Mars: Accelerating MapReduce with Graphics Processors

Wenbin Fang, *Student Member*, *IEEE*, Bingsheng He, Qiong Luo, and Naga K. Govindaraju, *Member*, *IEEE*

Abstract—We design and implement Mars, a MapReduce runtime system accelerated with graphics processing units (GPUs). MapReduce is a simple and flexible parallel programming paradigm originally proposed by Google, for the ease of large-scale data processing on thousands of CPUs. Compared with CPUs, GPUs have an order of magnitude higher computation power and memory bandwidth. However, GPUs are designed as special-purpose coprocessors and their programming interfaces are less familiar than those on the CPUs to MapReduce programmers. To harness GPUs' power for MapReduce, we developed Mars to run on NVIDIA GPUs, AMD GPUs as well as multicore CPUs. Furthermore, we integrated Mars into Hadoop, an open-source CPU-based MapReduce system. Mars hides the programming complexity of GPUs behind the simple and familiar MapReduce interface, and automatically manages task partitioning, data distribution, and parallelization on the processors. We have implemented six representative applications on Mars and evaluated their performance on PCs equipped with GPUs as well as multicore CPUs. The experimental results show that, the GPU-CPU coprocessing of Mars on an NVIDIA GTX280 GPU and an Intel quad-core CPU outperformed Phoenix, the state-of-the-art MapReduce on the multicore CPU with a speedup of up to 72 times and 24 times on average, depending on the applications. Additionally, integrating Mars into Hadoop enabled GPU acceleration for a network of PCs.

Index Terms—MapReduce, graphics processor, parallel computing, multicore processor, many-core architecture.

1 Introduction

MapReduce is a successful paradigm [15], originally proposed by Google, for the ease of distributed data processing on a large number of machines. In such a system, users specify two functions: 1) a *map* function to process an input key/value pair, and to generate a set of intermediate key/value pairs; 2) a *reduce* function to merge all intermediate key/value pairs associated with the same key. The system will automatically distribute and execute tasks on multiple machines [4], [15] or multiple CPUs in a single machine [32]. Thus, this paradigm reduces the programming complexity so that developers can easily exploit the parallelism in the underlying computing resources for complex tasks. Encouraged by the success of CPU-based MapReduce systems, in particular, Phoenix [32], we develop Mars, a MapReduce system accelerated with graphics processors, or GPUs.

GPUs can be regarded as massively parallel processors with an order of magnitude higher computation power (in terms of number of floating point operations per second) and memory bandwidth than CPUs [7]. Moreover, the computational performance of GPUs is improving at a rate higher than that of CPUs. However, it is a challenging task to program GPUs for general-purpose computing applications, including those that MapReduce users are familiar with. Specifically, GPUs are traditionally designed as special-purpose coprocessors for dedicated graphics rendering. As such, GPU cores are Single-Instruction-Multiple-Data (SIMD), which discourages complex control flows. Furthermore, GPU cores are virtualized, and threads are managed by the hardware. Finally, GPUs manage their own on-board device memory and require programmers to explicitly transfer data between the GPU memory and the main memory. Additionally, the architectural details of GPUs vary by vendors as well as by product releases, and programmer's access to these details is limited. All these factors make desirable a General Purpose Computation on GPUs (GPGPUs) framework on which users can develop correct and efficient GPU programs easily.

Recently, several GPGPU programming frameworks have been introduced, such as NVIDIA CUDA [5] and AMD Brook+ [1]. These frameworks significantly improve the programmability of GPUs; nevertheless, their interfaces are vendor-specific and their hardware abstractions may be unsuitable for complex applications, such as those running on MapReduce. Therefore, we propose Mars, a MapReduce framework to ease the programming of such applications on the GPU. Furthermore, the MapReduce framework of Mars enables the integration of GPU-accelerated code to distributed environment, like Hadoop, with the least effort. Our Mars system can run on multicore CPUs (MarsCPU), on CUDA-enabled NVIDIA GPUs (MarsCUDA) or Brook+-enabled AMD GPUs (MarsBrook), or on a combination of a multicore CPU and a GPU on a single machine. We further integrate Mars into Hadoop [4], an open-source CPU-based MapReduce system on a network of machines, which results in MarsHadoop, where each machine can utilize its GPU with MarsCUDA or MarsBrook in addition to its CPU with the original Hadoop. No matter what GPU and/or CPU Mars

• W. Fang is with the University of Wisconsin-Madison, Rm. 7373, The Computer Science Building, 1210 W. Dayton St., Madison, WI 53706-1685. E-mail: wenbin@cs.wisc.edu.
• B. He is with the Nanyang Technological University, N4-02C-115, School of Computer Engineering, 50 Nanyang Avenue, Singapore. E-mail: he.bingsheng@gmail.com.
• Q. Luo is with the Hong Kong University of Science and Technology, Room 3554, Academic Building, HKUST, Clear Water Bay, Kowloon, Hong Kong. E-mail: luo@cse.ust.hk.
• N.K. Govindaraju is with the Microsoft Corp., One Microsoft Way, Redmond, WA 98052. E-mail: nagag@microsoft.com.

Manuscript received 14 July 2010; accepted 19 July 2010; published online 23 Aug. 2010.
Recommended for acceptance by R. Bianchini.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2010-07-0423.
Digital Object Identifier no. 10.1109/TPDS.2010.158.

runs on, the Application Programming Interface (API) to the user is the same and is similar to that of existing CPU-based MapReduce systems.

Easing up GPU programming for MapReduce applications is the main goal of our work. However, a higher level abstraction for programming, specifically MapReduce, comes at a price of performance. In particular, we identify the following three technical challenges in implementing Mars on GPUs. First, since MapReduce divides up a task by data, load imbalance is an inherent problem in utilizing the massive thread parallelism on the GPU, especially because GPU threads are managed by the hardware. Second, GPUs lack efficient global synchronization mechanisms. Threads in Map or Reduce tasks are likely to have write conflicts on the output buffer. While atomic operations are enabled in recent GPUs, the overhead of atomic operations would harm the scalability of massive GPU threads [2]. We consider a lock-free scheme to minimize the synchronization overhead among GPU threads. Third, MapReduce applications are, in general, data intensive and their result sizes are data dependent. These two characteristics pose the following requirements on programming the GPU: 1) sufficient thread parallelism to hide the high latency and to utilize the high bandwidth of the device memory; and 2) preallocation of output buffers in the device memory for bulk DMA transfers, as GPU memory allocation is done through the CPU before the GPU program starts.

With these challenges in mind, we develop Mars for GPUs of two most common programming interfaces—CUDA and Brook+. We focus on MarsCUDA, rather than MarsBrook, because in our implementation and evaluation, CUDA was more flexible and had a higher performance than Brook+ for MapReduce applications.

In MarsCUDA, the massive thread parallelism on the GPU is well utilized as each thread is automatically assigned a key/value pair to work on. In *Map*, the system evenly distributes key/value pairs to each thread. In *Reduce*, we develop a simple but effective skew handling scheme to redistribute data evenly across all reduce tasks. To avoid write conflicts between threads, we adopt a lock-free scheme that guarantees the correctness of parallel execution with little synchronization overhead. Finally, we utilize the GPU-accelerated prefix sum primitive to compute output buffer sizes.

We evaluated the performance of Mars in comparison with its CPU-based counterparts and the native implementation without MapReduce. Our results demonstrate the effectiveness of our GPU-oriented optimization strategies. On average, our MarsCUDA is 22 times faster than the CPU-based MapReduce, Phoenix [32], and is less than three times slower than the hand-tuned native CUDA implementation. Additionally, the applications developed with Mars had a code size reduction up to seven times, compared with hand-tuned native CUDA code.

In summary, this paper makes the following contributions, in comparison with our previous work on Mars [22]:

- We have extended our general design of Mars from a GPU-only MapReduce framework to a MapReduce system with GPU acceleration enabled. With this extension, Mars components can work stand-alone on a single platform, e.g., MarsCUDA on CUDA, as well as to work together to utilize multiple processors, e.g., a CPU and a GPU on a single machine.
- We have implemented Mars for Brook+-based GPUs, which enables MapReduce to take advantage of another large group of GPUs in practice.
- We have developed simple but effective coprocessing schemes to utilize both the CPU and the GPU on a single machine. We have also integrated Mars into Hadoop to enable GPU-acceleration for individual machines in a distributed environment.

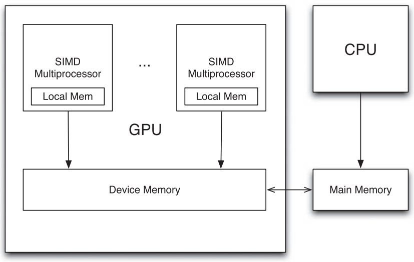

Fig. 1. The many-core architecture model for GPUs.

