Moim: A Multi-GPU MapReduce Framework

Mengjun Xie and Kyoung-Don Kang
Department of Computer Science
State University of New York at Binghamton
{mxie3,kang}@cs.binghamton.edu

Can Basaran
Middle East Technical University
Northern Cyprus Campus, Turkey
basaran@metu.edu.tr

Abstract

MapReduce greatly decrease the complexity of developing applications for parallel data processing. To considerably improve the performance of MapReduce applications, we design a new MapReduce framework, called Moim, which 1) effectively utilizes both CPUs and GPUs (general purpose Graphics Processing Units), 2) overlaps CPU and GPU computations, 3) enhances load balancing in the map and reduce phases, and 4) efficiently handles not only fixed but also variable size data. We have implemented Moim and compared its performance with an advanced multi-GPU MapReduce framework. Moim achieves $20\% - 90\%$ *speedup for different data sizes and numbers of the GPUs used for data processing.*

1 Introduction

MapReduce [7, 12] is a parallel programming model that greatly decreases the complexity of developing big data applications, such as web data analysis and scientific applications. Ideally, a user only needs to write two functions, map and reduce functions, which specify how to convert input <key, value> pairs into intermediate <key, value> pairs and how to reduce them into final < Skey, value> pairs in the map and reduce phase, respectively. The MapReduce runtime takes care of data partition, scheduling, and fault tolerance in a transparent manner.

However, MapReduce has several limitations. Although it is carefully designed to leverage inter-node parallelism in a cluster of commodity servers, it is *not designed to exploit the intra-node parallelism* provided by heterogeneous parallel processors, such as multicore CPUs and GPUs. In MapReduce, the intermediate <key,value> pairs produced by the mappers (map tasks) of a job are shuffled to one or more reducers (reduce tasks) via hashing based on the keys. Unfortunately, this approach may result in serious *load imbalance* among the reducers of a job, as the distribution of keys, e.g., URLs or natural language words, in certain applications, e.g., web document analysis or natural language processing, is highly skewed [17]. As the speed of a parallel job consisting of smaller tasks is determined by the slowest task, a larger degree of load imbalance may translate to a longer delay. Also, it is *challenging to efficiently process variable size data*, such as words in web documents or e-books.

To tackle these challenge, we design a new MapReduce framework, called Moim[1], which provides a number of new features as follows:

1. In each MapReduce phase, Moim efficiently utilizes the parallelism provided by both multicore CPUs and GPUs.
2. It overlaps CPU and GPU computations as much as possible to decrease the end-to-end delay.
3. It supports efficient load balancing among the reducers as well as the mappers of a MapReduce job.
4. The overall system is designed to efficiently process not only fixed but also variable size data.

Considerable work has been done to improve the performance of MapReduce by efficiently utilizing CPU cores [22], GPU cores [14, 15, 2], and multiple GPUs [9, 23]. However, these novel MapReduce frameworks [22, 14, 15, 2, 9, 23] do not aim to efficiently utilize heterogeneous processors, such as a set of CPUs and GPUs. Neither do they support the key features of Moim summarized above and discussed in detail in the remainder of the paper.

For performance evaluation, we have compared Moim with an advanced multi-GPU MapReduce framework, called GPMR [23], representing the current state of the art. We have used two widely used benchmarks, word count and matrix multiplication, which

[1]It means a group or herd in Korean.

process variable and fixed size data, respectively. The experimental results demonstrate Moim's effectiveness: Moim achieves 20% − 90% speedup for different data sizes and numbers of the GPU cores employed for data processing in a cluster that consists of 128 CPU cores, 6,144 GPU cores, 128 GB memory, and 8 TB storage in total.

The remainder of this paper is organized as follows. The design of Moim is described in Section 2. The performance evaluation results are discussed in 3. Related work is discussed in 4. Finally, Section 5 concludes the paper and discusses future work.

2 System Design

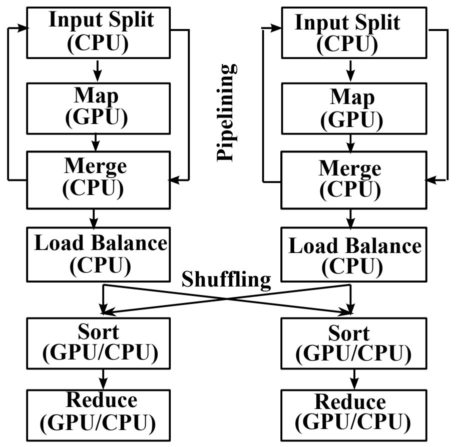

Figure 1. Workflow in Moim

Figure 1 shows the MapReduce phases of Moim. In the input split phase, input data are split into smaller pieces in multicore CPUs. We run the input split phase in CPUs, as most GPUs cannot directly access I/O devices such as disk.

