Mars: A MapReduce Framework on Graphics Processors

Bingsheng He   Wenbin Fang   Naga K. Govindaraju[#]   Qiong Luo   Tuyong Wang[*]

Hong Kong Univ. of Sci. & Tech.   [#]*Microsoft Corp.*   [*]*Sina Corp.*

{saven, benfaung, luo}@cse.ust.hk   nagag@microsoft.com   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 CPUs. Compared with commodity 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 hundreds of processors, and evaluated it in comparison with Phoenix, the state-of-the-art MapReduce framework on multi-core processors. 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 integrated Mars with Phoenix to perform co-processing between the GPU and the CPU for further performance improvement.

1. Introduction

Search engines and other web server applications routinely perform analysis tasks on data they collect from the Web. Due to the time criticalness and the vast amount of the data, high performance is essential for these tasks [8]. For instance, www.sina.com.cn deploys tens of high-end Dell servers to compute web stats such as the top 10 hottest web pages from its web logs every hour. 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 analysis applications [8][27]. It was originally proposed by Google for the ease of development of web search applications on a large number of CPUs. 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 implement their application logic using these two primitives, and the data processing tasks are automatically distributed and performed on multiple machines [8] or a multi-core CPU [23]. Thus, this framework reduces the burden on the developers so that they can exploit the parallelism without worrying about the details of the underlying computing resources. Encouraged by the success of the CPU-based MapReduce frameworks, we develop Mars, a MapReduce framework on graphics processors, or GPUs.

GPUs have recently been utilized in various domains, including high-performance computing [22]. They can be regarded as massively parallel processors with 10x faster computation and 10x higher memory bandwidth than CPUs [1]. Several GPGPU (General-Purpose computing on GPUs) languages, such as AMD CTM [2] and NVIDIA CUDA [20], are introduced so that the developer can write code running on the GPU without the knowledge of the graphics APIs (e.g., DirectX [4] or OpenGL [21]). However, the developer needs to be familiar with the GPU architecture to fully exploit the computation power of the GPU. Unfortunately, GPU architectural details are highly vendor-specific and usually insufficient compared with those about CPUs. Moreover, the lack of high-level programming abstractions in many of the non-graphics APIs limits the scalability and portability of programs on new architectures. 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 study how to provide a MapReduce framework on the GPU so that the developer can easily write programs on the GPU to achieve high performance.

Compared with CPUs, the hardware architecture of GPUs differs significantly. For instance, current GPUs provide parallel lower-clocked execution capabilities on over a hundred SIMD (Single Instruction Multiple Data) processors whereas current CPUs offer out-of-

order execution capabilities on a much smaller number of cores. Moreover, GPU cache sizes are 10x smaller than CPU cache sizes. 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 run-time 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 parallelism. Third, the core tasks of MapReduce, 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. Specifically, we develop a small set of APIs (Application Programming Interfaces) to facilitate programmers to define their application logic. These APIs have a low synchronization overhead, and are flexible to support variable-sized data types such as strings. Since GPUs currently have no built-in support for strings, we have developed an efficient string library for our framework. In the runtime system, the tasks are uniformly assigned to the threads. Each thread is responsible for a Map or a Reduce task with a small number of key/value pairs as input. Since the number of threads on the GPU is much larger than that on the CPU, the workload among threads on the GPU is distributed at a finer grain than that on the CPU. Furthermore, we have developed a lock-free scheme to manage the concurrent writes among different threads in the MapReduce framework.

We have implemented our MapReduce framework on a machine with an NVIDIA GeForce 8800 GPU (G80) and an Intel quad-core CPU. We implemented six common computation tasks in web applications using our framework and Phoenix [23], the state-of-the-art MapReduce framework on multi-core CPUs. Our case studies cover web searching, web document processing and web log analysis. For these six applications, the size of the source code written by the developer based on Mars is comparable to that on Phoenix. Moreover, our results indicate that our optimizations greatly improve the overall performance of these applications, and Mars achieves a 1.5-16 times performance improvement over Phoenix. Additionally, we have integrated Mars with Phoenix to utilize both the CPU and the GPU for further performance improvement.

Organization: The rest of the paper is organized as follows. We give a brief overview of the GPU and prior work on the GPU and the MapReduce framework in Section 2. We present our design and implementation of Mars in Section 3. In Section 4, we evaluate Mars in comparison with Phoenix using the six web applications. Finally, we conclude in Section 5.