1.1 Organization

The remainder of the paper is organized as follows: We give a brief overview of GPUs, and review prior work on GPGPU and MapReduce in Section 2. We present the design and implementation details of Mars in Sections 3 and 4, respectively. We present the extension to multiple machines in Section 5. In Section 6, we present our experimental results. Finally, we conclude in Section 7.

2 Preliminaries and Related Work

In this section, we first give a brief introduction on the GPU, and then review the related work on GPGPU as well as on MapReduce.

2.1 Graphics Processing Units (GPUs)

The GPU is an integral component of modern computers, ranging from handheld devices to high-end servers. GPUs are originally designed for gaming applications with fixed hardware pipelines for rendering. Due to the high computation power and rapidly improving programmability, they have recently become a powerful coprocessor for general purpose computing [7].

As shown in Fig. 1, we model the GPU as a many-core processor, which contains a number of SIMD multiprocessors. Such a many-core model is common to both AMD and NVIDIA GPUs. On the GPU board, there is GRAM device memory. The device memory has both a high bandwidth and a high access latency. For example, the NVIDIA GTX280 GPU has an access latency of 400-600 cycles, and the peak memory bandwidth between the device memory and the multiprocessors is around 140 GB/second.

Both NVIDIA CUDA and AMD Brook+ expose a parallel programming model, which does not require programmers to have knowledge of the graphics rendering pipeline. In this model, the system consists of a *host* (a CPU), and one or more *devices* (GPUs). GPUs are abstracted as massively data-parallel coprocessors. CUDA and Brook+ programmers write code using C/C++ syntax with extended keywords for kernel functions, which are GPU programs to be executed on *devices*.

Programming frameworks, such as CUDA and Brook+, greatly improve the programmability of the GPU. However, it is still a challenging task of developing efficient GPU programs for complex applications, such as those with MapReduce, because GPUs have a special-purpose coprocessor architecture and are vendor-specific on the programming frameworks for complex applications. Although the newly introduced OpenCL [6] is an industry standard further hiding hardware details from users, Mars is at a higher level of abstraction. OpenCL is a general-purpose programming language, with which Mars or other MapReduce frameworks can be developed.

2.2 GPGPU

General Purpose Computation on GPUs, or GPGPU, has recently emerged in various applications, such as linear algebra [26], [35], embedded system design [19], bioinformatics [12], databases [20], [21], [25], [24], [16], machine learning [13], data mining [17], [18], and distributed computing projects including Folding@home and Seti@-home. Recently, several GPGPU languages including AMD Brook+ [1] (extended from Brook [10]) and NVIDIA CUDA [5] have been proposed by GPU vendors. They usually expose a general purpose, massively multithreaded computing architecture and provide a programming environment similar to C/C++. High-level programming frameworks, such as Accelerator [34] and RapidMind [29], are also developed to better facilitate GPGPU programming. These programming frameworks require programmers to have knowledge of specific programming models, such as the stream programming model in Brook+ [10], or even more, knowledge of the GPU hardware details. By contrast, we propose to develop a MapReduce framework accelerated with GPUs to ease the development of a more complex class of data processing tasks. It provides a uniform MapReduce interface no matter whether it runs on the GPU, on the CPU, or both.

We now briefly survey recent work that developed GPGPU primitives as building blocks for various applications, in particular, those not covered in the survey by Owens et al. [31]. Sengupta et al. [33] proposed the segmented scan primitive. He et al. [23] proposed a multipass scheme to optimize the scatter and the gather operations. He et al. [25] further developed a small set of primitives such as prefix sum and split for relational databases. Additionally, CUDPP [3], 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, programmers need to write complex GPU code for data processing tasks. By contrast, our work further simplifies GPU programming for MapReduce programmers by providing them with a higher level and more familiar interface than the primitives.

This paper focuses on accelerating MapReduce on the GPU, and provides a GPU-based MapReduce framework to developers. As in the original MapReduce, it is up to developers' choice to use MapReduce or not according to their computational characteristics. Recent studies [27] have used data analysis techniques to categorize the computational characteristics of different workloads on the GPU. These techniques are helpful for developers to determine whether their workloads are suitable for Mars in specific and the GPU in general.

Our previous study on Mars [22] implemented the MapReuduce framework on CUDA-enabled GPUs. This work extends the previous work in two major aspects. First, we extend the CUDA-only Mars to another large group of GPUs, so that it can run on both NVIDIA and AMD GPUs. Second, we use GPU-only Mars as a component to work with CPU-based Mars on a single machine as well as with Hadoop in a distributed environment.

2.3 MapReduce

The MapReduce framework [15] is based on two primitives, Map and Reduce, from functional programming. The general form is as follows:

Map: $(k_1, v_1) \rightarrow list(k_2, v_2)$.
Reduce: $(k_2, list(v_2)) \rightarrow list(k_3, v_3)$.

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 key/value pairs. Programmers implement the application logic inside the Map function and the Reduce function. The MapReduce runtime manages the parallel execution of these two functions.

The following pseudocode illustrates a program written using MapReduce. This program counts the number of occurrences of each word in a collection of documents [15]. In this program, *Map* and *Reduce* are implemented using two system-provided APIs, *EmitIntermediate* and *Emit*, respectively.

```
Map(void *doc) {
1:  for each word $w$ in doc
2:      EmitIntermediate($w$, 1); // count each word once
}
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
}
```

There have been several MapReduce implementations since MapReduce was proposed [15]. Hadoop [4] is an open-source MapReduce implementation on clusters. Based on Hadoop, Yang et al. [36] added the merge operation to MapReduce for the ease of relational databases operations. Phoenix [32] is an efficient MapReduce runtime system on multicore CPUs. Kruijf and Sankaralingam [14] developed MapReduce on the Cell BE. Yeung et al. [37] implemented an FPGA-based MapReduce system.

Let us briefly introduce the implementation of Phoenix [32]. A key component in Phoenix is a scheduler, for buffer management and task distribution. The scheduler starts the Map stage by evenly dividing the input buffer into small chunks, and assigns the chunks to map workers dynamically. Each map worker runs in a CPU thread. The Reduce stage does not start until all Map tasks are done. The scheduler groups the intermediate output from the Map stage by key, and a Reduce worker processes values associated with the same key. Reduce tasks are assigned to workers dynamically. Each reduce worker maintains a static array for outputting results and sorts this static array using insertion sort. Finally, the scheduler merges all output arrays of reduce workers into a single one. Because the output data size is not known in advance, the scheduler first allocates buffers with a default small size, and then resizes the buffer as needed.

Our previous work on Mars implemented MapReduce on CUDA-enabled GPUs [22]. Catanzaro et al. [11] developed another MapReduce system on the GPU, but it required programmers to be aware of GPU hardware details, such as thread configuration and memory hierarchy. Finally, the Merge framework [28], focused on dynamically scheduling MapReduce tasks among multiple processors, dedicated to Intel products. By contrast, our current extensions on Mars hide hardware details from programmers and work on heterogeneous GPUs, a combination of CPU and GPU on a single machine, as well as a distributed system of multiple machines.

3 Design of Mars

In this section, we present our design for Mars, with emphasis on the GPU-based component. Our design is guided by the following three goals:

1. *Programmability*. Ease of programming encourages programmers to use the GPU for their tasks.
2. *Flexibility*. The design should be applicable to various multi-/many-core processors, e.g., multicore CPUs and AMD GPUs, and should be as expressive as the underlying runtime, e.g., NVIDIA CUDA, AMD Brook+, or pthreads, so that the system will work for a wide range of hardware and applications.
3. *High Performance*. The overall performance should be accelerated by the GPU effectively.

3.1 Overview

By examining the Phoenix design, we see that there are three potential sources of overhead. First, the tight coupling of the Map and Reduce stages makes every application go through both stages, no matter whether they need both stages or not. Second, a dynamic thread scheduler for task assignment heavily relies on locking to implement synchronization. Third, each reduce worker may require frequent data movement for sorting the static output array, and the data movement can become a bottleneck for the overall performance. The latest paper about Phoenix also points out this problem [38].

In the Mars design, we decide to separate a MapReduce workflow into three loosely coupled stages—*Map*, *Group*, and *Reduce*. The *Group* stage is designed to group *Map* output by key, which is the format for *Reduce* input. Our observation is that some applications need only the *Map* stage, some need both *Map* and *Group*, and some need all of the three stages. The Group stage is the same as running Reduce with the identity function in the original MapReduce system [15]. Our purpose of providing an explicit Group stage is to allow a MapReduce application with high flexibility to customize its workflow, and to avoid the overhead of entering unnecessary stages. No matter what configuration of the three stages is for an application, the MapReduce interface of Mars is unchanged—users write Map and/or Reduce functions when necessary.