In the map phase, the input data splits are processed by the mappers running in GPUs as shown in Figure 1. By processing the user-defined map() function, the mappers produce intermediate <key, value> pairs.

In the merge phase, intermediate <key, value> pairs having the same key are combined into one <key, value> pair. In Moim, the merge phase is processed in parallel using multiple CPU cores to decrease the merge delay and network transmissions necessary to transfer the <key, value> pairs to the reducers. Notably, the input split, map, and merge phases are overlapped via application-layer pipelining to decrease the delay as illustrated in Figure 1.

When the map and merge phases complete, the load balancing phase involving network I/O runs in CPUs. In this phase, Moim derives a load balancing plan to distribute load among the reducers as evenly as possible. Finally, the <key, value> pairs are optionally sorted and finally reduced.[2] A sort/reduce task runs in a GPU, if the <key, value> pairs to be processed can be accommodated in the GPU memory. Otherwise, it runs in the host. Thus, different chunks can be concurrently processed in CPUs and GPUs to decrease the processing delay. In this section, these steps are discussed in detail.

2.1 Parallel Input Split Phase

In MapReduce [7, 12], by default, input data is split at byte boundaries based on the total size of the input. However, this approach to splitting is insufficient for many applications, because record boundaries, e.g., English word boundaries, have to be respected. In such cases, the application is required to implement a split() function to tokenize the input respecting record-boundaries and provide a record-centric view of the input splits to the individual map tasks [13].

We observe that it is relatively straightforward to split raw input consisting of fixed-size data items. For example, a split() function can assign a constant number of floating point numbers to each mapper for a scientific application. However, it is non-trivial for an application to efficiently split variable size data. Moim provides a number of *parallel split functions and APIs* (Application Programming Interfaces) for several types of fixed and variable size data to alleviate the burden of application developers.

In our approach to splitting variable size data (especially text data), each CPU core is assigned a constant number of lines of input data (delimited by newline characters). Each core sequentially scans the assigned text lines to detect the beginning and length of each token, i.e., key. Each node that has one or more CPU cores runs this process to convert the raw data into tokens. We take this approach to take advantage of both inter-node and intra-node parallelism to speed up parallel split using the CPU cores in a cluster.

For variable size data, the parallel input split phase is performed in O(n/p) time where n is the input size in terms of the total number of bytes and p is the total number of the CPU cores in a cluster. In contrast, input split using a single core in each node finishes in O(n/c) time where c is the total number of CPUs in a cluster. (This is a common practice in MapReduce and Hadoop applications [2].) Thus, Moim can achieve considerable speedup when $p > c$.

[2]Moim requires an application to indicate whether or not it needs sorting, similar to MapReduce and Hadoop [7, 12].

After the parallel split phase, the Moim runtime evenly distributes the keys to the mappers in each node for enhanced load balancing, while maintaining the locality of data in a transparent manner. In fact, it is possible to evenly distribute tokens among all the mappers in the entire cluster. However, we do not take the approach, since it may require additional network transmissions orders of magnitude slower than computations.

2.2 Map and Merge Phase

In Moim, each map task processes the assigned data chunk by running the user specified map() function using GPU threads, and produces intermediate <key,value> pairs. At the end of the map phase, each GPU optionally sorts the produced <key,value> pairs depending on the application. For example, no sorting is necessary in matrix multiplication. However, sorting is needed to combine duplicate <key,value> pairs that have the same key into a single <key,value> pair in the merge phase in word count. If many <key,value> pairs can be combined together, the amount of the data to be shuffled, i.e., transferred, to the reducers can be decreased considerably. For sorting and merging, Moim uses the highly optimized *parallel sorting and merge* functions provided by Thurst [3]. In this way, it leverages both inter-node and intra-node parallelism to efficiently execute the map() function and combine the intermediate <key,value> pairs.

