Soren: Adaptive MapReduce for Programmable GPUs

Reza Mokhtari, Amin Abbasi, Farshad Khunjush, and Reza Azimi

School of Electrical and Computer Engneering
Shiraz University, Shiraz, Iran,
and
School of Computer Science,
Institute for Research in Fundamental Sciences (IPM),
P.O.Box 19395-5746, Tehran, Iran**.
{rmokhtari, aminabbasi, khunjush, azimi}@cse.shirazu.ac.ir

Abstract. In recent years the MapReduce programming model has been widely used for developing parallel data-intensive applications. As a result of its popularity, there exist many implementations of the MapReduce model on different parallel architectures including on massively parallel programmable GPUs. A basic challenge in implementing a MapReduce runtime system is the wide diversity of applications developed based on the model. That means a fixed implementation of the MapReduce runtime system may become suboptimal for some classes of applications. In this paper, we propose an adaptive framework for MapReduce on GPUs which is capable of monitoring key characteristics of applications and dynamically executing them efficiently in one of the three variations of the MapReduce engine it implements. Our preliminary results show that our adaptive method can significantly improve performance for many MapReduce applications (including a 11x performance speedup in one case) compared to a state-of-the-art MapReduce implementation on GPUs.

1 Introduction

In recent years specialized parallel architectures such as GPUs, the IBM Cell Processor, and custom-made accelerators have emerged as an attractive and economically viable solution for accelerating data-intensive applications. However, despite their high peak performance, low hardware costs, and higher power efficiency, these *accelerators* have not yet been fully utilized in mainstream computer systems. The main challenge in using these parallel architectures is that developing efficient codes for them is a non-trivial task. In other words, in order to fully utilize their performance potentials one has to have a detailed knowledge of the micro-architecture implementation. Therefore, a programming environment that is capable of abstracting architectural details and automating the process of program optimization is highly desirable.

** This research was in part supported by a grant from IPM. (No. CS1388-4-10)

The MapReduce programming model introduced by Google [3] has been widely used for data-processing applications both in academia and industry [1]. This model, which is inspired by functional programming models, allows programmers to focus solely on the core computational functions of the application and how the intermediate results should be aggregated; however, it leaves the implementation of data decomposition, task creation, synchronization, and data communication to a runtime system. Since much of the workflow of a parallel program in this model is abstracted away from the programmer, there can be many opportunities for automatic performance optimizations.

Previous research on implementing the MapReduce model for specialized parallel architectures has yielded promising results [10, 9, 5, 7]. A basic challenge in implementing a MapReduce runtime system is the wide diversity of the runtime behavior of the applications developed based on this model. This is primarily attributed to the fact that the MapReduce model makes no assumptions on the semantics or the volume of the intermediate results. As a result, a fixed set of data structures and algorithms for implementing the MapReduce runtime system may not be optimized for all classes of applications that run on top of it. Ideally, a MapReduce runtime system should adapt to different application behaviors automatically by applying specialized optimizations for each class of applications.

In this paper, we present a novel implementation of the MapReduce runtime system for massively parallel programmable GPUs, called Soren. We focus on GPUs because of their ever increasing peak computational power, reaching to several TeraFLOPs, as well as their low cost and availability. However, we strongly believe that many of the ideas and techniques presented in this paper are applicable to other types of specialized parallel architectures.

The contributions of this paper is threefold. First, we identify and resolve serious performance problems in existing MapReduce implementations for GPUs that are primarily due to the mismatch between the behavior of certain classes of applications and the schemes used for managing intermediate results in the available MapReduce engines. Second, we propose and partially implement a framework for a MapReduce runtime system, which is able to monitor the behavior of applications and dynamically adapt to it by using one of its three MapReduce engines. Finally, we propose and partially implement techniques that allow handling arbitrary amount of intermediate results and input data. Our preliminary results suggest that many applications can substantially benefit from an adaptive MapReduce runtime. In one case a 11x performance speedup is achieved while in most other cases a performance improvement of 50% to 150% is observed compared to a state-of-the art implementation of MapReduce on GPUs.

The rest of the paper is organized as follows. First, we provide a background to the MapReduce model and the architectural issues specific to GPUs that may pose a challenge in implementing MapReduce on GPUs. Next, we present the design and implementation of Soren. Then, we present our experimental setup and results. We then compare Soren with other implementations of MapReduce on different types of multiprocessors. We end this paper by presenting our conclusions and some ideas for future work.