Moreover, we decide to use a lock-free scheme for synchronization and to perform in-advance buffer allocation. One reason is to avoid heavy overheads of locking and buffer reallocation. The other reason is that current GPUs do not support locking or in-flight buffer reallocation. In our design, we statically distribute tasks to a massive number of GPU threads, so that we can fully utilize the parallelism of the GPU. We adopt a two-phase, lock-free scheme for result output. The basic idea is that, in the first phase, we calculate histograms on the size of output results for each thread, followed by a prefix sum operation on the histograms, so that we obtain both the exact output buffer size and the deterministic write position for each thread; in the second phase, we perform the actual computation and output. We will detail this strategy in Section 3.4.

3.2 Data Structure

Data structures in Mars affect the workflow, memory access patterns, and the expressiveness of the system.

Since the GPU does not support dynamic memory allocation on the device memory during the execution of the GPU code, this limitation rules out dynamic data structures, such as queues and linked lists, as used in other MapReduce implementations. Instead, we use plain arrays as the main data structure in Mars. The *Map* stage takes input records in the key/value form and outputs intermediate result records, which are in turn the input of the *Group* stage. The output of the *Group* stage is the input of the *Reduce* stage, and *Reduce* produces final output records. Each of the three sets—the input records, the intermediate records, and the output records—is stored in three arrays, i.e., the key array, the value array, and the directory index array. The directory index consists of an entry of <key offset, key size, value offset, value size> for each key/value pair. Given a directory index entry, we fetch the key or the value at the corresponding offset in the key array or the value array.

Variable-sized types, such as strings, are supported with the directory index, since current GPUs have no such build-in types yet. 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.

Some applications perform chained MapReduce procedures, where the output of one MapReduce procedure is the input of another one. Since the sets of input records, intermediate records, and output records are all in the three-array structure uniformly, chained MapReduce is supported gracefully in Mars.

3.3 Mars Workflow

Fig. 2 illustrates the workflow of Mars, assuming the data resides in the disk at the beginning. The Mars scheduler runs on the CPU and schedules tasks to the GPU. Mars has three stages, *Map*, *Group*, and *Reduce*.

Before the *Map* stage, Mars preprocesses on the CPU the input data from disk, transforming the input data to key/value pairs (input records) in main memory. After that, it transfers input records from the main memory to the GPU device memory.

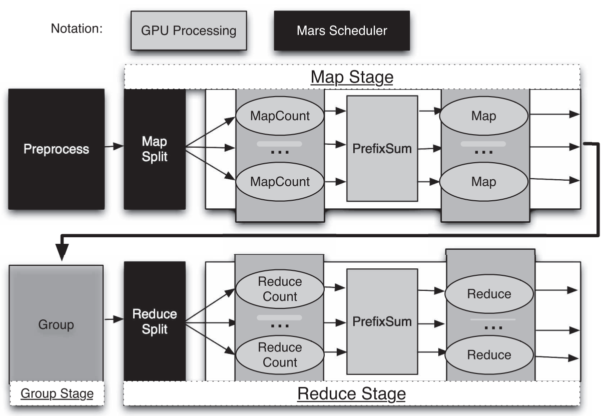

Fig. 2. The workflow of Mars on the GPU.

In the *Map* stage, *Map Split* dispatches input records to GPU threads, such that the workload for all threads is even. Each thread executes the user-defined MapCount function to compute a local histogram on the number and the total size of intermediate records that *Map* will output. Then, the runtime performs a GPU-based *Prefix Sum* on the local histograms to obtain the output size and the write position for each thread. Finally, after the CPU allocates the output buffer in the device memory, each GPU thread executes the user-defined Map function and outputs results. Since the write position for each thread is precomputed and has no conflict with any other threads, there will be no write conflict between concurrent threads. This lock-free scheme of MapCount, Prefix Sum, and Map is adapted from our previous work [25].

In the *Group* stage, both sort-based and hash-based approaches are available for grouping records by key. However, we adopt the sort-based, because some applications require to sort all output records, and the hash-based approach has to perform additional sort within each hash bucket.

In the *Reduce* stage, *Reduce Split* dispatches each group of records with the same key to a GPU thread. However, it may cause load imbalance between threads, since the number of records of different groups may vary widely. We adopt a skew handling scheme to alleviate the load imbalance problem (Section 3.6). The *Reduce* stage then works in a lock-free scheme, similar to that in *Map*, to obtain the result size and the write location for each thread. Finally, all Reduce workers output results to a single buffer.

Because these three stages are loosely coupled and not every application requires all stages, Mars allows users to customize the following workflows in their applications:

- MAP_ONLY. Mars executes the *Map* stage only and does not executes the *Group* or *Reduce* stage.
- MAP_GROUP. Mars executes the *Map* and *Group* stages and does not executes the *Reduce* stage.
- MAP_GROUP_REDUCE. Mars executes all three stages—*Map*, *Group*, and *Reduce* stages.

Because usually applications need a Map to transform input records, and a Group to prepare for the intermediate records to feed to Reduce, we exclude the other workflow configurations that skip either Map or Group in the presence of Reduce.

3.4 Lock-Free Scheme

With the array structure, we allocate the space on the device memory for the input data as well as for 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. To address these two problems, we adopt a previous lock-free output scheme for relational joins [25]. 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 MapCount invocation on a thread outputs three counts, i.e., the number of intermediate results, the total size of intermediate keys (in bytes), and the total size of intermediate values (in bytes). Based on intermediate key sizes (or value sizes), Mars computes a prefix sum on these sizes and produces an array of write locations. A write location is the start location in the output array for a map task to write. Based on the number of intermediate results, Mars computes a prefix sum and produces an array of start locations in the output directory index. Through these prefix sums, we also know the sizes of the arrays for the intermediate results. Finally, Mars allocates arrays in the device memory with the exact sizes for storing the intermediate results.

Second, each Map invocation on a thread outputs the intermediate key/value pairs to the output array. Since each Map has its deterministic and nonoverlapping positions to write to, the write conflicts are avoided.

The lock-free scheme is suitable for the massive thread parallelism on the GPU, even though it performs a MapCount in addition to a Map. The overhead of executing MapCount is application dependent and is usually small. For example, this overhead is negligible in the matrix multiplication in our study, since MapCount simply emits the size without performing the actual multiplication.

3.5 Rapid Group

The Group stage requires to sort intermediate records. However, we observe that some applications inherently have their intermediate records grouped after the Map phase, and each group has the same number of records. For example, [A,A,A,B,B,B,C,C,C] shows three groups with A, B, and C as the key, respectively, and each group is with the same size 3. For such applications, Mars provides a configuration parameter for users to whether the intermediate data are already grouped. The runtime automatically skips the time consuming sorting, and then dispatches each group of intermediate records with the same group size to Reduce workers. We name this strategy as "Rapid Group".

3.6 Skew Handling

We design a skew handling scheme to distribute workloads evenly across reduce workers, where the user-defined Reduce operation is commutative and associative. This scheme iteratively performs the *Reduce* stage in the following two steps. First, we divide the data into M equal-sized chunks. Second, we perform a reduction on each chunk. In this step, each of the M threads applies the reduce function on groups of records in a single chunk. Note, in each iteration, we perform reduction on the intermediate results with the same keys only.

TABLE 1
Mars APIs

Function Name	Description	Function Type
MAP_COUNT	It calculates the output buffer size of MAP.	User-implemented
MAP	The map function.	User-implemented
REDUCE_COUNT	It calculates the output buffer size of REDUCE.	User-implemented
REDUCE	The reduce function.	User-implemented
EMIT_INTERMEDIATE_COUNT	It emits the key size and the value size in MAP_COUNT.	System-provided
EMIT_INTERMEDIATE	It emits the key and the value in MAP.	System-provided
EMIT_COUNT	It emits the key size and the value size in REDUCE_COUNT.	System-provided
EMIT	It emits the key and the value in REDUCE.	System-provided

TABLE 2
Modules in the Mars System

Implementation	Software platform	Hardware platform
MarsCUDA	NVIDIA CUDA	an NVIDIA GPU
MarsBrook	AMD Brook+	an AMD GPU
MarsCPU	pthreads	a multi-core CPU
GPU/CPU co-processing	CUDA/Brook+ and pthreads	NVIDIA/AMD GPUs and multi-core CPUs

3.7 Mars 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 should implement by themselves, and the system-provided APIs, which the users can use as library calls. The definitions of these APIs are in Table 1.