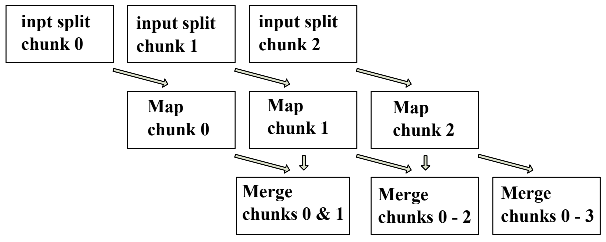

Figure 2. Three Stage Pipelining

As GPUs have finite memory and most of them lack virtual memory support, we may not be able to process the entire input data at once. In general, we instead need to divide data into chunks and process them iteratively. To (partially) hide the delay for iterative executions, we *pipeline and overlap* the input split, map, and merge phases executed in the CPU, GPU, and CPU, respectively, as shown in Figure 2. This technique, called *3-stage pipelining* in this paper, can efficiently deal with large data by overlapping considerable part of the parallel input split, map, and combine phases.

2.3 Load Balancing among Reducers

This step runs on CPUs, as it involves network communications to transmit the intermediate <key, value> pairs to the reducers. Instead of using MapReduce's load balancing scheme [7, 12], we design a new load balancing algorithm by adapting the sample sort algorithm (also called parallel sorting by regular sampling) [16] and a bin packing heuristic. Our load balancing method is summarized in Algorithm 1. Also, it is illustrated via an example in Figure 3.

```
Algorithm 1 Load Balancing among Reducers
input: intermediate <key, value> pairs
output: <key, value> pair assignments to reducers
1: Each node picks a small fraction of the intermediate
   <key, value> pairs as samples at regular intervals.
2: Each node sends its samples to the predetermined
   master node in the cluster.
3: The master sorts the samples using a parallel sort-
   ing algorithm.
4: The master multicasts the sorted global samples,
   $(s_1, s_2, ..., s_\ell)$ to the slaves.
5: Using the sorted samples, each node computes the
   size, i.e., the number of <key,value> pairs, of each
   partition where partition $i$ $(1 \le i \le \ell + 1)$ is a col-
   lection of <key, value> pairs whose keys are bigger
   than $s_{i-1}$ but smaller than or equal to $s_i$ where
   $s_0 = -\infty$ and $s_{\ell+1} = \infty$.
6: Each node sends the sizes of its partitions to the
   master that computes the aggregated global mag-
   nitude of each partition.
7: The master determines which partitions will be as-
   signed to which reducers, i.e., bins, using a modi-
   fied bin packing heuristic for load balancing.
8: The master multicasts the load distribution infor-
   mation to the salves, which in turn shuffle their
   partitions to the destination nodes following the
   load balancing decisions made by the master.
```

In Step 1 of Algorithm 1, we borrow an idea from sample sort. Specifically, Moim takes samples at regular intervals as shown in Figure 3 (a), similar to the sample sort algorithm [16]. For example, Moim can select 0.1% of the intermediate <key,value> pairs by selecting 1 sample at every 1000 <key,value> pairs. In this paper, Moim determines the sampling interval to ensure that all the samples collected from every node can be stored together in the master's available memory.

In Steps 2 - 4, we modify the sample sort algorithm for load balancing purposes. In the original sample sort [16], each node broadcasts its samples to all the

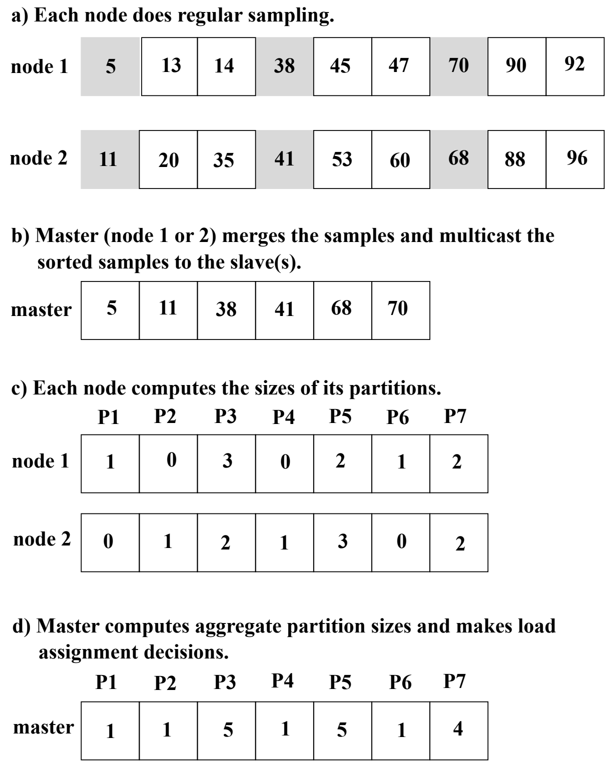
a) Each node does regular sampling.
node 1
5 13 14 38 45 47 70 90 92
node 2
11 20 35 41 53 60 68 88 96
b) Master (node 1 or 2) merges the samples and multicast the sorted samples to the slave(s).
master
5 11 38 41 68 70
c) Each node computes the sizes of its partitions.
P1 P2 P3 P4 P5 P6 P7
node 1
1 0 3 0 2 1 2
node 2
0 1 2 1 3 0 2
d) Master computes aggregate partition sizes and makes load assignment decisions.
P1 P2 P3 P4 P5 P6 P7
master
1 1 5 1 5 1 4