2 Background

2.1 MapReduce

The key advantage of the MapReduce model, which is influenced by the functional programming model to a great extent, is that it allows programmers to focus only on the core functions for processing data and aggregation of intermediate results and leave all the machinery that is required to execute their application on a potentially large parallel computer infrastructure to a runtime system. In this model, programmers provide two main functions: (a) a *map* function that processes a subset of input data and *emit*s some intermediate key/value pairs, and (b) a *reduce* fucntion which takes a set of key/value pairs and combines pairs with the same key value into an aggregate result.

The MapReduce runtime system automatically splits the input data into chunks and create many *map tasks* each of which running the user-specified map function on individual input data chunks. The runtime system also creates a number of *reduce tasks* and forwards intermediate results generated by the map tasks to them. Since both the map and reduce functions are usually specified based on a *shared-nothing* assumption, they can run in parallel. The only synchronization required among these tasks is in the form of producer-consumer relationship between the map tasks that generate intermediate results and the reduce tasks that consume them. The reduce tasks can also run in several stages and hence form a hierarchy of tasks, each of which feeding their higher level tasks with partially-reduced results.

2.2 GPU-specific Constraints

Scalability: Current GPU architectures support having hundreds of processing elements and many more hardware threads in order to hide DRAM lantecy. Since in the MapReduce model no parallelism is specified within a map or reduce function, each map or reduce task is naturally assigned to a single GPU thread. That means, unlike in conventional mulit-core environments, there are potentially thousands of map or reduced tasks in flight. Therefore, any implementation of MapReduce basic functions on GPUs must be scalable to a very large number of threads.

SIMD Behavior: GPU threads cannot run completely indepdently in parallel because the underlying hardware implementation for instruction scheduling is partially shared among clusters of cores. As a result, GPU programs execute in a SIMD fashion to some extent. This is in contrast to conventional multi-core systems where threads can execute completely different sequence of instructions independently. The implication of this characteristic for task scheduling in a MapReduce runtime system is that the scheduled tasks must be identical in

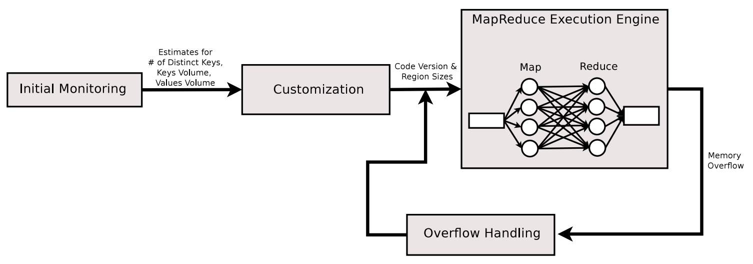

Initial Monitoring
Estimates for # of Distinct Keys, Keys Volume, Values Volume
Customization
Code Version & Region Sizes
MapReduce Execution Engine
Map
Reduce
Memory Overflow
Overflow Handling

Fig. 1. The general workflow of Soren.

terms of the sequence of instruction they execute most of the time. Otherwise, heavy serialization penalties are inflicted to GPU programs due to *code divergence*.

Memory Access and Management: GPUs normally have a DRAM with a modest size of few GBytes. The access latency to DRAM is normally in the range of a few hundred cycles, but the available read/write bandwidth to DRAM is quite larger than a normal DRAM on CPU. Therefore, GPUs employ a large number of threads in order to hide the DRAM access latency by exploiting the abundant bandwidth. Memory is often allocated before a computation kernel is launched on GPU. This is mainly due to lack of proper support for dynamic memory allocation in the existing platforms such as NVIDIA's CUDA or OpenCL.

3 Design and Implementation

There are two key principles in designing Soren. The first one is to have the ability to automatically adapt to different application behaviors. That means the MapReduce runtime system must be capable of monitoring metrics about the applications while they are running and applying required performance tunings automatically.

The second design principle is to allow the MapReduce runtime to handle both input data and intermediate results with arbitrary size. While appropriate mechanisms are provisioned in designing Soren to fully achieve this goal, we have only partially implemented it at this point in time.