2. Preliminary and Related Work

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

2.1. Graphics Processors (GPUs)

GPUs are widely available as commodity components in modern machines. Their high computation power and rapidly increasing programmability make them an attractive platform for general-purpose computation. Currently, they are used as co-processors for the CPU [1]. The programming languages include graphics APIs such as OpenGL [21] and DirectX [4], and GPGPU languages such as NVIDIA CUDA [20], AMD CTM [2], Brook [5] and Accelerator [26]. With these APIs, programmers write two kinds of code, the kernel code and the host code. The kernel code is executed in parallel on the GPU. The host code running on the CPU controls the data transfer between the GPU and the main memory, and starts kernels on the GPU. Current co-processing frameworks provide the flexibility of executing the host code and other CPU processes simultaneously. It is desirable to schedule the tasks between the CPU and the GPU to fully exploit their computation power.
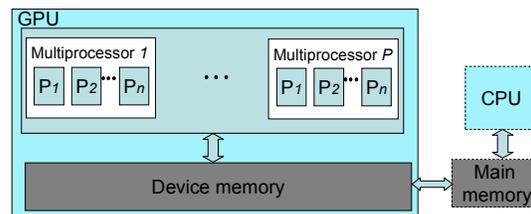

Figure 1. The GPU architecture model. This model is applicable to AMD's CTM [2] and NVIDIA's CUDA [20].

The GPU architecture model is illustrated in Figure 1. The GPU consists of many SIMD multi-processors and a large amount of device memory. At any given clock cycle, each processor of a multiprocessor executes the same instruction, but operates on different data. The device memory has 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. Moreover, the GPU device memory communicates with the main memory, and cannot directly perform the disk I/O.

The threads on each multiprocessor are organized into *thread groups*. The thread groups are dynamically scheduled on the multiprocessors. Threads within a

thread group share the computation resources such as registers on a multiprocessor. A thread group is divided into multiple schedule units. Given a kernel program, the *occupancy* of the GPU is the ratio of active schedule units to the maximum number of schedule units supported on the GPU. A higher occupancy indicates that the computation resources of the GPU are better utilized.

The GPU thread is different from the CPU thread. It has low context-switch and low creation time as compared to CPU's.

The GPU has a hardware feature called *coalesced access* to exploit the spatial locality of memory accesses among threads. When the addresses of the memory accesses of the multiple threads in a thread group are consecutive, these memory accesses are grouped into one.

2.2. GPGPU (General-Purpose computing on GPUs)

GPUs have been recently used for various applications such as matrix operations [18], FFT computation [17], embedded system design [10], bioinformatics [19] and database applications [13][14]. For additional information on the state-of-the-art GPGPU techniques, we refer the reader to a recent survey by Owens et al. [22]. Most of the existing work adapts or redesigns a specific algorithm on the GPU. In contrast, we focus on developing a generic framework for the ease of the GPU programming.

We now briefly survey the techniques of developing GPGPU primitives, which are building blocks for other applications. Govindaraju et al. [13] presented novel GPU-based algorithms for the bitonic sort. Sengupta et al. [24] proposed the segmented scan primitive, which is applied to the quick sort, the sparse matrix vector multiplication and so on. He et al. [15] proposed a multi-pass scheme to improve the scatter and the gather operations, and used the optimized primitives to implement the radix sort, hash searches and so on. He et al. [16] further developed a small set of primitives such as map and split for relational databases. These GPU-based primitives improve the programmability of the GPU and reduce the complexity of the GPU programming. However, even with the primitives, developers need to be familiar with the GPU architecture to write the code that is not covered by primitives, or to tune the primitives. For example, tuning the multi-pass scatter and gather requires the knowledge of the memory bandwidth of the GPU [15].

In addition to single-GPU techniques, GPUs are getting more involved in distributed computing projects such as Folding@home [11] and Seti@home [25]. These projects develop their data analysis tasks on large volume of scientific data such as protein using the computation power of multiple CPUs and GPUs. On the other hand, Fan et al. [9] developed a parallel flow simulation on a GPU cluster. Davis et al. [7] accelerated the force modeling and simulation using the GPU cluster. Goddeke et al. [12] explores the system integration and power consumption of a GPU cluster of over 160 nodes. As the tasks become more complicated, there lacks high-level programming abstractions to facilitate developing these data analysis applications on the GPU.

Different from the existing GPGPU computation, this study aims at developing a general framework that hides the GPU details from developers so that they can develop their applications correctly, efficiently and easily. Meanwhile, our framework provides the run-time mechanism for the high parallelism and the memory optimizations for a high performance.