Figure 3. Load Balancing Example (Either node 1 or node 2 is the master.)

other nodes in the cluster. In contrast, in Step 2 of Algorithm 1, each node transmits its samples to the master node that is pre-selected based on, for example, the reliability of the node to avoid all-to-all broadcasts in the sample sort algorithm. In Steps 3 and 4, the master sorts the samples (Figure 3 (b)) using a parallel sorting function provided by Thrust [3]. It multicasts the sorted samples to the slaves.

In Step 5, each node computes the number of the <key, value> pairs in each partition as illustrated in Figure 3 (c). Specifically, each CPU core in each node does a binary search at a time for an intermediate <key, value> pair over the sorted samples to find to which partition this <key, value> pair belongs. If there are m keys in the samples, this search takes $O(\frac{n}{p}\lg m)$ time when there are n <key, value> pairs and p available CPU cores. In Step 6, each slave node reports the sizes of its partitions to the master.

In Step 7, based on the information received from the slaves, the master computes the global size of each partition, i.e., the total number of the <key, value> pairs in the partition counted across the entire cluster of the nodes as illustrated in Figure 3 (d). As load balancing is an NP-complete problem, we apply a modified bin packing heuristic in Step 7 rather than trying to find an optimal algorithm. Specifically, we have modified the worst fit decreasing (WFD) method for bin packing to 1) sort the partitions in descending order of their global sizes, 2) assign the next partition to an empty reducer node (if any) that has been assigned no <key,value> pairs yet, and 3) assign the next partition to the reducer node that currently has the smallest number of <key, value> pairs to process, if there is no empty reducer node anymore. Note that we have changed the WFD heuristic by adding the second step, since our purpose is not minimizing the number of the reducer nodes (i.e., bins), but load balancing among the available reducers. In Step 8, each node actually shuffles its data to the destination reducers determined in Step 7.

2.4 Reduce Phase

In this phase, the reducers execute an optional sorting step. A user is required to specify whether or not sorting is required and provide a cmp() function to compare keys, similar to MapReduce and Hadoop [7, 12], if comparison-based sorting is necessary for the application. After the optional sorting of <key, value> pairs, the user-specified reduce() function is executed by each reducer.

A reducer node assigns <key,value> pair partitions to its GPU(s). If one partition can be accommodated in the memory of a GPU as a whole, it is processed in the GPU. Otherwise, it is processed in the host using the CPU cores. Hence, if there are multiple partitions of intermediate <key, value> pairs, they can be processed in CPUs and GPUs simultaneously, because different partitions have different keys due to our partitioning and load balancing methods described in Algorithm 1.

In summary, Moim supports 1) parallel split and load balancing among mappers, 2) parallel processing of map tasks using a large number of GPU threads, 3) 3-stage pipelining and overlapping of CPU and GPU computations, iii) parallel combining of <key,value> pairs using multicore CPUs, 4) enhanced load balancing among the reducers, and 5) parallel processing of reduce tasks using CPU and GPU threads.

3 Performance Evaluation

In this section, the experimental settings for performance evaluation and performance results are discussed.