Fig. 1 displays the general workflow of Soren designed based on the two abovementioned principles. The first stage in the execution of an application on top of Soren is *Initial Monitoring* where basic characteristics about the application are collected and fed to the next stage, which is *Customization*. In this stage, an appropriate variation of the MapReduce execution engine is selected based on the application characteristics collected in the previous stage. Moreover, the required memory size for different data structures in the MapReduce engine is estimated and properly initialized. Then the *MapReduce Execution Engine* starts processing the input data by creating and executing *map* and *reduce* tasks. This stage can lead to application completion unless an overflow in one of the engine's data structures occurs. In this case an *Overflow Handling* stage is launched in order to re-allocate the required memory space, load existing intermediate results onto it, and resume the execution engine. In the remaining of this section we describe the details of the design and implementation of each of these stages in Soren.

3.1 Soren Execution Engine

First, we present the details of our implementation of the MapReduce execution engine. The key design decision made in this implementation is to collocate intermediate keys and values per each key in memory as they are being emitted by map tasks. This is in contrast to previous designs in which each map or reduce task is assigned its exclusive memory area [5]. We discuss the advantages and drawbacks of this design principle shortly.

The GPU memory in Soren is partitioned into four contiguous regions: the input data region, the hash table, the key space, and the values space. The size of each region is initially set using an estimation method. However these regions can grow or shrink dynamically based on the memory usage requirements of applications. The details of the mechanisms for estimating region sizes and dynamically resizing regions is described in Section 3.3.

Having contiguous memory regions for each data structure is not a fundamental issue in our design. However, it allows us to implement simple dynamic memory allocation schemes that are specific to the type of data that is maintained in each region. An alternative is to use a general, GPU-based dynamic memory allocation scheme such as Xmalloc [6] for all types of meta-data.

Fig. 2 shows the major data structures in Soren each of which located in one of the four regions mentioned above. The central structure is a hash table which is used for looking up the $\langle key, value\rangle$ pairs being emitted by map tasks. The hash key for lookup into the hash table is built based on the key provided by the *emit* operation. Each entry in the hash table contains pointers to locations inside the *keys space* and the *values space* where the actual values for both keys and their corresponding values are stored. Since it is not easy to use dynamic memory allocations in current GPU technologies, we use a two-dimensional matrix for storing the hash table entries. The elements in each column of this matrix share the same key hash values. Storing hash elements in a column-major fashion allows concurrent threads to exploit memory coalescing features in GPUs.

Whenever a *map* task emits a $\langle key, value\rangle$ pair function, the Soren runtime searches the specified key in the hash table. If the key does not exist in the hash table, a new entry is allocated and the provided key and value are inserted into the *keys space* and *values space* respectively. However, if the key already exists in the hash table, the value is simply appended into the list of values designated for the key in the *values space*. In both cases, synchronization among concurrent GPU threads is required as there might be a race for an entry in the hash table. This is in contrast to the previous design [5] where each map task has its own preallocated memory area to which intermediate results can be emitted without synchronization. Although we expect the overhead of synchronization to slow down the *map* phase, our experimental results indicate that the performance gain

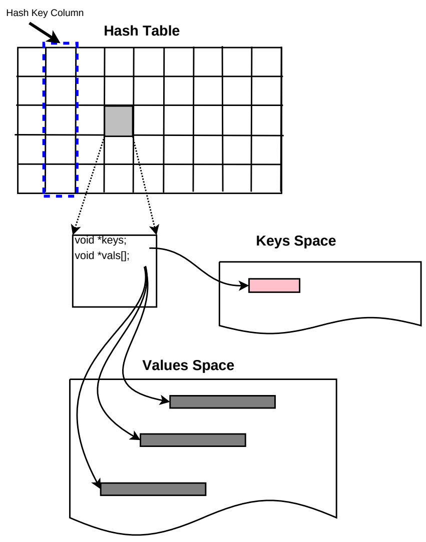


Fig. 2. The major data structures of Soren MapReduce runtime. A column-major hash table allows Soren to benefit from memory coalescing as new rows are appended to the table.

of removing potentially expensive *sort* phase can easily outweigh the overhead for certain types of applications.