2.3. MapReduce

The MapReduce framework is based on two functional programming primitives, which 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 applies on every input key/value pair (k1,v1) and outputs a list of intermediate key/value pairs (k2,v2). The reduce function applies to all intermediate values associated with the same intermediate key and produces a list of output values. Programmers define the application logic using these two primitives. The parallel execution of these two tasks is managed by the system runtime. The following pseudo code illustrates a program based on the MapReduce framework. This program counts the number of occurrences of each word in a collection of documents [8]. The programmer needs to define two APIs, `Map` and `Reduce`, using other two APIs provided in the MapReduce framework, `EmitIntermediate` and `Emit`, respectively.

```
// the input is a document.
// the intermediate output: key=word; value=1.
Map(void *input) {
for each word w in input;
    EmitIntermediate(w, 1);
}
// input: key=word, the value list consisting of 1s.
// output: key=word; value=occurrences.
Reduce(String key, Iterator values) {
int result = 0;
for each v in values
    result += v;
Emit(w, result);
}
```

Since the MapReduce framework was proposed by Google [8], it has been applied to various domains such as data mining, machine learning, and bioinformatics. Additionally, there have been several implementations and extensions of the MapReduce framework. Hadoop [3] is an open-source MapReduce implementation like google. Phoenix [23] is an efficient implementation on the multi-core CPU. Chu et al. [6] applied the MapReduce framework to ten machine learning algorithms on the multi-core CPU. Yang et al. [27] introduced the merge operation into the MapReduce framework for relational databases. The existing work focuses on the parallelism of multiple CPUs or multiple cores within a single CPU. To the best of our knowledge, this is the first implementation of the MapReduce framework on the GPU, which is a massively thread parallel processor. Since GPU is a commodity component on the modern machine, our framework complements these existing frameworks. Specifically, we have integrated our GPU-based framework with the CPU-based framework to utilize both the CPU and the GPU.

3. Design and Implementation

In this section, we present our design and implementation for Mars. Mars exploits the massive thread parallelism within the GPU. The programmer is required to define a small set of APIs that are similar to those in the CPU-based MapReduce framework. The rest of implementation details such as the GPU runtime are hidden by our framework.

3.1. Design Goals

Our design is guided by the following goals:

1) Ease of programming. Programming with graphics APIs requires a lot of effort from the developer to generate the correct code. In contrast, ease of programming encourages developers to use the GPU for computation processing. Additionally, our framework must support variable-sized types such as strings on the GPU, because processing of variable-sized types is common in the MapReduce programs such as the web applications.

2) Performance. Our goal is to develop a MapReduce framework on the GPU that achieves a performance that is comparable to or better than the state-of-the-art CPU counterparts.

3) Flexibility. Our framework must provide interfaces to allow developers to define their application logic, and to specify the runtime parameters for performance tuning.

4) Portability. Our goal is to present an intermediate runtime that maps the programs onto the underlying hardware. The programs written using the MapReduce framework require few or no changes.
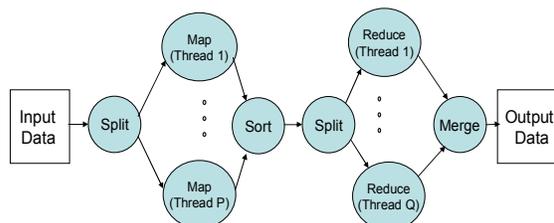
3.2. System Workflow and Configuration

Figure 2. The basic flow of Mars on the GPU. The framework consists of two stages, Map and Reduce.

Algorithm 1: **Mars, the MapReduce framework on the GPU.**

Parameters:	
dirInput	The pointer to the directory index <key offset, key size, value offset, value size> for each input key/value pair.
keyInput	The pointer to the data array for the input keys.
valInput	The pointer to the data array for the input values.
dirInter	The pointer to the directory index <key offset, key size, value offset, value size> for each intermediate key/value pair.
keyInter	The pointer to the data array for the intermediate keys.
valInter	The pointer to the data array for the intermediate values.
dirOutput	The pointer to the directory index <key offset, key size, value offset, value size> for each output key/value pair.
keyOutput	The pointer to the data array for the output keys.
valOutput	The pointer to the data array for the output values.

(1)	Prepare input key/value pairs in the main memory and generate three arrays, *dirInput*, *keyInput* and *valInput*.
(2)	Initialize the parameters in the run-time configuration.
(3)	Copy the three input arrays from the main memory to the GPU device memory.
(4)	Start the Map stage on the GPU and generate intermediate key/value results: *dirInter*, *keyInter*, and *valInter*.
(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 result: *dirOutput*, *keyOutput*, and *valOutput*. Otherwise, we copy *dirInter*, *keyInter*, and *valInter* to *dirOutput*, *keyOutput*, and *valOutput*, respectively.
(7)	Copy the result, *dirOutput*, *keyOutput*, and *valOutput*, from the GPU device memory to the main memory.

The basic flow of Mars is shown in Figure 2. It is designed based on our many-core architecture model. Algorithm 1 describes the flow in more detail.