Job	Input	Split	Map	LB	Sort	Reduce
WC	variable	yes	yes	yes	yes	yes
MM	fixed	no	yes	no	no	no

Table 1. Properties of the benchmarks (LB = Load Balancing)

Job	Small	Medium	Large
WC	500M words	800M words	1B words
MM	2048 * 2048	4096 * 4096	10240 * 10240

Table 2. Input Data Size of Word Count (WC) and Matrix Multiplication (MM)

3.1 Experimental Settings

We use GPMR [23]−an advanced open-source multi-GPU MapReduce framework−as the baseline for performance comparisons. We use the popular matrix multiplication and word count benchmarks, since they represent fixed-size and variable-size data applications, respectively. Table 1 summarizes their properties discussed in the following.

- *Word Count:* The purpose of this benchmark is to count the frequency of each word's occurrences in the given input. The output is <word, frequency> pairs sorted in non-ascending order of frequencies. This application needs to run every phase as indicated in Table 1.
- *Matrix Multiplication:* The inputs are two matrixes A and B, and output is multiplication of two matrices. As the size of input data items that are floating point numbers is fixed, no input split phase for tokenization is needed. A user only has to specify the map function, since no other phase is needed for this benchmark as described in Table 1. (The split phase in Table 1 means splitting variable size data into tokens.)

We have implemented matrix multiplication and word count using CUDA [19] and Thrust [3], respectively. CUDA supports GPU programming. Thrust is a parallel programming library that supports heterogeneous parallel programming across CPUs and GPUs. In addition, we support network communications among the nodes in a cluster via MPI (Message Passing Interface).

In this paper, we used a mini-cluster of 4 nodes connected by a 1 Gbps switch for experiments. Each node has 32 CPU cores (two AMD Opteron Md1 6272 processors), 32 GB memory, a Corsair 120GB SSD, and a 2 TB hard disk. Also, each node has three PNY GeForce GTX580 GPU cards, each of which provides 512 cores and 1.5 GB memory. Therefore, in total, our cluster provide 128 CPU threads, 6,144 GPU cores, 128 GB host memory, and 18 GB GPU memory. Our nodes provide considerable computational parallelism; however, the available network bandwidth is relatively limited. Thus, efficient parallel data processing from the beginning to the end, i.e., from the input split to reduce phases, as well as efficient combining of duplicate <key, value> pairs and load balancing are important to avoid the potential bottleneck.

3.2 Performance Evaluation Results

In this section, we present the performance results for the different data sets shown in Table 2 as well as different numbers of GPUs, i.e., 4, 8, and 12 GPUs. For word count, we have generated 500 million, 800 millin, and 1 billion words using Wikipedia data sets [26]. For matrix multiplication, we have used randomly generated floating point numbers. Note that the same data sets are used to compare the performance of Moim and GPMR.

3.2.1 Word Count

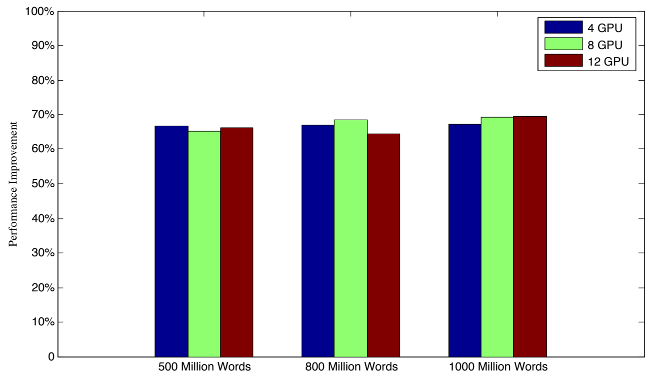

Figure 4. Performance Improvement over GPMR for Word Count

As shown in Figure 4, Moim achieves an average 67% speedup over GPMR [23] in terms of the total delay measured in wall-clock time, because GPMR does not support the four key features of Moim summarized in Section 1. Furthermore, in its implementation of word count [11], GPMR pre-hashes keys, i.e., words. It uploads the hash values to the GPUs that run the map() function. By doing this, it avoids the difficulty of managing variable size data. However, information loss occurs in GPMR, if two or more words happen to have the same hash value due to a hash collision. In such a case, the different words are reduced together into a single key in GPMR. In contrast, Moim does not cause any information loss, while substantially expediting parallel data processing.