4 Single-Machine Implementations

In this section, we present the implementation details of Mars on a single machine. The current Mars system consists of four modules (Table 2). All these four modules share the common design of Mars and provide the same MapReduce interface to the user. They can run on different hardware platforms: MarsCUDA on an NVIDIA GPU, MarsBrook on an AMD GPU, MarsCPU on a multicore CPU, and the GPU/CPU coprocessing module on both the CPU and the GPU through combining the aforementioned modules. Different modules in Mars allow programmers to take advantage of different processors on a single machine. Because our machines cannot host a multi-GPU configuration due to limited extension slots, we have not explored multi-GPU coprocessing.

4.1 MarsCUDA

We implemented MarsCUDA using NVIDIA CUDA. We used the GPU Prefix Sum routine from CUDPP [3] to implement the lock-free scheme, and the GPU Bitonic Sort routine for the *Group* phase. CUDA exposes sufficient hardware details of NVIDIA GPUs, so that we can apply some optimizations in MarsCUDA runtime.

4.1.1 Memory Access

Coalesced access. We utilize the NVIDIA GPU feature of coalesced access to improve the memory performance. In CUDA, simultaneous device memory accesses by threads in a half-warp (warp is an NVIDIA term for a group of 32 threads for scheduling) can be coalesced into a single memory transaction, which significantly reduces the number of device memory accesses. We implement the access to the directory index arrays as coalesced.

Local memory. NVIDIA GPUs provide the programmable on-chip local memory (or *shared memory* [30]), for sharing data among threads running on the same multiprocessor. It is important to fully utilize the local memory to reduce the costly accesses to the GPU memory. In Mars, data sharing or communication only happens in the Group stage. MarsCUDA runtime automatically uses a GPU-based bitonic sort [22] to exploit this memory hierarchy in the Group stage. Mars does not expose the local memory to the user-defined functions in the Map and the Reduce stages. Since local memory is programmer-controlled fast memory, it introduces complexity and needs the effort from the programmer. This is a trade-off between performance and programmability. Nevertheless, users who are aware of the GPU memory hierarchy and need such data sharing can exploit the local memory in implementing the Map (or Reduce) function.

Built-in vector types. Data accesses in the GPU device memory should be aligned to make sure the correctness and achieve high memory bandwidth. Fortunately, GPUs support built-in vector types [30], including *float4* and *int4*. The alignment requirement is automatically fulfilled for built-in types. In addition, the GPU is able to issue a single load instruction to read data of built-in type, of size up to 16 bytes. Compared with reading an array one *float* or *int* at a time, the number of compiler-generated instructions for reading *float4* or *int4* is greatly reduced and the overall performance is improved.

Page-locked host memory. CUDA supports page-locked host memory (a.k.a pinned), which prevents the operating system for paging the locked memory buffer, yielding high transfer bandwidth between the device memory and the host memory [30]. The MarsCUDA runtime utilizes the page-locked host memory mechanism, in order to reduce the data transfer overhead. Our test demonstrated that page-locked memory can double the memory transfer rate through PCI-E bus than pageable memory.

4.1.2 Parallelism

Since CUDA exposes the thread configuration, we utilize the parallelism by assigning the tasks to a large number of threads. The thread configuration, i.e., the number of thread blocks and the number of threads per thread block, is related to both hardware and software factors: 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, and 2) the characteristics of the map and the reduce tasks, e.g., the degree of memory- or computation-intensiveness.

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. CUDA provides an offline calculator[1] 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 block and the number of registers used per thread as input, and outputs the occupancy and the number of active thread blocks 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 block in multiples of 32 (the schedule unit size) ranging from 32 to 512 (the maximum number of threads per thread block), until the occupancy is higher than a predefined threshold. Thus, we get the number of threads per thread block and the number of thread blocks. In practice, we set the occupancy threshold to be 2/3, so that the GPU is sufficiently busy, and each thread block receives adequate computation resources.

4.2 MarsBrook

We implement MarsBrook on AMD GPUs using the stream programming model Brook+ [1]. Due to the limitation of Brook+, MarsBrook is less advanced than MarsCUDA in both expressivity and performance. Nevertheless, as programming support of Brook+ improves, MarsBrook can demonstrate a higher flexibility and performance.

MarsBrook requires users to specify the data types of keys and values statically, and each record is of a fixed size. Type conversion is not allowed in Brook+. Unlike CUDA, Brook+ does not allow the developer to access data in GPU memory by arbitrary address. Instead, data in GPU memory are accessed using a *stream*, which is essentially a sequentially accessed array of fixed-sized elements. Random access in a stream is achieved by providing another predefined stream, consisting of indexes of target elements to access. Although the Mars APIs are the same on CUDA and on Brook+, as listed in Table 1, using Mars on CUDA is more flexible than using that on Brook+, when the Mars user develops a user-defined function.

Moreover, MarsBrook has relatively limited room for performance optimization. The reason is that Brook+ does not expose detailed hardware features, e.g., fast on-chip local memory, coalesced memory access, or GPU thread configuration.

4.3 MarsCPU

We implement MarsCPU using the pthreads library on linux for multithreading. Instead of adopting lock-based task scheduling as in Phoenix, MarsCPU inherits the lock-free design of GPU-based Mars, which we expect to scale to hundreds of cores for future many-core CPUs. MarsCPU deploys CPU threads to perform Map and Reduce tasks. If there are N Map (or Reduce) tasks and T CPU threads, where N is usually much larger than T, then a thread processes $\lceil N/T \rceil$ tasks. We implement a CPU multi-threaded parallel mergesort for the Group stage.

1. http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls.

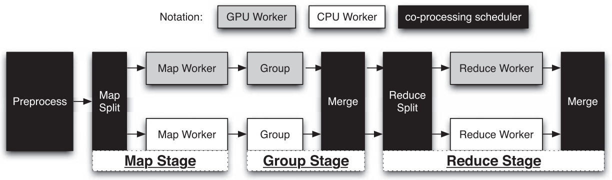

Fig. 3. The workflow of GPU/CPU coprocessing.

4.4 GPU/CPU Coprocessing

The workflow of GPU/CPU coprocessing is shown in Fig. 3. There are also mainly three stages, *Map*, *Group*, and *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 S, 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 $\frac{SI}{1+S}$ and $\frac{I}{1+S}$ bytes to the GPU and the CPU workers, respectively. The speedup S can be obtained by either calibration or predictive model [27].

When a processor finishes a *Map* task, it performs a local *Group* on intermediate results. The runtime merges all intermediate results. When all the processors finish their tasks, the *Map* stage ends.

The *Reduce* stage takes the intermediate results from the *Group* stage as input. Similar to the *Map* stage, the coprocessing scheduler statically assigns the data chunks to the processors. When all the processors finish their tasks, the runtime merges all local results.

Mars dispatches workload between the GPU worker and the CPU worker only if the following conditions are satisfied. First, the Map and Reduce stages take up high proportion of the entire running time on the CPU worker. If components other than the Map and Reduce stages contribute to a large portion of running time, the GPU worker is not able to make large performance acceleration. Second, the GPU worker and the CPU worker have comparable performance. The benefit of using the CPU worker diminishes as the speedup of the GPU worker over the CPU worker becomes higher.

With the GPU/CPU coprocessing module, Mars can harness the computation power of NVIDIA GPUs, AMD GPUs, and multicore CPUs on the same machine by integrating MarsCUDA, MarsBrook, and MarsCPU modules as components.

5 Multimachine Implementations

In this section, we present the integration of Mars into a CPU-based distributed MapReduce system, specifically Hadoop in our implementation. This integration benefits from both

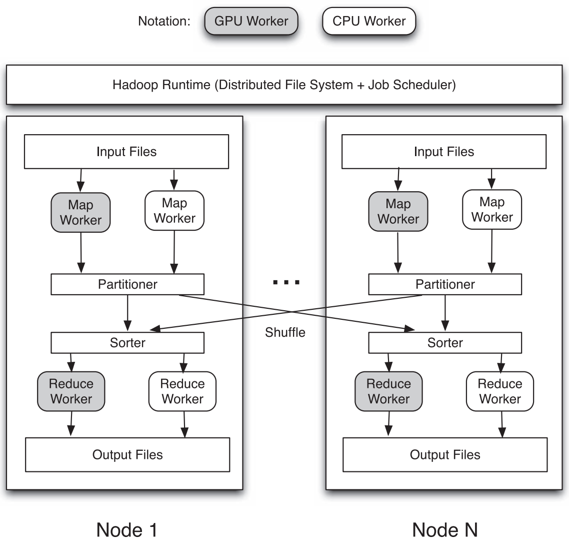

Fig. 4. MarsHadoop. Some Map and Reduce tasks are performed on the GPU, and others are on the CPU.

worlds: Hadoop utilizes CPUs on multiple machines and provides fault tolerance and other features of a distributed system; Mars utilizes the GPU to accelerate local computation. We denote Mars-enabled Hadoop as MarsHadoop.