Similar to the CPU-based MapReduce framework, Mars has two stages, Map and Reduce. In the Map stage, a split operator divides the input data into multiple chunks such that the number of chunks is equal to the number of threads. Thus, a GPU thread is responsible with only one chunk. The runtime parameters for the Map including the number of thread groups and the number of threads per thread group are determined according to the occupancy of the GPU. This thread configuration can also be specified by the programmer. We discuss the thread configuration in more detail in Section 3.4. 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 divides the sorted intermediate key/value pairs into multiple chunks. The number of chunks is equal to the number of threads. The thread configuration is set in a similar way to that for the Map. Each chunk is assigned to a GPU thread. Note that the key/value pairs with the same key are assigned to the same chunk. Additionally, the thread with a larger thread ID is assigned with a chunk consisting of key/value pairs of larger keys. This ensures that the output of the Reduce stage is sorted by the key.

Table 1. The configuration parameters of Mars.

Parameter	Description	Default
noReduce	Whether a reduce stage is required (If it is required, noReduce=F; otherwise, noReduce=T).	F
noSort	Whether a sort stage is required (If it is required, noSort=F; otherwise, noSort=T).	F
tgMap	Number of thread groups in the Map stage.	128
tMap	Number of thread per thread group in the Map stage.	128
tgReduce	Number of thread groups in the Reduce stage.	128
tReduce	Number of thread per thread group in the Reduce stage.	128

Table 1 summarizes the parameters and their default values in the configuration of Mars. These parameters are similar to those in the existing CPU-based MapReduce framework [23]. All these parameter values can be specified by the developer for performance tuning. Through exposing these parameters to the developer, Mars provides flexibility to specify whether the sort and the Reduce stage in the

framework are required, and to specify the thread configuration for the performance tuning.

The main data structure in Mars is array. Since dynamic memory allocation on the device memory is not supported on the GPU, we allocate the space on the device memory for the input data as well as the result output before executing the kernel program.

Since the size of the output from the Map and the Reduce are unknown, there are two challenges for buffer management in emitting the results from the Map and the Reduce. The first one is that the GPU lacks incremental memory allocation on the device memory during the kernel execution. The second one is that write conflicts occur when multiple threads write results to the shared output region in the device memory. Most GPUs do not provide hardware-supported atomic operations. To address these two challenges, we design a two-step output scheme. The first step computes the size of results generated by each thread. Based on these sizes, we can compute the write positions of each thread. The second step outputs the results to the device memory. Our scheme is similar to the previous scheme of result output of relational joins [16]. The major difference is that we need to handle the output for both the key and the value in the MapReduce.

Since the output scheme for the Map is similar to that for the Reduce, we present the scheme for the Map only.

First, each Map task outputs two counts, the total size of keys and the total size of values generated by the Map task. Based on these counts of all map tasks, we compute a prefix sum on these counts to get an array of write locations, each of which is the start location in the device memory for the corresponding map to write. Through the prefix sum, we also know the buffer size for the intermediate result.

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

3.3. APIs

Similar to the existing MapReduce frameworks such as Hadoop [3] and Phoenix [23], Mars provides a small set of APIs for the developers. These APIs are implemented with C/C++. Developers do not require any knowledge of graphics rendering APIs or the system runtime handling the massive threading parallelism in our MapReduce framework.

5

Table 2. The API functions in our MapReduce framework.

APIs of user-defined functions	Optional
`void MAP_COUNT(void *key, void *val, size_t keySize, size_t valSize)` Counts the size of output results (in bytes) generated by the map function.	no
`void MAP(void *key, void val, size_t keySize, size_t valSize)` The map function. Each map task executes this function on the key/value pair.	no
`void REDUCE_COUNT(void* key, void* vals, size_t keySize, size_t valCount)` Counts the size of output results (in bytes) generated by the reduce function.	yes
`void REDUCE(void* key, void* vals, size_t keySize, size_t valCount)` The reduce function. Each reduce task executes this function on the key/value pairs with the same key.	yes
`int compare(const void *d_a, int len_a, const void *d_b, int len_b)` User-defined function comparing two keys.	yes
APIs of functions provided by runtime	Optional
`void EMIT_INTERMEDIATE_COUNT(size_t keySize, size_t valSize);` Used in MAP_COUNT to emit the key size and the value size of an intermediate result.	yes
`void EMIT_INTERMEDIATE(void* key, void* val, size_t keySize, size_t valSize);` Used in MAP to emit an intermediate result.	yes
`void EMIT_COUNT(size_t keySize, size_t valSize);` Used in RECORD_COUNT to emit the key size and the value size of a final result.	yes
`void EMIT(void *key, void* val, size_t keySize, size_t valSize);` Used in REDUCE to emit a final result.	yes
`void GET_VAL(void *vals, int index);` Used in REDUCE to get a value of a given key in its value list.	yes
`Spec_t* GetDefaultConfiguration(char *logFileName)` Gets a default runtime configuration.	no
`void AddMapInputRecord(Configuration_t* conf, void* key,  void* val,  size_t keySize,  size_t  valSize);` Adds a key/value pair to the map input structure in the main memory.	no
`void MapReduce(Configuration_t * conf);` Starts the main MapReduce procedure.	no
`void FinishMapReduce();` Postprocessing such as logging when MapReduce is done.	no

Table 2 shows the complete list of our APIs. Similar to Phoenix [23], Mars has two kinds of APIs, APIs of user-defined functions and APIs of functions provided by the runtime. The developer calls the APIs of functions provided by the runtime to run the entire MapReduce framework on the input data. Additionally, the emit functions provided in Mars enable the programmer to output the intermediate/final results.