Moim achieves the speedup due to 1) parallel split and load balancing among mappers, 2) 3-stage pipelining, 3) parallel merge of <key,value> pairs, and 4) load balancing among reducers described in Section 2. More specifically, Figure 5 shows the speedup of parallel split over serial split that uses only one CPU core in each node to tokenize 500 million words, 800 million words, and 1 billion words in terms of the wall-clock time. As there are 4 nodes in our cluster, serial split uses only 4 cores in total without leveraging all the CPU cores in each node. On average, parallel split achieves a 70%, 83%, 88.5%, and 91% speedup over serial split by using 8, 16, 24, and 32 CPU cores, respectively. We observe that the performance scales in a sublinear fashion possibly due to the contention for shared resources, such as the last level cache, system bus, and memory controller, in the multicore architecture. A thorough investigation of a more efficient approach is reserved for future work.

In addition, Figure 6 shows the degree of load balance achieved by Moim in the word count benchmark for 500 million words. The figure shows the numbers of <key,value> pairs assigned to the reducers running in 12 GPUs. From the results, we observe that load is relatively well balanced despite the skewed distribution of English words due to the effectiveness of our load balancing method. Also, the total number of the <key,value> pairs shuffled to the reducers is substantially smaller than (less than 17% of) 500 million due to our parallel merge phase. Moim has shown similar results for 800 million and 1 billion words. Our approach contrasts to the key-based hashing technique for load distribution that may produce considerable load imbalance among the reducers. Despite the drawback, the hashing scheme is widely employed in most MapReduce frameworks including [23, 7, 12, 22, 14, 15, 2, 9].

3.2.2 Matrix Multiplication

As shown in Figure 7, Moim achieves 21% – 47%, 21% – 27%, and 23% – 24% speedup over GPMR in terms of the total delay measured in wall-clock time using 4, 8, and 12 GPUs to multiply two matrices of 2048 × 2048, 4096 × 4096, and 10240 × 10240 data elements each. As the data size increases, the degree of speedup generally decreases, as the available parallelism becomes less sufficient to process bigger data.

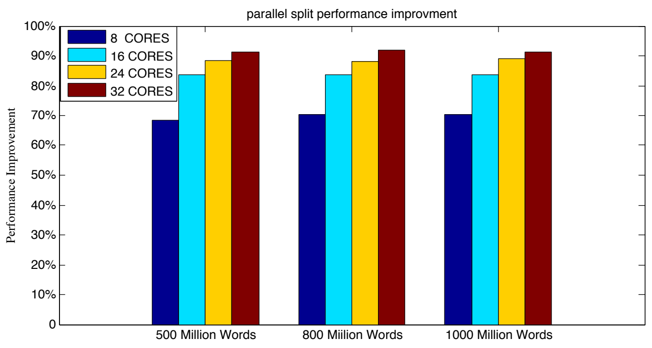

Figure 5. Speedup of Parallel Split over Serial Split

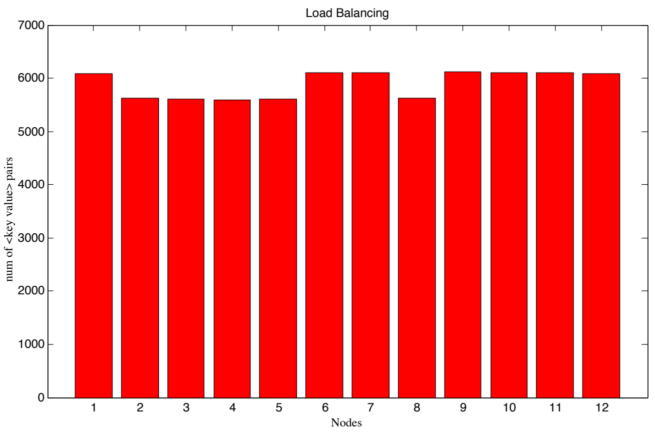

Figure 6. Load Balancing Results among Reducers in 12 GPU Cards

As matrix multiplication deals with fixed size data and requires no reduce phase, parallel split, parallel combining, and load balancing among reducers are irrelevant. Nevertheless, in this experiment, we show that it is possible to efficiently process not only variable but also fixed size data in Moim. To multiply two matrices A and B, GPMR divides each of them into smaller tiles, i.e., sub-matrices [11]. It first multiplies the tiles of the two input matrices, and runs another map function to compute the final result. Instead, Moim divides A into rows and B into columns, and evenly distributes A's rows and B's columns to the GPUs. Each GPU in turn multiples the assigned rows and columns in a single phase. One may argue this approach is not applicable, if one row or column is too big to fit into the GPU memory. In such a case, Moim can use the tiling technique, providing similar performance to GPMR.