In order to provide mutual exclusion a lock is assigned for each column of the hash table. Acquiring and releasing locks are implemented using *atomic instructions* available in most modern GPUs. By executing an atomic instruction a thread can read, modify, and write back a location in GPU memory in a single step without allowing other threads to either read or modify the specified location. The following codes illustrate the acquire and release operations.

```
acquire_lock:
  do {
    oldState = atomicExch(&locks[hIndex], 1);
  } while(oldState == 1);

release_lock:
  atomicExch(&locks[hIndex], 0);
```

Incremental Combine An important advantage of collocating values of each key as being emitted is that it allows for partial reduction of the intermediate results during the *map* phase. As the *reduce* function provided by the user may not be associative, an associative variation of it must be provided by the user to allow to *combine* intermediate results incrementally. That is to partially reduce values emitted for a key into a single value.

There are two main benefits in enabling incremental combine. First, it can control the footprint of the intermediate results in the GPU memory by constantly shrinking a large number of values into one. Otherwise, some applications

Metric Name	Description	Usage
$KeyValPairs$	the number of $\langle key, value\rangle$ pairs emitted	estimating the Hash Table size
$KeySize$	the volume of the emitted $keys$	estimating the size of the Keys Space
$KeySize$	the volume of the emitted $values$	estimating the size of the Values Space
$DistinctKeys$	the number of distinct keys emitted	code variation selection

Table 1. The metrics collected during the *Initial Monitoring* stage and their usages.

may overflow the GPU memory by emitting too many $\langle key, value\rangle$ pairs during their map phase. Moreover, setting a limit on the volume of the intermediate results may enable optimizations through caching, which is not yet implemented in Soren. Second, having a *combine* function allows the intermediate results to be reduced with a higher-level of concurrency especially for applications that do not emit many distinct keys. Writing an incremental *combine* function is not a substantial burden on programmers as it can be easily derived from existing reduce function. In fact, if the *reduce* function is associative, it can be directly used as the incremental *combine* function. Other well-known MapReduce implementations also suggested adding a *combine* function to enable many useful performance optimizations [12].

3.2 Initial Monitoring

The *Initial Monitoring* stage executes the map stage of an application similar to the normal execution with two differences. First, it only processes a small fraction of the input data called $TrainingData$. Secondly, it is used only to collect performance metrics and therefore, it does not actually store the emitted values. The basic assumption is that in general for data-processing application developed based on the MapReduce model a small part of the input data reveals sufficient information to speculate the behavior of applications. In other word, we expect key behavioral characteristics of applications to be more or less uniform across the entire input data

Currently, four metrics are collected in the initial monitoring. The description and usage of each of these metrics is mentioned in Table 1. The actual value of these metrics over the entire execution of an application is estimated by linear projection, i.e., multiplying the collected metrics to the ratio of total input data size divided by the size of the $TrainingData$. In the current implementation the $TrainingData$ is set to be the first 20% of the input data.

3.3 Handling Overflows

An *Overflow Handling* mechanism is triggered whenever the actual size of one of the main data structures exceeds the size predicted in the initial monitoring stage. If a thread runs out of space during emitting the results, it saves its current state in a global array, atomically increments a global *overflow flag*, and executes no-ops. All of the threads check the *overflow flag* before attempting to emit. If the *overflow flag* reaches a certain threshold, all active threads return causing the kernel to terminate and the control of the program is returned back to the code running on CPU. The overflow handling mechanism copies the existing content of the data structure that caused overflow into an area on CPU memory and deallocates its memory on GPU. Then, it reallocates the data structures with a larger size on GPU memory and copies back the existing content onto it. After completing overflow handling a new GPU kernel is launched to resume the *map* phase from the point that overflow occurred.

Although the overflow handling mechanism is fairly costly, we expect overflow to be an infrequent event. This is mainly because the estimates produced at the end of the initial monitoring phase is usually accurate.

3.4 Customization

The main task in the customization stage is to select a variation of the implementation for the MapReduce runtime system that best suits an application. There are three different variations in our current implementation. The basic design that is described so far is the default variation. The other variations deal with two special cases: *No-Reduce* and *Small Number of Keys*. Next, we describe the details of the key differences between these two variations and the default variation as well as the type of applications that benefit from each of these variations.