The main feature of Mars is simplicity, which shares the wisdom of the existing MapReduce frameworks. To use the framework, the developer needs to write code for the five APIs of user-defined functions at most. These APIs are similar to those in the CPU-based MapReduce framework. In each of these APIs, we use *void** type so that the developer can manipulate strings and other complex data types conveniently on the GPU. Additionally, the developer can define the application logic such as the key comparison in Mars.

The two-step design in the buffer management results in the difference in the APIs between the existing CPU-based MapReduce implementation and our GPU-based implementation. Existing CPU-based MapReduce implementations typically have four major APIs include `Map`, `Reduce`, `EmitIntermediate` and `Emit`. In contrast, we have defined two APIs on the GPU to implement the functionality of each of the four CPU-based APIs. One is to count the size of results. The other one is to output the results.

The following pseudo code illustrates the Map for implementing the word count application using Mars. For each word in the document, i.e., the key, the `MAP_COUNT` function emits a pair of the total length of the words and the size of the counts (an integer). In the `MAP` function, for each word, the function emits the intermediate results of the word and the count of one corresponding to the `MAP_COUNT`.

```
//the Map of implement word count using Mars
//the key is the document
MAP_COUNT(key, val, keySize, valSize){
  for each word w in key
      interKeySize+= sizeof(w);
      interValSize+= sizeof(int);
  EMIT_INTERMIDIATE_COUNT(interKeySize, interValSize);
}
MAP (key, val, keySize, valSize) {
  for each word w in key
      EMIT_INTERMIDIATE (w, 1);
}
```

3.4. Implementation Techniques

Thread parallelism. The number of thread groups and the number of threads per thread group is related to multiple co-related 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 task, e.g., it is memory- or computation-intensive. If a task is memory-intensive and suffers from memory stalls of fetching data from the device memory, a larger number of threads can be used to hide the memory stalls. Since the cost of the Map and the Reduce function is unknown, it is difficult to find the optimal setting for the thread configuration.

We set the number of threads per thread group and the number of thread groups based on the occupancy of the GPU. After the kernel code is compiled, we obtain the computation resources such as the number of registers that each thread requires for executing the kernel code. Based on this compilation information and the total computation resources on the GPU, we set the number of threads per thread group and the number of thread groups to achieve a high occupancy at run time. In practice, we set the number of thread groups to be 128, and the number of threads per thread group to 128, which is sufficient for a high occupancy.

Coalesced accesses. We utilize the hardware feature of coalesced accesses to improve the performance of the Map and the Reduce. Since the optimization to the Map is similar to that to the Reduce, we use the Map as an example. The directory index and the key/value pair are stored consecutively in arrays. Thus, we assign the key/value pairs to each thread according to the coalesced access pattern. 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 nature of the GPU, the memory addresses of the memory accesses from a thread group are consecutive and these accesses are coalesced into one. With the coalesced access, the memory bandwidth utilization is improved.

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

Handling variable-sized types. The variable-size types are supported in our MapReduce framework. The directory index for key/value pairs stores the indirection information, i.e., offsets and lengths. The value is fetched according to the offset and the length. If two tuples need to be swapped, we swap the two tuples in the directory index without modifying their values. To further improve the GPU programmability, we develop libraries for the common type processing in the MapReduce framework on the GPU. Specifically, we develop a string manipulation library on the GPU. The APIs in the library are consistent with those in C/C++ library on the CPU. Moreover, we access the string data using the built-in vector type to optimize the memory accesses in the string library.

Sort. We use bitonic sort [13] on the GPU. Sort is performed on the results of the Map. Nevertheless, we do not always need the results with the key in the strict ascending/decreasing order as long as the results with the same key value are stored consecutively. In that case, we can use the hashing technique that hashes the key into a 32-bit integer. 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 them. Given a good hash function, the probability of comparing the key values is low. Thus, the cost of the sorting algorithm can be reduced. This technique is implemented as an option in the user-defined compare function.

File manipulation. Currently, the GPU cannot directly fetch data from the hard disk to the device memory. 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 multi-threading to perform the I/O task. Second, we use the API, `AddMapInputRecord`, provided in Mars to prepare the input key/value pairs from the buffer in the main memory. Finally, the input key/value pairs are copied from the main memory to the device memory.