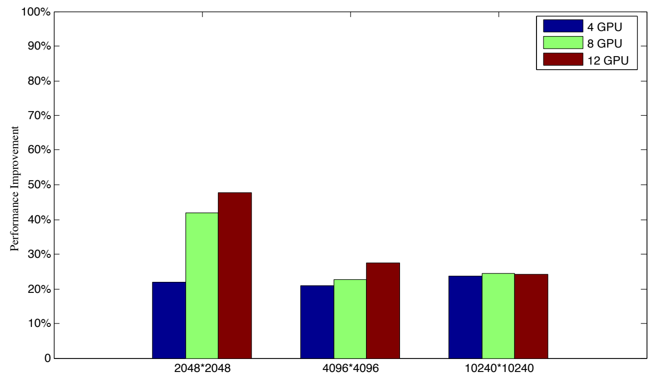

Figure 7. Performance Improvement over GPMR for Matrix Multiplication

4 Related Work

In this section, we give a brief overview of the MapReduce frameworks closely related to Moim.

Single GPU Frameworks: Mars [14] has implemented a MapReduce framework using a GPU to decrease the processing delay. Due to the lack of dynamic memory management in a GPU, Mars [14] computes the amount of the GPU memory needed to process variable size data in the first pass. In the second pass, it processes a real MapReduce job. As the first pass performs every MapReduce phase except for producing actual results, Mars executes a MapReduce job almost twice. Although Mars is extended to use two GPUs later [8], the overall system designed is maintained.

MapCG [15] avoids two pass executions by supporting dynamic memory allocation via atomic operations in a GPU. However, atomic operations require serial processing subject to performance penalties.

Grex [2] avoids the repeated executions and atomic operations of Mars and MapCG by doing parallel split of variable size data in a GPU. However, parallel split in Grex consumes a lot of GPU memory. For variable size data, any byte in the raw input can be the beginning of a token. Thus, GPU memory space for a pointer should be reserved for each byte in the input.

Multi-GPU Frameworks: Mithra [9] uses Hadoop to cluster GPU nodes to leverage the parallelism provided by GPUs and powerful features of Hadoop, such as fault tolerance. However, it also inherits the drawbacks of Hadoop, such as the lack of intra-node parallel data processing using multicore CPUs and potential load imbalance among the reducers. Neither does it intend to overlap CPU and GPU computations.

GPMR [23] is an efficient MapReduce framework designed for a GPU cluster. Unlike Moim, efficient processing of data with variable sizes or skewed distributions is not GPMR's design objective. It does not support parallel split or merge using multicore CPUs for variable size data. For integer keys, it distributes <key,value> pairs to reducers in a round robin manner; however, the authors admit the simple round robin scheme is insufficient for load balancing among reducers even for integer keys [23].

MGMR [5] is a multi-GPU MapReduce framework that takes advantage of multiple GPUs in a single machine. It applies the original sample sort algorithm [16] to shuffle the intermediate <key,value> pairs produced in the map phase to the multiple GPU cards in one node that run reducers. Although sample sort is an efficient parallel sorting algorithm, it does not support load balancing for data with a skewed distribution. Thus, the GPUs may suffer from load imbalance. MGMR focuses on the efficient utilization of GPUs without trying to exploit CPU cores as well. It does not intend to overlap CPU and GPU computations. In addition, they only use radix sort applicable just to primitive data types.

Multicore CPU and Cell Processor Frameworks: Phoenix++ [22] is a MapReduce implementation for a multicore shared-memory architecture; however, it does not use GPUs to accelerate MapReduce jobs. CellMR [21] is a MapReduce framework designed for the IBM Cell B.E. architecture [21]. Moim leverages the immense parallelsim provided by GPUs, while using multicore CPUs more widely available than Cell processors.

As GPU computational capacities of GPUs increase, GPUs are used to support diverse applications, such as sorting [25], text processing [6], Fast Fourier Transform [10], matrix operations [4], databases [1], encryption and decryption [24], intrusion detection [20], and biological applications [18]. These efforts call for a more general parallel programming framework, such as MapReduce, to exploit the parallel computational power provided by CPUs and GPUs with lower complexity compared with manual architectural optimizations.

