performance speedup of accessing data with built-in vector types.

Figure 10 shows the performance speedup of accessing data with built-in vector types on the GPU. The measurement is with both hashing and coalesced access optimizations. Using built-in vector type reduces the number of memory requests and improves the bandwidth utilization. It improves the overall performance by 1.09-2.15x. The performance speedup depends on the size of the data accessed in a task. For instance, the performance speedup for MM and PVC is high, because each Map in MM and PVC requires fetching long integer vectors and a web log entry, respectively, and the built-in vector greatly helps. In contrast, the speedup for the other applications is small, because fetching data using the built-in vector type is not frequent.

4.4 Results on Mars+

Figure 11 shows the performance speedup of the CPU worker in Mars+ over Phoenix. The overall performance of our CPU worker is comparable to or better than that of Phoenix. There are two possible reasons for this performance speedup. One is that applications written using Phoenix always have a reduce stage whereas ours may not have. The other reason is that Phoenix has lock overhead whereas ours does not. Specifically for II, our CPU worker is 5.4x faster than Phoenix.

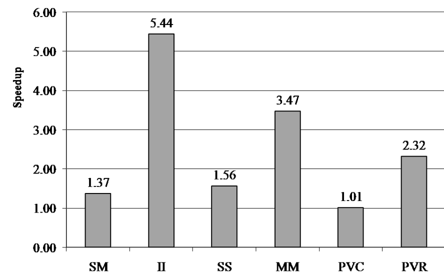

Figure 11. Performance speedup of the CPU worker in Mars+ over Phoenix.

Figure 12 shows the performance speedup of Mars+ over Mars. Overall, co-processing utilizes the computation power

of both the CPU and the GPU, and yields a considerable performance improvement over using the GPU only. The performance speedup of co-processing mainly depends on the performance comparison between the CPU processing and the GPU processing. For example, the performance improvement on the SS is small, because the CPU-based SS is much slower than the GPU-based SS. Thus, adding the CPU processing makes insignificant performance improvement on SS. In contrast, the co-processing scheme further reduces the execution time for other applications by partitioning the task between the CPU and the GPU.

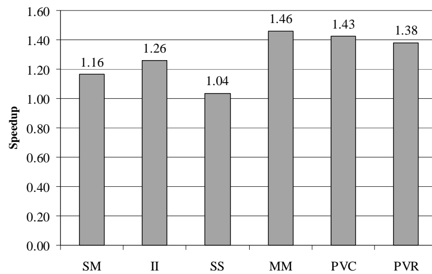

Figure 12. Performance speedup of Mars+ over Mars.

5 Conclusion

Graphics processors have emerged as a commodity platform for parallel computing. However, the developer requires the knowledge of the GPU architecture and much effort in developing GPU applications. Such difficulty is even more for complex and performance-centric tasks such as web data analysis. Since MapReduce has been successful in easing the development of web data analysis tasks, we propose a GPU-based MapReduce for these applications. With the GPU-based framework, the developer writes their code using the simple and familiar MapReduce interfaces. The runtime on the GPU is completely hidden from the developer by our framework. Moreover, our MapReduce framework yields up to 16 times performance improvement over the state-of-the-art CPU-based framework.

Finally, we plan to implement our MapReduce framework on AMD GPUs. We are also interested in integrating Mars into the existing MapReduce implementations such as Hadoop for the parallelism among different machines. The code and documentation of our framework can be found at http://www.cse.ust.hk/gpuqp/.

References

[1] A. Ailamaki, N. Govindaraju, S. Harizopoulos, and D. Manocha. Query co-processing on commodity processors. VLDB, 2006.

[2] AMD CTM. *http://ati.amd.com/products/streamprocessor/*, 2007.

[3] Apache Hadoop. *http://lucene.apache.org/hadoop/*, 2006.

[4] D. Blythe. The direct3d 10 system. SIGGRAPH, 2006.

[5] C. Chu, S. Kim, Y. Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. Neural Information Processing Systems, 2007.

[6] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. OSDI, 2004.

[7] J. Feng, S. Chakraborty, B. Schmidt, W. Liu, and U. D. Bordoloi. Fast schedulability analysis using commodity graphics hardware. 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 2007.

[8] Folding@home. *http://www.scei.co.jp/folding*, 2007.

[9] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. Gputerasort: high performance graphics coprocessor sorting for large database management. SIGMOD, 2006.

[10] M. Harris, J. Owens, S. Sengupta, Y. Zhang, and A. Davidson. Cudpp: Cuda data parallel primitives library. 2007.

[11] B. He, N. Govindaraju, Q. Luo, and B. Smith. Efficient gather and scatter operations on graphics processors. ACM/IEEE Supercomputing, 2007.

[12] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. V. Sander. Relational joins on graphics processors. SIGMOD, 2008.

[13] C. Jiang and M. Snir. Automatic tuning matrix multiplication on graphics hardware. PACT, 2005.

[14] W. Liu, B. Schmidt, G. Voss, and W. Wittig. Streaming algorithms for biological sequence alignment on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 18:1270–1281, 2007.

[15] H. Nguyen. *GPU gems 3*. Addison-Wesley, 2008.

[16] NVIDIA Corp. . *CUDA Occupancy Calculator*, 2007.

[17] NVIDIA CUDA. *http://developer.nvidia.com/object/cuda.html*, 2007.

[18] OpenGL. *http://www.opengl.org/*, 2007.

[19] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26, 2007.

[20] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. HPCA, 2007.

[21] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for gpu computing. ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, 2007.

[22] SETI@home. *http://setiathome.berkeley.edu/*, 2007.

[23] D. Tarditi, S. Puri, and J. Oglesby. Accelerator: using data parallelism to program gpus for general-purpose uses. ASPLOS, 2006.

[24] H. Yang, A. Dasdan, R. Hsiao, and D. S. Parker. Map-reduce-merge: Simplified relational data processing on large clusters. SIGMOD, 2007.

[25] R. Yates and B. Neto. *Modern information retrieval*. Addison Wesley, 1 edition, 1999.