We use the *Hadoop Streaming* technology[2] to integrate Mars into Hadoop. *Hadoop Streaming* enables the developers to use their own custom Map or Reduce implementation in Hadoop. In our implementation, we use the Mars executable to read the input from *stdin* and to emit the output to *stdout*. Thus, the Map and the Reduce tasks can be performed on the GPU, and other tasks, such as task scheduling and failure handling, are performed by Hadoop. Finally, since current GPUs do not support multitasking, we configure MarsHadoop to run GPU-based tasks sequentially.

Fig. 4 illustrates the workflow of MarsHadoop. A Map Worker/Reduce Worker in MarsHadoop is the same as a Map Worker/Reduce Worker shown in Fig. 3; in other words, it can be from MarsCUDA, MarsBrook, or MarsCPU, depending on the underlying processor. In the configuration of Fig. 4, Node 1 simultaneously runs two Map Workers, on a GPU and a CPU, respectively.

6 Experimental Evaluation

In this section, we evaluate Mars on a single machine using a microbenchmark of six applications in comparison with their CPU-based counterparts and native GPU-based implementations. We also evaluate the performance of MarsHadoop on two connected machines.

6.1 Experimental Setup

Our experiments were performed on three PCs, A, B, and C. Table 3 shows their hardware configuration. Both PCs A and B run 32-bit CentOS 5.1 Linux with kernel 2.6.18, NVIDIA CUDA 2.2, and the GPU driver 185.18.14. PC C runs 32-bit Windows XP Pro SP3, with Brook+ 1.01.0 beta, and the GPU driver 8.561. All hard drives on these PCs are SATA magnetic hard disks with 7,200 rpm. On all PCs, the main memory and the device memory are connected by PCI-E bus with a theoretical bandwidth of 4 GB/second.

2. http://hadoop.apache.org/common/docs/r0.15.2/streaming.html.

6.2 Microbenchmark

We have implemented the following six real-world applications for evaluating the MapReduce framework.

String Match (SM): Each Map task searches a portion of the input file to check whether the target string is in the portion. Neither the *Group* nor the *Reduce* stage is needed.

Matrix Multiplication (MM): Matrix multiplication is used intensively in analyzing the relationship of two documents. Given two matrices M and N, each Map task 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 the *Group* nor the *Reduce* stage is needed.

Black-Scholes (BS): Black-Scholes model [8] is used for calculating the price for European options according to a partial differential equation. For each option, a Map task computes the prices for the call and put prices of an option, and emits a structure containing the price of the option call and the price of the option put as the key, and the option id as the value. The Group stage is to rank the price of option calls. No Reduce stage is needed.

Similarity Score (SS): It is used in web document clustering. The characteristics of a document are represented using a feature vector of floating point numbers. 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 task computes the similarity score for two documents. It outputs the intermediate pair with the score as the key and the pair of the two document IDs as the value. The *Group* stage is required to rank the pair-wise similarity scores and no *Reduce* stage is required.

Principal component analysis (PCA): This application computes the mean vector and the covariance matrix of a set of points in the first two steps in PCA. The input data are stored in a matrix. The whole process contains two MapReduce invocations in a chain. The first MapReduce procedure is to find the mean for each row in the matrix, and the second is to calculate the covariance matrix. Neither Group nor Reduce stage is needed in the first MapReduce invocation. A Map task computes the mean for a row. In the second invocation, each Map task is to calculate the covariance of two rows. The Group stage is required to sort the row pairs by row IDs. No Reduce phase is needed.

Monte Carlo (MC): Monte Carlo [9] is used to compute option pricing in financial engineering. The Monte Carlo numeric integration is to mathematically estimate the expectation of the price of option call. Each Map task is to compute the expected value of a random sample for an option, and to emit the option ID as the key, while the expected value of the random sample as the value. The Group stage and the Reduce stage are required to calculate the mean of all the samples for each option. In this application, all the options have the same number of samples, and the intermediate results are ordered by option ID already. Mars does not need to perform sorting in the Group stage.

TABLE 3
Machine Configurations

Machine	PC A	PC B	PC C
GPU	NVIDIA GTX280	NVIDIA 8800GTX	ATI Radeon HD 3870
# GPU core	240	128	320
GPU Core Clock (MHz)	602	575	775
GPU Memory Clock (MHz)	1107	900	2250
GPU Memory Bandwidth (GB/s)	141.7	86.4	72.0
GPU Memory Capacity (MB)	1024	768	512
CPU	Intel Core2 Quad Q6600	Intel Core2 Quad Q6600	Intel Pentium 4 540
CPU Clock (MHz)	2400	2400	3200
# CPU core	4	4	2
CPU Memory Capacity (MB)	2048	2048	1024

TABLE 4
The Input Data Sizes of the Microbenchmark

Applications	Small	Medium	Large
String Match	size: 55MB	size: 105MB	size: 160MB
Matrix Multiplication	256x256	512x512	1024x1024
Black-Scholes	# option: 1,000,000	# option: 3,000,000	# option: 5,000,000
Similarity Score	# feature: 128, # documents: 512	# feature: 128, # documents: 1024	# feature: 128, # documents: 2048
PCA	1000x256	2000x256	4000x256
Monte Carlo	# option: 500, # samples per option: 500	# option: 500, # samples per option: 2500	# option: 500, # samples per option: 5000

The above applications are commonly used in benchmarking MapReduce implementations in the previous studies [32], [13]. SM, MM, and PCA are adopted from the Phoenix suite [32], SS is a common component in web applications, while BS and MC are prevalent in financial engineering and are adopted from CUDA SDK. In particular, the workflow of these applications differ: SM and MM only have the *Map* stage; BS, SS, and PCA have *Map* and *Group* stages; and MC has all the three stages. PCA has a chain of multiple MapReduce procedures, whereas other applications have only one MapReduce invocation each.

Within a single machine, we used three data sets for each application (S, M and L) to evaluate the scalability of the MapReduce framework. The input for SM is textual data and is adopted from Phoenix [32]; The input for all the other applications contains randomly generated real numbers, ranging from zero to one. All these input data are stored as files in the hard disk. We summarize the size of input data for each application in Table 4.

With the microbenchmarks, we have compared the performance and programmability of the MapReduce frameworks between the CPU and the GPU. The third party MapReduce on the CPU is the latest release of Phoenix in version 2.0.0. As for native implementation, we have implemented the applications directly on CUDA and pthreads.

Metrics. The wall time is the major metric for the performance evaluation. We measure the elapsed time of each application from reading data from the disk till generating results in the main memory. We ran each experiment five times and report the average value. The variation of elapsed time between runs is negligible. The performance speedup on A over B is defined as the running time of B divided by the running time of A. The performance slowdown on A over B is defined as the running time of A divided by the running time of B.

We use the number of code lines written by the user as the metric for comparing the programmability of different MapReduce implementations as well as the native implementation with CUDA and Brook+. Note that we exclude comments and empty lines from the code size counting.

6.3 Results on a Single Machine

On a single machine, we have compared the performance and programmability of the MapReduce frameworks between the CPU and the GPU. We have implemented the six applications on MarsCUDA, MarsCPU, and the latest release of Phoenix in version 2.0.0. We have also implemented the applications directly on CUDA and pthreads, respectively, including thread configuration, data distribution, task execution, buffer management, and various memory optimizations.

We present the results on the NVIDIA GPU in detail, and briefly present the results on the AMD GPU, mainly demonstrating the feasibility.

6.3.1 Results on MarsCUDA and MarsCPU

Programmability. Table 5 shows the comparison of user code size, for implementing the microbenchmark with MarsCUDA, MarsCPU, Phoenix, and CUDA. By design, the code sizes with MarsCUDA are the same as those with MarsCPU. In general, the applications with MarsCPU have a similar code size to those with Phoenix. If the *Group* stage is required, applications like SS with MarsCUDA have a much smaller code size than when manually written using CUDA, due to an optimized but lengthy group function on CUDA. The user code size of MarsCUDA is up to seven times smaller than that of the native implementation with CUDA, which demonstrates the ease of GPU programming using Mars.