4. 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 is with a G80 GPU and a recently-released Intel Core2Duo Quad-Core processor running Linux Fedora 7.0. The hard drive is 160G SATA magnetic hard disk. The hardware configuration of the PC is shown in Table 3. The GPU uses a PCI-EXPRESS bus to transfer data

between the main memory and the device memory with a theoretical bandwidth of 4 GB/s. The GPU and the CPU have a theoretical bandwidth of 86.4 GB/s and 10.4 GB/s, respectively. Based on our measurements, for scans, the G80 achieves a memory bandwidth of around 69.2 GB/s whereas the quad-core CPU has 5.6 GB/s.

Table 3. Hardware configuration

	GPU	CPU
Processors	1350MHz $\times 8 \times 16$	2.4 GHz $\times 4$
Data cache (shared memory)	16KB $\times 16$	L1: 32KB $\times 4$, L2: 4096KB $\times 2$
Cache latency (cycle)	2	L1: 2 , L2: 8
DRAM (MB)	768	2048
DRAM latency (cycle)	200	300
Bus width (bit)	384	64
Memory clock (GHz)	1.8	1.3

To evaluate the efficiency of our framework, we compared our framework with Phoenix [23], the state-of-the-art MapReduce framework on the multi-core processor. Phoenix uses Pthreads to implement the runtime of the MapReduce framework on the multi-core processor. In our experiment, the number of cores used in Phoenix is set to four, i.e., the number of cores in our machine.

We implemented our GPU-based MapReduce using CUDA [20], which is a GPGPU programming framework for recent NVIDIA GPUs. It exposes a general-purpose, massively multi-threaded parallel computing architecture and provides a programming environment similar to multi-threaded C/C++. 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 [23]. The other applications such as data mining in the experiments of Phoenix are not used, because we focus on web data analysis applications. The third and the fourth ones are common tasks in the web search [28]. The last two are the daily routines for analyzing the statistics on the web page accesses in www.sina.com.cn.

Table 4 shows the description and the data sets used in our experiment for each application. We used three datasets for each application (S, M, and L) to test scalability issues on the MapReduce framework. The data set sizes are comparable to those in the experiments of Phoenix [23]. The web pages are crawled from the Web. The values in the matrix or the vector are randomly generated float numbers. The web log entries are randomly generated simulating the random accesses to the web pages. We varied other characteristics such as the number of URLs and the access frequency distribution and obtained similar results. All these input data are stored in the hard disk.

Table 4. 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 page view number of each URL in the web log.	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): String match is used as exact matching for a string in an input file. Each Map searches one line in the input file to check whether the target string is in the line. For each string it finds, it emits an intermediate pair of the string as the key and the position as the value. No Reduce stage is required.

Inverted Index (II): It scans a set of HTML files and extracts the positions for all links. 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. No Reduce stage is required.

Similarity Score (SS): It is used in web document clustering. The characteristics of a document are represented as a feature vector. Given two document features, $\vec{a}$ and $\vec{b}$, the similarity score between these two documents is defined to be $\frac{\vec{a} \cdot \vec{b}}{|\vec{a}||\vec{b}|}$. This application 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. 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. No Reduce stage is required.

Page View Count (PVC): It obtains the number of distinct page views from the web logs. Each entry in the web log is represented as *<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. In the first MapReduce, each Map takes the pair of an 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): With the output of the Page View Count, the Map in Page View Rank takes the pair of the page access count as the key and the URL as the value, and obtains the top ten URLs that are most frequently accessed. No Reduce stage is required.

In summary, these applications have different characteristics. As for the computation, MM and SS are more computation intensive than other applications. As for the number of occurrences of MapReduce, PVC has two 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 5. The code size is measured in number of source code lines. In general, the application with Mars has a similar code size as that with Phoenix. Programming with Mars uses our own string manipulation library while programming with Phoenix uses the standard string library in C/C++. The code in user-defined functions in Mars is simpler than that in Phoenix, because each Map or Reduce task in Mars handles fewer tuples than that in Phoenix. Thus, the code size by the developer with Mars can be smaller than that with Phoenix.

Table 5. The size of the source code in user-defined functions using Mars and Phoenix.

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

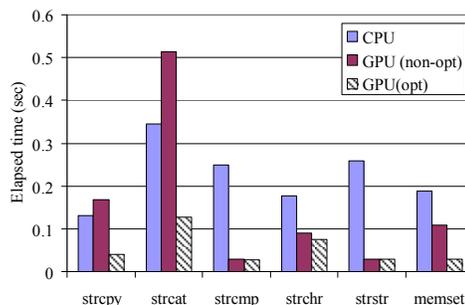## 4.2. Results on String Library

Figure 3. Performance comparison of the string libraries in C/C++ and Mars.