No-Reduce Case: This is a special case for MapReduce applications that have an empty *reduce* function and the entire computation for these applications is done during the *map* phase. Consequently, there is no need for sorting or merging the intermediate results emitted by the *map* tasks. An example of such applications is Matrix Multiplication.

Since the reduce phase is empty in this case, paying the extra contention costs for collocating $\langle key, value\rangle$ pairs of same keys is not justified anymore. As a result, we revert to a variation of the Mars implementation of the MapReduce framework in this case [5]. That means, each thread emits its intermediate results in a specific address range which is precomputed by a prefix sum. The main downside of this approach is that it requires a *MapCount* provided by programmers. We intend to extend our design to remove this requirement by estimating per-thread space during the *Initial Monitoring* stage.

Small Number of Keys: This special case is for MapReduce applications that emit small number of distinct keys. That means, the $\langle key, value\rangle$ pairs are clustered around a small number of distinct keys with each cluster containing potentially very large number of pairs. Examples of such applications include Histogram. This special case is activated whenever the $DistinctKeys$ metric collected in the initial monitoring stage is less than a configurable threshold. We set this threshold in our design to be 1000.

Having a small number of distinct keys creates substantial contention in our default design as we insert the $\langle key, value\rangle$ pairs for each key into its corresponding cluster as they are emitted. This is because a potentially large number of threads race to insert $\langle key, value\rangle$ pairs into a small number of clusters. The serialization resulted by eliminating this race is so costly that it may outweigh the benefits of removing the extra sort or merge phase.

Workload	**Description**	**Small dataset**	**Medium dataset**	**Big dataset**
Word Count	Counts the number of occurrences of each word in a document	10MB	50MB	100MB
Histogram	Computes the RGB histogram of an image	3MB	6MB	10MB
PageViewCount	Counts the number of distinct page views from web logs	5MB	40MB	65MB
Distribute Grep	Counts lines in all files in that matches a regular expression	10MB	50MB	100MB
Matrix Multiplication	Calculates the product of two matrices	500×500	1000×1000	2000×2000

Table 2. Applications description and their dataset size.

Another issue with a small number of keys is that having too few clusters limits parallelism at the reduce stage as there are too few reduce tasks each of which responsible for a potentially very large amount of data. To resolve this issue MapReduce runtime systems often break sorted intermediate results into *chunks* to allow more reduce tasks run concurrently. We adopt a similar idea to solve both of the above-mentioned problems.

For applications with too few distinct keys we augment each key with the $ThreadBlockId$ of the running thread to compose a new hashing key in the form of $\langle key, ThreadBlockId\rangle$. As a result, the cluster of values for each distinct key is broken into many smaller subclusters. This relaxes the contention problem as a smaller number of threads compete for adding pairs to each subcluster. Once the *map* phase is finished, the *incremental combine* operation provided by the user is applied on values emitted into each subcluster to produce a single $value$. Then a simple *merge* operation combines single values of clusters with the same *key* but different $ThreadBlockIds$ into a single *super-cluster* with the maximum size equal to the number of thread blocks. Using *incremental combine* on a potentially large number of clusters provides substantial parallelism. Finally, the *reduce* operation is applied on the fairly small super-clusters.

4 Experimental Setup

We performed our experiments on a PC with an NVIDIA Geforce GTX 480 GPU [8] with 480 cores each running at 1.4GHz and 1.5GBytes of DRAM with a maximum bandwidth of 177GByte/s. We used NVIDIA CUDA 3.0 on Linux as a base for the implementation of Soren. We rely on atomic operations that are available in CUDA compute capability 1.1 or higher.

We ported several applications to evaluate the performance of Soren. We selected three programs from Mars sample applications: *Word Count* and *Page View Count* that have both *map* and *reduce* functions, and *Matrix Multiplication* that only have a *map* function. Two more applications are implemented, *Histogram* and a simplified version of *Distributed Grep*. In addition, three input data sizes are used to evaluate our system for all applications. Table 2 shows these data sizes for each workload. Next, we briefly describe the functionality of each application and how they are organized in the MapReduce model.