TABLE 5
Comparison of Application Code Size on MarsCPU, MarsCUDA, Phoenix, and CUDA

Applications	Phoenix	MarsCUDA/MarsCPU	CUDA
String Match	206	147	157
Matrix Multiplication	178	72	68
Black-Scholes	199	147	721
Similarity Score	125	82	615
Principal component analysis	297	168	583
Monte Carlo	251	203	359

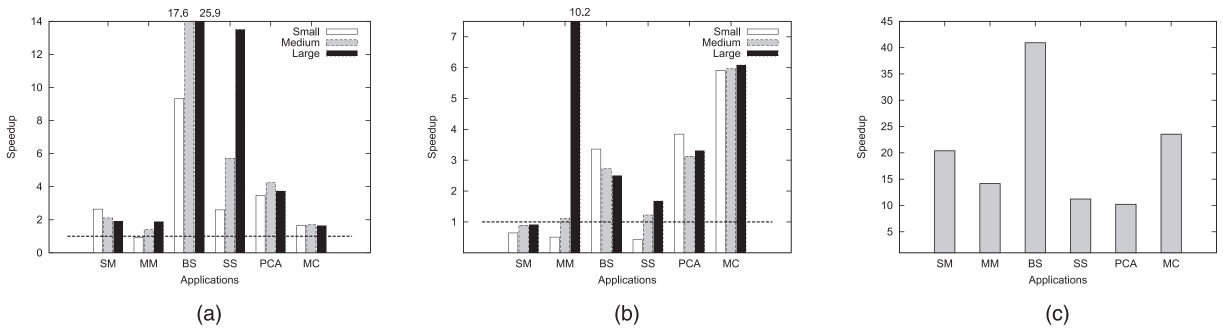

Fig. 5. Performance evaluation for MarsCPU and MarsCUDA on the microbenchmark. (a) Performance speedup on MarsCPU over Phoenix. (b) Performance speedup on MarsCUDA over MarsCPU (The entire MapReduce). (c) Performance speedup on MarsCUDA over MarsCPU (Map and Reduce stages only).

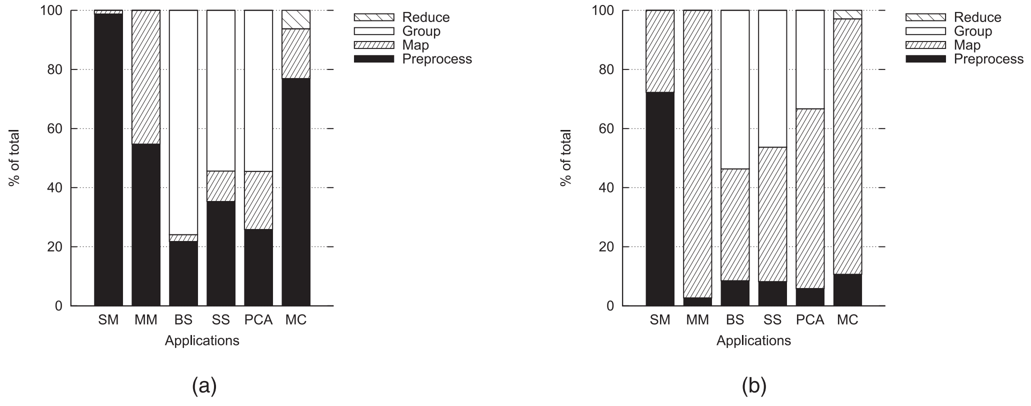

Fig. 6. Time breakdown of MarsCUDA and MarsCPU on the microbenchmark. (a) Time breakdown of MarsCUDA. (b) Time breakdown of MarsCPU.

Overall performance on MapReduce. We conducted the performance evaluation of MarsCUDA and MarsCPU on PC A by comparing with Phoenix. Fig. 5 shows the overall performance comparison. Both MarsCUDA and MarsCPU outperform Phoenix for the six applications due to the general lock-free design of Mars.

The overall performance of MarsCPU is generally better than that of Phoenix, achieving a speedup of up to 25.9x. Applications written using Phoenix always have a *Reduce* stage, whereas using ours they may not have. Phoenix maintains a global 2D array of pointers to keys array. Each keys array is in essence a contiguous buffer as a bucket for hashing and is sorted by insertion sort when a new key arrives. Such design incurs two serious performance bottlenecks. First, lock-based synchronization is needed. Second, lots of memory buffer movements (calling memmove()) are required for insertion sort in the static array. In contrast, the design of Mars is lock free and each Map task or Reduce task has deterministic output buffer size and writing positions, so neither lock nor memory management overhead would be introduced. In particular, BS and SS that require to rank distinct real numbers are over 10x slower on Phoenix than on MarsCPU. That is because Phoenix has to deploy millions of identity reduce tasks for these two applications. Our profiling results obtained from Intel VTune show that over 99 percent of the total execution time of BS and SS on Phoenix is contributed to the *memmove()* operations in the Reduce stage.

As shown in Fig. 5c, MarsCUDA utilizes the GPU hardware to accelerate the Map and Reduce stages for the six applications and outperforms MarsCPU in the two stages by 21x on average, and up to 40.9x. Please note that, this speedup is obtained without specific performance tuning on the GPU code, e.g., exploiting local memory. When it turns to the overall performance, MarsCUDA has a 10x speedup over MarsCPU for MM, and 6x for MC, but not so impressive speedup for the other applications (Fig. 5b). In order to figure out the source of slowdown in overall speedup, we further investigate the time breakdown of each application on the large data set for both MarsCUDA and MarsCPU.

We divide the total execution time into four components, including the time for 1) preprocessing input data ("Preprocess"), including input file I/O, generating key/value pairs, and transfering data from main memory to device memory, 2) the *Map* stage ("Map"), 3) the *Group* stage ("Group"), and 4) the *Reduce* stage ("Reduce"). MarsCPU totally runs on multicore CPUs, and thus, the PCI-E I/O time in Fig. 6b is always zero. MarsCUDA generally has a larger portion of preprocess time, involving key/value pair preparation and PCI-E I/O. In addition, the GPU-based Group stage has limited speedup over the CPU-based. We use Amdahl's law to explain this speedup involving parallel and sequential executions. Take SM for example. Although the GPU accelerates the Map phase by 20 times, the Map only takes up some 25 percent in MarsCPU. According to Amdahl's law, the theoretical speedup of MarsCUDA over MarsCPU is at most 1.3. Our measurement is close to this theoretical speedup.

Scaling. We used the clock rate scaling tool NVClock[3] to vary the NVIDIA GPU's core clock rate and memory clock rate, in order to evaluate the impact of hardware capability on MarsCUDA. Figs. 7a and 7b show the performance result of the six applications running on MarsCUDA with the large data set.

3. http://www.linuxhardware.org/nvclock/.

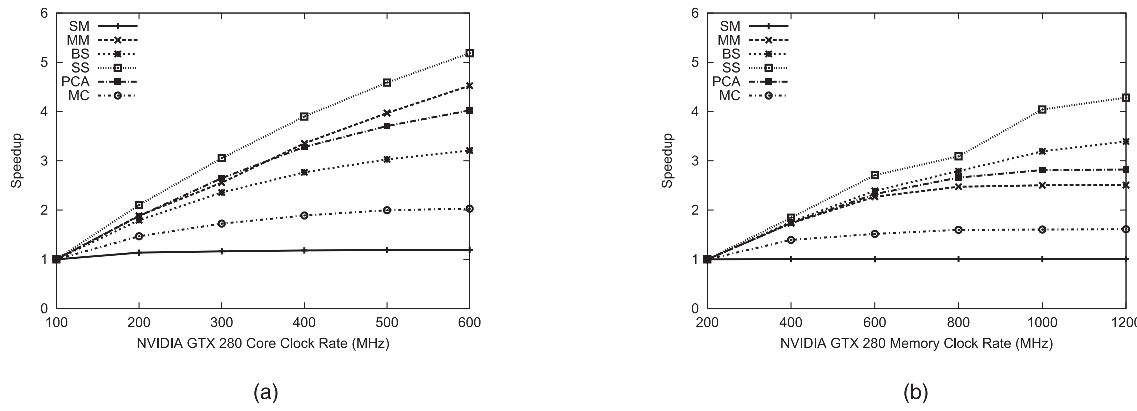

Fig. 7. Varying clock rates on GTX 280. (a) Baseline: running at 100 MHz core clock rate. Memory clock rate: fixed to 1,100 MHz. (b) Baseline: running at 200 MHz memory clock rate. Core clock rate: fixed to 600 MHz.