Figure 3 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 8 million. The average string length is 16. The string operations are performed using multiple threads on the CPU. The measurements on the GPU with accessing data using `char4` and `char` are denoted as "opt" and "non-opt", respectively. We define the *speedup* to be the ratio of the execution time on the CPU to that on the GPU. The optimized GPU implementation achieves 2-9X speedup over the CPU implementation.

String manipulations in the library have different performance comparison between the CPU and the GPU. For the memory intensive ones 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 APIs, i.e., `strcmp`, `strchr`, and `strstr`, the difference between the optimized and the un-optimized GPU implementation is small, because only part of the string is accessed.

4.3. Results on MapReduce

Figure 4 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 applications with different computation characteristics. For computation-intensive applications such as SS and MM, Mars is over 4X faster than Phoenix. For memory-intensive applications such as II and SM, Mars is slightly faster than Phoenix. These applications are simple in the computation. Thus, they achieve a smaller performance speedup using the GPU.
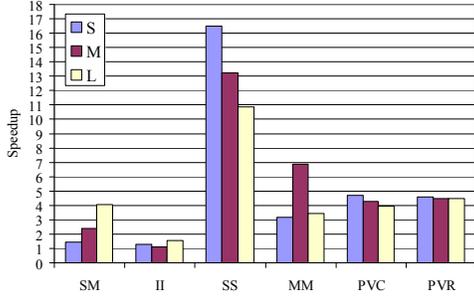
Figure 4. Performance speedup between Phoenix and the optimized Mars with the data size varied.

We next investigate the time breakdown of each application. We divide the total execution time of a GPU-based application into four components including the time for file I/O, the Map stage, 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 can greatly improve the overall performance for these two applications. In contrast, the total time of Map, Sort and Reduce is dormant 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 those applications.
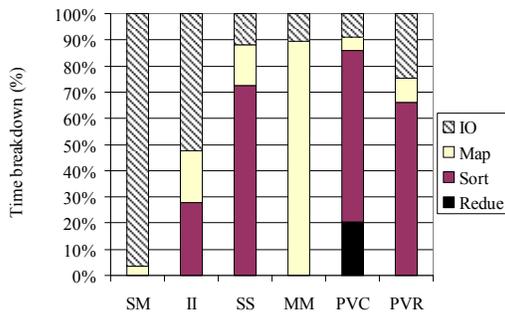

Figure 5. The time breakdown of Mars on the six applications with the large data set.

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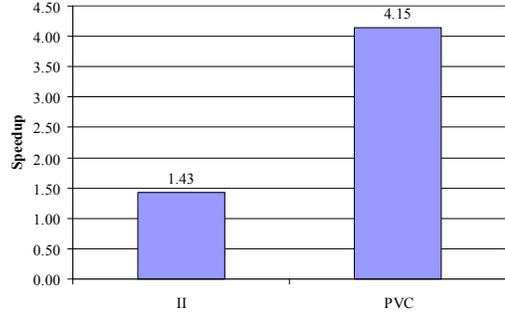

Figure 6. Performance speedup with hashing in Mars. The hashing technique improves the overall performance by 1.4-4.1X.

Figure 6 shows performance speedup of the hashing technique for II and PVC on the GPU. 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 results for the other four applications are now shown, because the hashing technique is not used in those applications. To separate the performance impact of the hashing technique from the other two optimization techniques, we measured these numbers without using built-in vector or coalesced access. 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 MapReduce and the hashing technique is used more frequently.
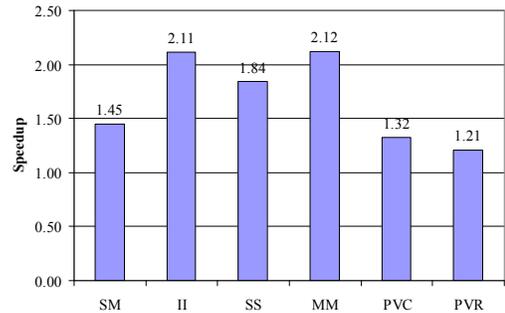

Figure 7. Performance speedup of coalesced accesses in Mars. The coalesced access improves the overall performance by 1.2-2.1X.

Figure 7 shows the performance speedup of coalesced accesses on the six applications. Note that these measurements are obtained with hashing but without using built-in vector. The coalesced access improves the memory bandwidth utilization, which yields a performance speedup of 1.2-2.1X.
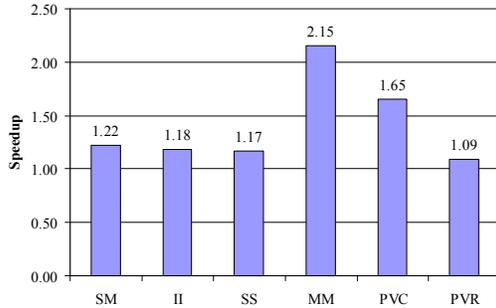
10

Figure 8. Performance speedup of accessing data with built-in vector types in Mars.