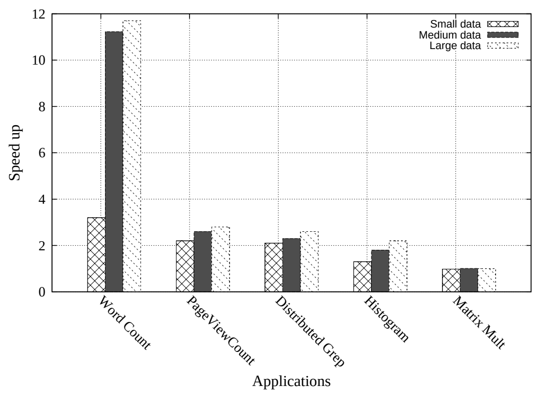


Fig. 3. Applications speed-up in Soren compared to Mars.

Word Count: Each *map* task processes several lines in a document. For each word in those lines, it generates an intermediate pair in the $\langle word, 1\rangle$ form. Each *reduce* task takes a key and sums up the emitted values.

Page View Count: Input file contains many log entries each of which is a 3-ary tuple $\langle URL, IP, Cookie\rangle$: the URL of the accessed page, the host IP, and the cookie generated when accessing the page respectively. This application has two passes of MapReduce execution. The first one removes all of the duplicate tuples with same IP address. The second one counts the number of distinct IP addresses that accessed the page. In the first MapReduce pass, each *map* task takes one line of the input file and outputs an intermediate pair in the $\langle\langle URL, IP\rangle, null\rangle$ form. No *reduce* and phase is required in this pass. In the second pass, each *map* task takes a key generated from the first MapReduce pass and outputs pairs in the $\langle URL, 1\rangle$ form, and each *reduce* task sums up the values for each URL.

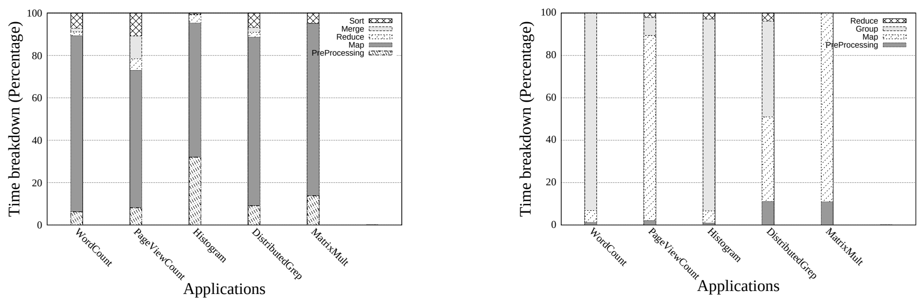
Histogram: Each *map* task gets a portion of an image and outputs an intermediate pair with in the $\langle RGBValue, 1\rangle$ form. Each *reduce* task takes an RGB value as the key and sums up all the values for it. An additional *sort* phase is required to sort the resulting histogram based on the RGB values.

Distributed Grep: Each *map* task takes a portion of an input file and outputs a $\langle word, 1\rangle$ pair if a word in the line matches a given expression. The *reduce* function simply sums up pairs with same keys to produce the set of words that matched the given pattern and their frequencies.

Matrix Multiplication: Each *map* task multiplies a row of the first matrix by a column of the second matrix and generates a pair with the position of the result entry as the key and its content as the value.

5 Results

Fig. 3 illustrates the achieved speed-up of the test applications compared to Mars with varied dataset sizes. As shown in this figure, Word Count exhibits the best speed-up due to its huge number of distinct keys which allows Soren to distribute



(a) The time breakdown of Soren. (b) The time breakdown of Mars.

Fig. 4. Comparison of time breakdowns for Soren and Mars on the five applications with large dataset.

new $\langle key, values\rangle$ pairs evenly across the hash table and thus minimizing the lock contention. Moreover, the grouping of the intermediate results in Soren takes far less time compared to Mars. This is mainly because in Mars grouping values with the same key needs pointers and actual data to be sorted. Instead, in Soren this task is carried out during the *map* phase.

As depicted in Fig. 3, Page View Count and Distributed Grep applications achieve less speed-up because their number of distinct keys are fewer than that in Word Count. This causes more contention when the *map* function inserts new pairs in the hash table. Furthermore, the total number of pairs emitted by the *map* function in these two applications is considerably less than that in Word Count. For applications with small number of keys Soren and Mars perform similarly. However, the gap between Soren and Mars grows as the number of distinct keys increases.