5 Conclusions and Future Work

Although MapReduce greatly decreases the complexity of developing applications for parallel data processing, there is ample room for performance improvement. In this paper, we present a new MapReduce framework, called Moim, which strives to 1) effectively utilize both CPUs and GPUs, 2) overlap CPU and GPU computations, 3) improve load balancing in the map and reduce phases, and 4) efficiently handle not only fixed but also variable size data. We have implemented Moim and compared its performance with an advanced multi-GPU MapReduce framework [23] that represents the state of the art. In our experiments, Moim achieves $20\% - 90\%$ speedup for word count and matrix multiplication that process variable and fixed size data, respectively. In the future, we will investigate more efficient techniques to further enhance the performance, while exploring cost-effective methods for fault tolerance.

Acknowledgment

This work was supported, in part, by the NSF grant CNS-117352.

References

[1] P. Bakkum and K. Skadron. Accelerating SQL Database Operations on a GPU with CUDA. In *Workshop on General-Purpose Computation on Graphics Processing Units*, 2010.

[2] C. Basaran and K.-D. Kang. Grex: An Efficient MapReduce Framework for Graphics Processing Units. *Journal of Parallel and Distributed Computing*, 73(4):522 – 533, 2013.

[3] N. Bell and J. Hoberock. Thrust: A productivity-oriented library for CUDA. In W. mei W. Hwu, editor, *GPU Computing Gems: Jade Edition*. Morgan Kaufmann, 2011.

[4] J. Bolz, I. Farmer, E. Grinspun, and P. Schrder. Sparse matrix solvers on the gpu: Conjugate gradients and multigrid. *ACM Transactions on Graphics*, 22:917–924, 2003.

[5] Y. Chen, Z. Qiao, H. Jiang, K.-C. Li, and W. W. Ro. MGMR: Multi-GPU Based MapReduce. In *International Conference on Grid and Pervasive Computing*, 2013.

[6] P. K. Chong, E. K. Karuppiah, K. K. Yong, and M. Berhad. A Multi-GPU Framework for In-Memory Text Data Analytics. In *International Conference on Advanced Information Networking and Applications Workshops*, 2013.

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

[8] W. Fang, B. He, Q. Luo, and N. K. Govindaraju. Mars: Accelerating MapReduce with Graphics Processors. *IEEE Transactions on Parallel Distributed Systems*, 22(4):608–620, 2011.

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

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

[11] GPMR: MapReduce for GPU Clusters. `http://code.google.com/p/gpmr/`.

[12] Hadoop project. `http://hadoop/apache/org`.

[13] Class InputFormat<K,V>. `http://hadoop.apache.org/docs/current/api/org/apache/hadoop/mapreduce/InputFormat.html`.

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

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

[16] X. Li, P. Lu, J. Schaeffer, J. Shillington, P. S. Wong, and H. Shi. On the versatility of parallel sorting by regular sampling. *Journal of Parallel Computing*, 19(10):1079–1103, Oct. 1993.

[17] J. Lin and C. Dyer. *Data-Intensive Text Processing with MapReduce.* Morgan and Claypool Publisher, 2010.

[18] S. Manavski and G. Valle. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC Bioinformatics*, 9(S-2), 2008.

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

[20] Q. Qian, H. Che, R. Zhang, and M. Xin. The Comparison of the Relative Entropy for Intrusion Detection on CPU and GPU. In *Australasian Conference on Information Systems*, 2010.

[21] M. M. Rafique, B. Rose, A. R. Butt, and D. S. Nikolopoulos. CellMR: A Framework for Supporting Mapreduce on Asymmetric Cell-Based Clusters. In *IEEE International Symposium on Parallel & Distributed Processing*, 2009.

[22] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Phoenix++: Modular MapReduce for Shared-Memory Systems. In *International Workshop on MapReduce and its Applications*, 2011.

[23] J. A. Stuart and J. D. Owens. Multi-GPU MapReduce on GPU Clusters. In *IEEE International Parallel and Distributed Processing Symposium*, 2011.

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

[25] I. Tanasicx, L. Vilanovax, M. Jordáx, J. Cabezasx, I. G. Navarroxz, and W. mei Hwu. Comparison Based Sorting for Systems with Multiple GPUs. In *Workshop on General Purpose Processor Using Graphics Processing Units*, 2013.

[26] `http://dumps.wikimedia.org/enwiki/`.