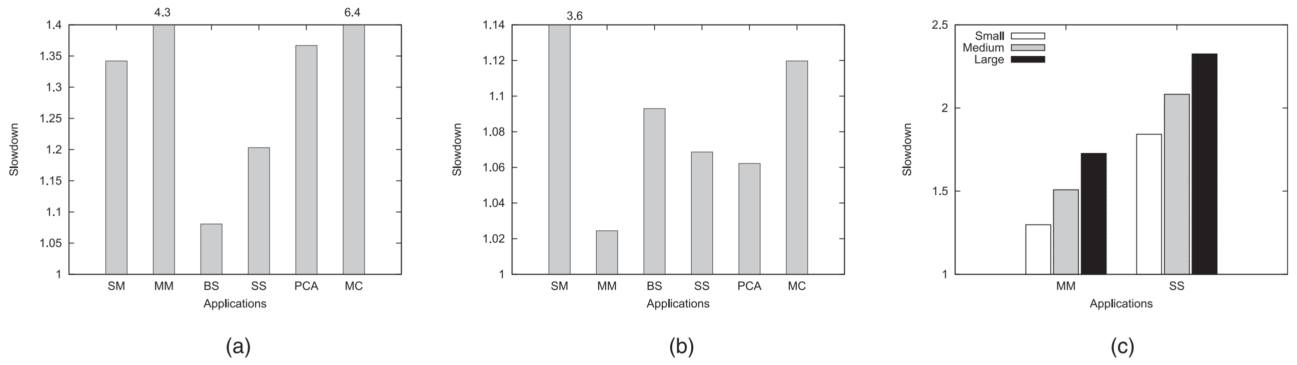

Fig. 8. The performance slowdown of Mars over native implementations. (a) MarsCUDA over CUDA. (b) MarsCPU over pthreads. (c) MarsBrook over Brook+.

In general, most applications (except for SM) on MarsCUDA are sensitive to both core clock rate and memory clock rate. This result indicates that MarsCUDA can scale well as the GPU evolves. SM is not sensitive to the hardware scaling, since its GPU computation time is relatively small (as shown in Fig. 6a).

Comparison with native implementation. Fig. 8a shows the performance slowdown of the six applications on MarsCUDA over the native implementation, with large data set. Overall, the implementation of applications based on MarsCUDA has roughly the same performance as on CUDA. However, MM and MC performs much poorer on MarsCUDA, mainly due to two reasons. One reason is rooted at the potential deficiency of MapReduce compared with a native implementation, as a previous study has already demonstrated [32]. The other reason is that MarsCUDA does not automatically exploit the local memory to improve the temporal locality due to the lack of knowledge about specific applications. Similarly, Fig. 8b illustrates that applications on MarsCPU has roughly the same performance as on pthreads.

6.3.2 Results on MarsBrook

Due to the limitation of Brook+, we have developed only two numerical applications (i.e., MM and SS) on MarsBrook. Table 6 shows the code size of applications written in MarsBrook compared with the native implementation in Brook+. The result is consistent with the comparison between MarsCUDA and the native CUDA implementation. For example, the native implementation of SS has a much larger code size than that on MarsBrook, since SS requires a *Group* stage.

Fig. 8c shows the performance slowdown of two applications by using MarsBrook over the native implementation. The implementation on top of MarsBrook is up to twice slower than the native implementation, which is the price to pay for ease of programming.

TABLE 6
Code Size of MM and SS between MarsBrook and Brook+

Applications	MarsBrook	Brook+
MM	66	93
SS	66	611

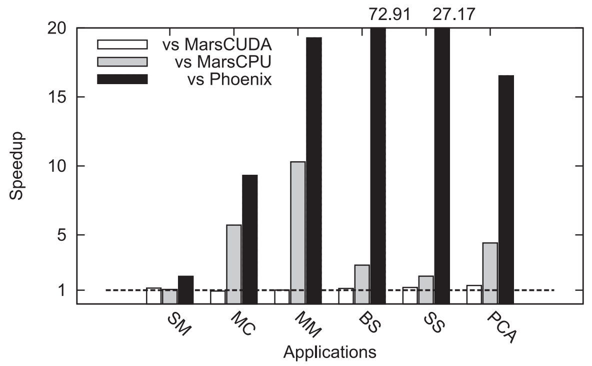

Fig. 9. Performance speedup of GPU/CPU coprocessing module over MarsCUDA, MarsCPU, and Phoenix.

3. http://www.linuxhardware.org/nvclock/.

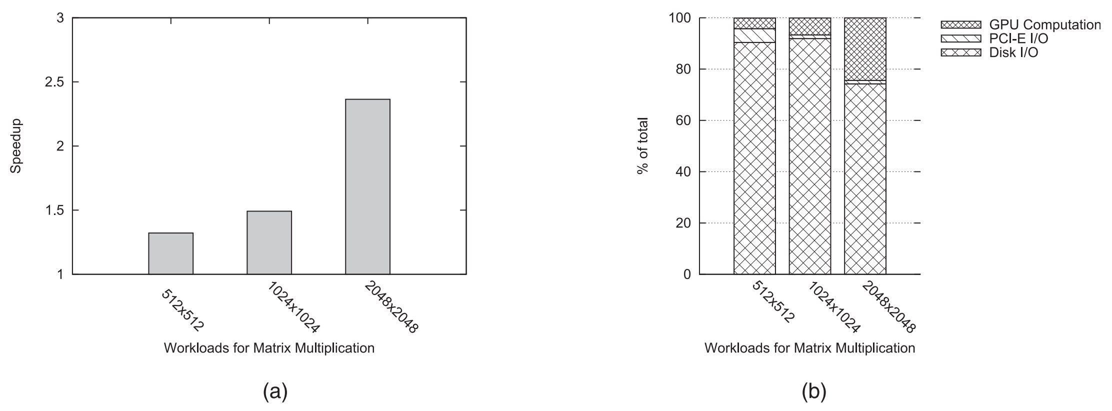

Fig. 10. Matrix multiplication on MarsHadoop. (a) Performance speedup on MarsHadoop over native Hadoop. (b) Time breakdown on MarsHadoop.

6.3.3 *Results on GPU/CPU Coprocessing of Mars*

We used MarsCUDA and MarsCPU as two components in the coprocessing. Fig. 9 shows the performance speedup of the GPU/CPU coprocessing module over MarsCUDA, MarsCPU, and Phoenix, on the large data set. Overall, coprocessing utilizes the computation power of both the CPU and the GPU, and yields a considerable performance improvement over using MarsCPU or Phoenix on a CPU. However, the speedup of using coprocessing over using stand-alone MarsCUDA is limited.

The workload dispatching between MarsCUDA and MarsCPU in coprocessing mainly depends on the performance comparison between the CPU processing and the GPU processing. The theoretical speedup of coprocessing over MarsCUDA would be $(S+1)/S$, where S is the speedup of using MarsCUDA over using MarsCPU. For example, if the speedup S is 10, then using coprocessing would only outperform using stand-alone MarsCUDA by a factor of $\frac{10+1}{10} = 1.1$. Therefore, for compute-intensive applications MM, BS, SS, MC, and PCA, using coprocessing cannot boost the performance considerably over using the stand-alone MarsCUDA. For SM that spends most time in preprocessing, using coprocessing can hardly achieve the theoretical speedup $\frac{1+1}{1} = 2$. Nevertheless, applications using coprocessing of MarsCUDA and MarsCPU still outperform Phoenix with a speedup of 24 times on average, and 72 times at maximum.

6.4 Results on MarsHadoop

We experimented MM on MarsHadoop. We configured Hadoop on PCs A and B: PC A as the master node, while PC A itself and PC B as slave nodes.

Fig. 10a shows the performance speedup of MarsHadoop over the native Hadoop implementation on MM. As the matrix size varied, MarsHadoop is up to 2.8 times faster than the native Hadoop implementation. We further examine the time breakdown in the slave node, and the results are shown in Fig. 10b. As the matrix size increases, the ratio for the computation time grows, indicating that Mars starts to help. The disk I/O is mainly due to the extra I/O caused by Hadoop streaming.

7 Conclusion

Graphics processors have become an efficient accelerator for high-performance computing. This paper proposes Mars, which harnesses the GPU computation power and high memory bandwidth to accelerate MapReduce frameworks. Mars is applicable to run on NVIDIA GPUs, AMD GPUs, multicore CPUs, and Hadoop-based distributed systems. Our empirical studies show that Mars improves the programmability of both the NVIDIA and the AMD GPUs, and the GPU acceleration with an NVIDIA GTX280 achieved a speedup of an order of magnitude over a quad-core CPU. Utilizing both the GPU and the CPU further improved GPU-only performance by 40 percent for some applications. Additionally, integrating Mars into Hadoop enabled GPU acceleration for a network of PCs.

The code and documentation of Mars can be found at http://www.cse.ust.hk/gpuqp/.

Acknowledgments

This work was done when the first author (Wenbin Fang) was a student in Hong Kong University of Sciences and Technology (HKUST).