Histogram achieves the least speed-up among the applications with both *map* and *reduce* phases. It is worth mentioning that the number of distinct keys in Histogram is fairly small (between 0 to 768). This incurs substantial contention on locks that protect hash table entries. We also measured the execution time of Histogram in Soren when the adaptation was disabled. In this case, Soren performs worse than Mars due to having a small number of keys. With the adaptation enabled, however, Soren increases the number of distinct keys by augmenting the keys with $ThreadBlockId$.

Matrix Multiplication uses the *map_only* workflow both in Mars and Soren which dictates to Soren not to use the hash table. Consequently, the runtime system uses a simple data structure similar to that in Mars to store intermediate results. This leads to an execution time that is exactly the same as in Mars.

Although achieving high performance in Soren depends on a fast implementation of atomic operations on GPUs, our experiments on a GT200-based GPU (with a supposedly slower implementation of the atomic operations) show only a slight drop in speed-up compared to the GTX-480 GPU (the detailed comparison is omitted due to space limits). We attribute Soren's performance stability to its optimizations that prevents high contentions in memory accesses.

Fig. 4(a) shows the breakdown of execution time in Soren. As expected, the *map* phase is responsible for a substantial portion of the execution time. There are several reasons to this; first, most of the grouping that needs to be applied to $\langle key, value\rangle$ pairs is done during the *map* phase. Secondly, the *incremental combine* operation starts in the middle of the *map* phase leading to a longer *map* phase and a shorter *reduce* phase. Finally, it should be noted that the time shown for the *map* phase in Fig. 4(a) also includes the time for the *Initial Monitoring* stage where 20% of the input data is used for estimating application metrics.

Another reason for having a shorter *reduce* phase is that in Soren this phase can be parallelized more effectively. The reason for this is that due to constantly applying the *incremental combine* operation, the total number of values that remain for each key by the end of the map phase is limited. As a result, the intermediate results that are fed to the *reduce* tasks are fairly balanced. Therefore, no single *reduce* task can substantially slow down the entire phase.

The *merge* phase is also short primarily because all it does is to $\langle key, value\rangle$ pairs from the cells of the hash table into a simple list data structure which is used to deliver the output results to the user. Operations in the *merge* phase can be done using a lock-free scheme. However, a prefix sum must be computed before the copy operation.

Finally, in the *sort* phase, we only sort the pointers to the $\langle key, value\rangle$ pairs to eliminate final data copy overhead. Similar to Mars, Soren uses a bitonic sort algorithm on the GPU [4] for sorting the pointers. However, in Mars the sort function must be called several times not to sort not only the pointers but the actual data as well.

Fig. 4(b) shows the same breakdown for Mars. Unlike Soren, Mars has a fast *map* phase as its *emit_intermediate* only stores intermediate results in pre-allocated spaces in a lock-free scheme. Instead, the *group* phase in Mars involves several levels of sort operations to sort pointers, keys, and values in a sequence. This potentially expensive sequence of sorts is required because the list of values to be fed to the *reduce* function for each single key must be contiguously collocated.

6 Related Work

Phoenix is a widely used MapReduce implementation for homogeneous chip-multiprocessors [11, 12]. The main focus in the design of Phoenix is on achieving scalability through NUMA-aware memory management. Each *map* thread emits intermediate results on a space allocated on the closest memory module to the CPU the thread is scheduled on. In contrast, the GPU DRAM is not NUMA and the amount of local memory for each streaming multiprocessor is not sufficiently large to allow having per-thread space for intermediate results.

In terms of adaptability, Phoenix attempts to tune some parameters such as map chunk size based on applications behavior. However, it is the programmer's responsibility to provide the Phoenix runtime with information about the application such as the estimated size of the emitted keys and the volume of intermediate results whereas in Soren these information are automatically collected during the initial monitoring phase.

Similar to Soren, in Phoenix the $\langle key, value \rangle$ pairs for each are collocated online as they are being emitted. As a result, Phoenix can also support the *incremental combine* operation in order to reduce the volume of non-local memory traffic. In Soren, however, the *incremental combine* operation is used for different purposes, i.e., (i) to dynamically collapse the intermediate results emitted for each key to prevent the need for allocating extra memory dynamically (which is an expensive operation on GPUs), (ii) to reduce the total footprint of the MapReduce runtime on GPU memory.