Figure 8 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 Map or Reduce task. For instance, the performance speedup for MM and PVC is high, because each Map in MM and PVC requires fetching long integer vectors or a web log entry, and the built-in vector greatly helps. In contrast, the speedup for the other applications is small, because fetching data using built-in vector type is not frequent. Note, the sort algorithm with the hashing technique has a low probability of fetching the original data. Thus, the performance impact of using built-in vector type is little for the applications such as PVR.

4.4. Results on Co-Processing on CPU and GPU

Figure 9 shows the performance speedup of co-processing on both the CPU and the GPU over processing on the GPU only. We used the large data set. The key/value pairs are assigned to Phoenix on the CPU and Mars on the GPU according to the speedup between Phoenix and Mars. Overall, the 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 processing of Phoenix is much slower than the GPU processing of Mars on SS. Thus, adding the CPU computation makes insignificant performance improvement on SS. In contrast, the co-processing scheme further reduces the execution time for other applications by scheduling the tasks among the CPU and the GPU.
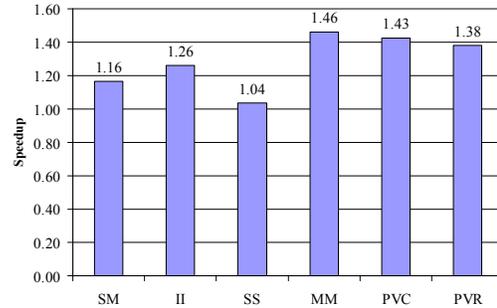

Figure 9. Performance speedup of co-processing on the CPU and the GPU over processing on the GPU only.

5. Conclusion

Graphics processors have emerged as a commodity platform for parallel computation. However, the developer requires the knowledge of the GPU architecture and much effort in tuning the performance. 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, it is an interesting future direction to extend our MapReduce framework to other application domains such as data mining. We are also interested in integrating Mars into the existing MapReduce implementations such as Hadoop so that the MapReduce framework can take the advantage of the parallelism among different machines as well as the parallelism within each machine. The code and documentation of our framework can be found at http://www.cse.ust.hk/gpuqp/.

6. Reference

[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/.

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

[4] D. Blythe. The Direct3D 10 system. SIGGRAPH 2006.

[5] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston and P. Hanrahan. Brook

for GPUs: Stream Computing on Graphics Hardware. SIGGRAPH, 2004.

[6] 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.

[7] D. Davis, R. Lucas, G. Wagenbreth and J. Tran and J. Moore. A GPU-enhanced cluster for accelerated FMS. High Performance Computing Modernization Program Users' Group Conference, 2007.

[8] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. OSDI, 2004.

[9] Z. Fan, F. Qiu, A. Kaufman and S. Y. Stover. GPU cluster for high performance computing. ACM/IEEE SuperComputing, 2004.

[10] J. Feng, S. Chakraborty, B. Schmidt, W. Liu, U. D. Bordoloi. Fast Schedulability Analysis Using Commodity Graphics Hardware. Proc. of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 2007.

[11] Folding@home, http://www.scei.co.jp/folding.

[12] D. Göddeke, R. Strzodka, J. M. Yusof, P. McCormick, S. Buijssen, M. Grajewski and S. Turek. Exploring weak scalability for FEM calculations on a GPU-enhanced cluster. Parallel Computing 33:10-11. pp. 685-699. 2007.

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

[14] A. Gress and G. Zachmann. GPU-ABiSort: Optimal Parallel Sorting on Stream Architectures. Proc. 20th IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS), 2006.

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

[16] B. He, K. Yang, R. Fang, M. Lu, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational joins on graphics processors. Technical Report HKUST-CS07-06, HKUST, Mar 2007.

[17] D. Horn. Lib GPU FFT, http://sourceforge.net/ projects/gpufft/. 2006.

[18] E. Larsen and D. McAllister. Fast matrix multiplies using graphics hardware. ACM/IEEE Supercomputing, 2001.

[19] W. Liu, B. Schmidt,G. Voss, and W. Wittig. Streaming algorithms for biological sequence alignment on GPUs. IEEE Transactions on Parallel and Distributed Systems, vol. 18, no. 9, pp. 1270-1281, Sept., 2007.

[20] NVIDIA CUDA (Compute Unified Device Architecture), http://developer.nvidia.com/object/cuda.html.

[21] OpenGL, http://www.opengl.org/.

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

[23] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, C. Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. Symposium on High Performance Computer Architecture (HPCA), 2007.

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

[25] SETI@home, http://setiathome.berkeley.edu/.

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

[27] H. Yang, A. Dasdan, R. Hsiao, and D. S. Parker. Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters. SIGMOD, 2007.

[28] R. Yates, B. Neto. Modern information retrieval. Addison Wesley, 1st edition, 1999.