References

[1] *AMD Brook+.* http://ati.amd.com/technology/streamcomputing/, 2010.
[2] *CUDA—Tutorial 5—Performance of Atomics.* http://supercomputingblog.com/cuda/cuda-tutorial-5-performance-of-atomics, 2010.
[3] *CUDPP.* http://gpgpu.org/developer/cudpp/, 2010.
[4] *Hadoop.* http://ati.amd.com/technology/streamcomputing/, 2010.
[5] *NVIDIA CUDA.* http://www.nvidia.com/cuda, 2006.
[6] *OpenCL.* http://www.khronos.org/opencl/, 2008.
[7] A. Ailamaki, N.K. Govindaraju, S. Harizopoulos, and D. Manocha, "Query Co-Processing on Commodity Processors," *Proc. 32nd Int'l Conf. Very Large Data Bases (VLDB),* 2006.
[8] F. Black and M.S. Scholes, "The Pricing of Options and Corporate Liabilities," *J. Political Economy,* vol. 81, no. 3, pp. 637-54, May-June 1973.
[9] P.P. Boyle, "Options: A Monte Carlo Approach," *J. Financial Economics,* vol. 4, pp. 323-338, 1977.
[10] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream Computing on Graphics Hardware," *Proc. ACM SIGGRAPH,* 2004.
[11] B. Catanzaro, N. Sundaram, and K. Keutzer, "A Map Reduce Framework for Programming Gpus," *Proc. Third Workshop Software Tools for MultiCore Systems (STMCS),* 2008.
[12] M. Charalambous, P. Trancoso, and R. Stamatakis, "Initial Experiences Porting a Bioinformatics Application to a Graphics Processor," *Proc. 10th Panhellenic Conf. Informatics,* 2005.
[13] C.-T. Chu, S.K. Kim, Y.-A. Lin, Y.Y. Yu, G. Bradski, A.Y. Ng, and K. Olukotun, "Map-Reduce for Machine Learning on Multicore," *Proc. Neural Information Processing Systems Conf. (NIPS),* 2006.

[14] M. de Kruijf and K. Sankaralingam, "Mapreduce for the Cell B.E. Architecture," technical report, Univ. of Wisconsin at Madison, 2007.

[15] J. Dean and S. Ghemawat, "Mapreduce: Simplified Data Processing on Large Clusters," *Proc. Sixth Conf. Symp. Opearting Systems Design and Implementation (OSDI),* 2004.

[16] W. Fang, B. He, and Q. Luo, "Database Compression on Graphics Processors," *Proc. 36th Int'l Conf. Very Large Data Bases (VLDB),* 2010.

[17] W. Fang, K.K. Lau, M. Lu, X. Xiao, C.K. Lam, P.Y. Yang, B. He, Q. Luo, P.V. Sander, and K. Yang, "Parallel Data Mining on Graphics Processors," Technical Report HKUST-CS08-07, Hong Kong Univ. of Science and Technology (HKUST), 2008.

[18] W. Fang, M. Lu, X. Xiao, B. He, and Q. Luo, "Frequent Itemset Mining on Graphics Processors," *Proc. Fifth Int'l Workshop Data Management on New Hardware (DaMoN '09),* pp. 34-42, 2009.

[19] J. Feng, S. Chakraborty, B. Schmidt, W. Liu, and U.D. Bordoloi, "Fast Schedulability Analysis Using Commodity Graphics Hardware," *Proc. 13th IEEE Int'l Conf. Embedded and Real-Time Computing Systems and Applications (RTCSA),* 2007.

[20] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha, "GPUTeraSort: High Performance Graphics Co-Processor Sorting for Large Database Management," *Proc. ACM SIGMOD,* 2006.

[21] N.K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha, "Fast Computation of Database Operations Using Graphics Processors," *Proc. ACM SIGMOD,* 2004.

[22] B. He, W. Fang, Q. Luo, N.K. Govindaraju, and T. Wang, "Mars: A Mapreduce Framework on Graphics Processors," *Proc. 17th Int'l Conf. Parallel Architectures and Compilation Techniques (PACT),* 2008.

[23] B. He, N.K. Govindaraju, Q. Luo, and B. Smith, "Efficient Gather and Scatter Operations on Graphics Processors," *Proc. ACM/IEEE Conf. Supercomputing,* 2007.

[24] B. He, M. Lu, K. Yang, R. Fang, N.K. Govindaraju, Q. Luo, and P.V. Sander, "Relational Query Coprocessing on Graphics Processors," *ACM Trans. Database Systems,* vol. 34, no. 4, pp. 1-39, 2009.

[25] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander, "Relational Joins on Graphics Processors," *Proc. ACM SIGMOD,* 2008.

[26] C. Jiang and M. Snir, "Automatic Tuning Matrix Multiplication Performance on Graphics Hardware," *Proc. 14th Int'l Conf. Parallel Architectures and Compilation Techniques (PACT),* 2005.

[27] A. Kerr, G. Diamos, and S. Yalamanchili, "Modeling GPU-CPU Workloads and Systems," *Proc. Third Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU-3),* 2010.

[28] M.D. Linderman, J.D. Collins, H. Wang, and T.H. Meng, "Merge: A Programming Model for Heterogeneous Multi-Core Systems," *Proc. 13th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS),* 2008.

[29] M.D. McCool, "Data-Parallel Programming on the Cell BE and the GPU Using the Rapidmind Development Platform," *Proc. GSPx Multicore Applications Conf.,* 2006.

[30] NVIDIA Corp., *NVIDIA CUDA Programming Guide 2.0,* 2008.

[31] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A.E. Lefohn, and T.J. Purcell, "A Survey of General-Purpose Computation on Graphics Hardware," *Computer Graphics Forum,* vol. 26, no. 1, pp. 80-113, 2007.

[32] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating Mapreduce for Multi-Core and Multiprocessor Systems," *Proc. IEEE 13th Int'l Symp. High Performance Computer Architecture (HPCA),* 2007.

[33] S. Sengupta, M. Harris, Y. Zhang, and J.D. Owens, "Scan Primitives for GPU Computing," *Proc. 22nd ACM SIGGRAPH/EUROGRAPHICS Symp. Graphics Hardware,* 2007.

[34] D. Tarditi, S. Puri, and J. Oglesby, "Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses," *Proc. 12th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS),* 2006.

[35] V. Volkov and J.W. Demmel, "Benchmarking GPUs to Tune Dense Linear Algebra," *Proc. ACM/IEEE Conf. Supercomputing,* 2008.

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

[37] J.H. Yeung, C. Tsang, K. Tsoi, B.S. Kwan, C.C. Cheung, A.P. Chan, and P.H. Leong, "Map-Reduce as a Programming Model for Custom Computing Machines," *Proc. 16th Int'l Symp. Field-Programmable Custom Computing Machines (FCCM),* 2008.

[38] R. Yoo, A. Romano, and C. Kozyrakis, "Phoenix Rebirth: Scalable Mapreduce on a NUMA System," *Proc. Int'l Symp. Workload Characterization (IISWC),* 2009.

Wenbin Fang received the the BE degree in computer sciences from Nankai University in 2007, and the MPhil degree in computer sciences from the Hong Kong University of Science and Technology (HKUST) in 2010. He is a graduate student in Computer Sciences Department at the University of Wisconsin-Madison. He is interested in parallel and distributed systems, with a focus on performance analysis, design, and implementation issues. He is a student member of the IEEE and the IEEE Computer Society.

Bingsheng He received the bachelor's degree in computer science from Shanghai Jiao Tong University in 2003, and the PhD degree in computer science from the Hong Kong University of Science and Technology in 2008. He is an assistant professor in the Division of Computer Science, School of Computer Engineering, Nanyang Technological Univeristy, Singapore. His research interests are database systems and high-performance computing.

Qiong Luo received the the BS and MS degrees in computer sciences from Beijing (Peking) University, China, in 1997 and 1992, respectively, and the PhD degree in computer sciences from the University of Wisconsin-Madison in 2002. She is an associate professor at the Department of Computer Science and Engineering, Hong Kong University of Science and Technology (HKUST). Her research interests are in database systems, with a focus on architecture-conscious databases, query processing in pervasive computing and sensor networks, and web data management.

Naga K. Govindaraju received the BTech degree in computer science from the Indian Institute of Technology, Bombay, in 2001, and the MS and PhD degrees in computer science from the University of North Carolina at Chapel Hill in 2003 and 2004, respectively. He is currently a senior researcher and manager of the Applications Incubation group in the Extreme Computing Group at Microsoft Corporation. Before joining Microsoft, he was a research assistant professor in the Department of Computer Science at the University of North Carolina, Chapel Hill. His research focuses on the design of efficient parallel algorithms to solve several computational problems in computer graphics, databases, and high-performance computing. He is a member of the IEEE and the IEEE Computer Society.