Merge [7] is a MapReduce for heterogeneous multiprocessors that provides support for dynamic adaptation. However, the adaptation in Merge is primarily focused on customization of the user-provided functions either through smart code selection or automatic code synthesis based on generic code patterns.

MapReduce is also implemented on the IBM CellBE processor [2, 10, 9]. The architectural difference between CellBE and GPUs is the lack of fine-grained, hardware-level global memory access on CellBE and the explicit management of per-core local memory. These characteristics force the *map* or *reduce* tasks to work on local memory most of the time. As a result, the intermediate results must be merged and sorted at the end of the *map* phase. Results from [9] show that the overhead of sorting can be potentially substantial. However, due to architectural differences between CellBE and GPUs a head-to-head comparison between this overhead versus the contention overhead in Soren is not possible.

Mars is the most notable MapReduce implementation for GPUs [5]. In fact, Soren inherits much of the infrastructure code from Mars. The details of the differences between the two implementations are mentioned in Section 3. To summarize, Soren's default implementation collocates the intermediate results for each key as they are being emitted in order to remove the overhead of an extra *sort* phase. Moreover, Soren is capable of adapting to application behavior by automatically selecting an efficient implementation of MapReduce.

7 Concluding Remarks

In this paper, we present an adaptive framework for MapReduce for GPUs. Our framework is capable of monitoring the behavior of applications and dynamically selecting a suitable variation of the implementation of the runtime system for each application. Moreover, by using an incremental combine phase and an overflow handling mechanism, we are able to handle potentially large volumes intermediate results. The evaluation of our framework using standard MapReduce benchmarks shows that substantial performance improvement can be achieved compared to a state-of-the-art implementation of MapReduce on GPUs.

There are several steps that we would like to take as our future work. First, we believe a more diverse set of applications must be included in our evaluation to be able to investigate the effectiveness of our adaptive design. This process may expose other classes of applications that warrant developing new variations of

MapReduce to be included in our framework. Secondly, we intend to extend our overflow handling mechanism to allow us to handle arbitrary sizes of input data. Finally, we would like to do a more thorough analysis of the micro-architectural behavior of GPU under our implementation to be able to come up with further fine-grained optimization techniques such as the removal of branch divergence and improving memory coalescing.

References

1. Apache. Hadoop. http://wiki.apache.org/hadoop/PoweredBy.
2. M. de Krujif and K. Sankaralingam. MapReduce for CellBE architecture. *IBM Journal of Research and Development*, 53(5):10:1–10:12, 2009.
3. Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design and Implementation*, San Francisco, CA, 2004.
4. N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTeraSort: high performance graphics co-processor sorting for large database management. In *SIGMOD 06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 325–336, New York, NY, USA, 2006.
5. B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: A mapreduce framework on graphics processors. In *Proceedings of the Intl. Conference on Parallel Architectures and Compilation Techniques (PACT)*, Toronto, Canada, October 2008.
6. Xiaohuang Huang, Christopher I. Rodrigues, Stephen Jones, Ian Buck, and Wen mei Hwu. Xmalloc: A scalable lock-free dynamic memory allocator for many-core machines. In *Proceedings of the 10th IEEE International Conference on Computer and Information Technology*, 2010.
7. M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng. Merge: A programming model for heterogeneous multi-core systems. In *Proceedings of the Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Seatle, WA, March 2008.
8. NVidia. Nvidia's next generation cuda compute architecture: Fermi version 1.1. 2009.
9. Anastasios Papagiannis and Dimitrios S. Nikolopoulos. Rearchitecting mapreduce for heterogeneous multicore processors with explicitly managed memories. In *Proc. of the 39th International Conference on Parallel Processing (ICPP)*, San Diego, CA, September 2010.
10. M. Mustafa Rafique, Benjamin Rose, Ali R. Butt, and Dimitris Nikolopoulos. CellMR: A framework for supporting mapreduce on asymmetric cell-based clusters. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Rome, Italy, May 2009.
11. Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *Proceedings of the 13th Intl. Symposium on High-Performance Computer Architecture (HPCA)*, Phoenix, AZ, February 2007.
12. Richard M. Yoo, Anthony Romano, and Christos Kozyrakis. Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system. In *in Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, Austin, TX, October 2009.