2013 IEEE International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing

GCMR: A GPU Cluster-based MapReduce Framework for Large-scale Data Processing

Yiru Guo*, Weiguo Liu*, Bin Gong*, Gerrit Voss† and Wolfgang Müller-Wittig†
*School of Computer Science and Technology, Shandong University, China
Email: guoyiru@mail.sdu.edu.cn, {weiguo.liu, gb}@sdu.edu.cn
†Fraunhofer IDM@NTU, Nanyang Technological University, Singapore
Email: {vossg, wolfgang.mueller-wittig}@fraunhofer.sg

***Abstract*—MapReduce is a very popular programming model to support parallel and distributed large-scale data processing. There have been a lot of efforts to implement this model on commodity GPU-based systems. However, most of these implementations can only work on a single GPU. And they can not be used to process large-scale datasets. In this paper, we present a new approach to design the MapReduce framework on GPU clusters for handling large-scale data processing. We have used Compute Unified Device Architectures (CUDA) and MPI parallel programming models to implement this framework. To derive an efficient mapping onto GPU clusters, we introduce a two-level parallelization approach: the inter node level and intra node level parallelization. Furthermore in order to improve the overall MapReduce efficiency, a multi-threading scheme is used to overlap the communication and computation on a multi-GPU node. Compared to previous GPU-based MapReduce implementations, our implementation, called GCMR, achieves speedups up to 2.6 on a single node and up to 9.1 on 4 nodes of a Tesla S1060 quad-GPU cluster system for processing small datasets. It also shows very good scalability for processing large-scale datasets on the cluster system.**

***Keywords*-MapReduce, CUDA, MPI, GPU Cluster.**

I. Introduction

MapReduce is a very popular parallel programming framework, which was firstly proposed by Google [4], for processing large-scale datasets. Map and Reduce are the two major functions in the MapReduce framework. In the map procedure, input datasets are divided into smaller chunks and distributed to all compute nodes for processing. The Reduce function will collect the records with the same key and process them as one input record. The MapReduce framework has been widely used in practice. Examples include scientific computing [3], database operations [15], [1], image processing [8], and bioinformatics [9].

Due to the continuing rapid growth of data size in a wide variety of application domains, there are still high demands for faster MapReduce solutions. Recently, many efforts have been put to accelerate MapReduce using various parallel architectures. Because of the high performance-to-price ratio and widespread availability, design and development of GPU-based MapReduce framework have been put to the forefront of high-performance computing. Examples of such frameworks include Mars [6], StreamMR [5], MapCG [7], and GPMR [14].

In this paper we present GCMR, a new implementation to accelerate MapReduce for large-scale data processing on GPU clusters. We have implemented GCMR using the CUDA and MPI programming models. To gain efficiency we have used a two-level parallelization scheme – the *inter node level* and *intra node level* to implement our framework. The performance of GCMR is tested using both small-scale and large-scale datasets for three commonly used applications. Compared to previous GPU-based MapReduce implementations, our implementation shows much better performance and scalability on a GPU cluster.

The rest of this paper is organized as follows. In Section II, we introduce features of CUDA, the MapReduce model and give a brief summary of previous work on accelerating MapReduce on different architectures. Section III presents our method to implement the GPU cluster-based MapReduce framework. Performance is evaluated in Section IV. Finally, Section V concludes the paper.

* Weiguo Liu is the corresponding author.

II. Related Work

A. CUDA Programming Model and Tesla Architecture

Compute Unified Device Architecture (CUDA) is a hardware and programming model for issuing and managing computations on the GPU as a data-parallel computing device without the need of mapping them to a graphics API [10]. For now, it is available for NVIDIA Geforce, Quadro, and Tesla GPU products.

CUDA programs contain a sequential part, called a kernel. The kernel is written in a C-like programming language. It represents the operations to be performed by a single thread and is invoked as a set of concurrently executing threads. These threads are organized in a hierarchy consisting of so-called thread blocks and grids. A thread block is a set of concurrent threads and a grid is a set of independent thread blocks. The hierarchical organization into blocks and grids has implications for thread communication and synchronization. Threads within a thread block can communicate through a *per-block shared memory* (PBSM) and may synchronize using barriers. However, threads located in different blocks cannot communicate or synchronize directly. Besides the PBSM, there are four other types of memory: per-thread private local memory, global memory for data shared by all threads, texture memory and constant memory. Texture memory and constant memory can be regarded as fast read-only caches.

The Tesla architecture supports CUDA applications using a scalable processor array. The array consists of a number of *streaming multiprocessors* (SM). Each SM contains eight scalar processors, which share a PBSM. All threads of a thread block are executed concurrently on a single SM. The SM executes threads in small groups of 32, called *warps*, in a *single-instruction multiple-thread* (SIMT) fashion. Thus, parallel performance is generally penalized by data-dependent conditional branches and improved if all threads in a warp follow the same execution path.

B. Previous Work on Accelerating MapReduce

MapReduce is a parallel programming framework based on functional programming models. It was firstly proposed by Google [4] for large-scale data processing. By using the MapReduce libraries, users can express the computation tasks as the *map* and *reduce* functions which are defined as follows:

$$\begin{array}{lll} \text{map} & (k1, v1) & \rightarrow \text{list}(k2, v2) \\ \text{reduce} & (k2,\ \text{list}(v2)) & \rightarrow \text{list}(v2) \end{array}$$

The *map* function takes a set of input $(key, value)$ pairs $(k1, v1)$ and generates a set of corresponding intermediate $(key, value)$ pairs $(k2, v2)$. The MapReduce libraries group together all of the intermediate values associated with the same key and pass them to the *reduce* function. The *reduce* function accepts an intermediate key $k2$ and a list of values list$(v2)$ associated with that key. It merges these values to form a possibly smaller set list$(v2)$. Typically, just zero or one output value is produced per *reduce* invocation.

Due to its popularity in large-scale data processing, there have been a lot of efforts to accelerate MapReduce on various parallel architectures. Previous approaches to accelerate MapReduce can be classified into two categories: (1) multi-threading, and (2) reconfigurable architectures. Multi-threaded approaches run on multi-core CPUs, on SIMD multiprocessor GPUs or on the CELL/BE. Phoenix [12] is a MapReduce implementation on multi-core CPUs. TMR [2] further optimizes the resource usage on multi-core CPUs using tiling. A solution on the CELL/BE was introduced in [11]. Reconfigurable solutions are based on FPGAs. FPMR [13] is an example of FPGA-based MapReduce solution.

The main advantage of GPUs compared to FPGA is that they are commodity components. In particular, most users already have access to PCs with modern GPUs. For these users this direction provides a zero-cost solution. The paper presented by He et al. [6] illustrates the Mars system which is the first GPU-based MapReduce implementation. MapCG [7] is another MapReduce implementation supporting both multi-core CPUs and Nvidia GPUs. StreamMR [5] is an OpenCL-based MapReduce framework which supports both Nvidia and AMD GPUs. Mars, MapCG and StreamMR have limited scalability since they only support a single GPU node and small-scale datasets. Our solution overcomes these bottlenecks by using a two-level parallelization approach as well as a multi-threading overlap scheme. Experiments show that our implementation can be used to process large-scale datasets efficiently on multi-GPU clusters.

III. Design and Implementation

A. Overall Design

Fig. 1 shows the framework of our implementation. It is mainly composed of four stages: (1) splitting input data to chunks, (2) map and localReduce on multiple GPUs, (3) shuffling intermediate records between different nodes, and (4) reducing new intermediate records to emit the final outputs.

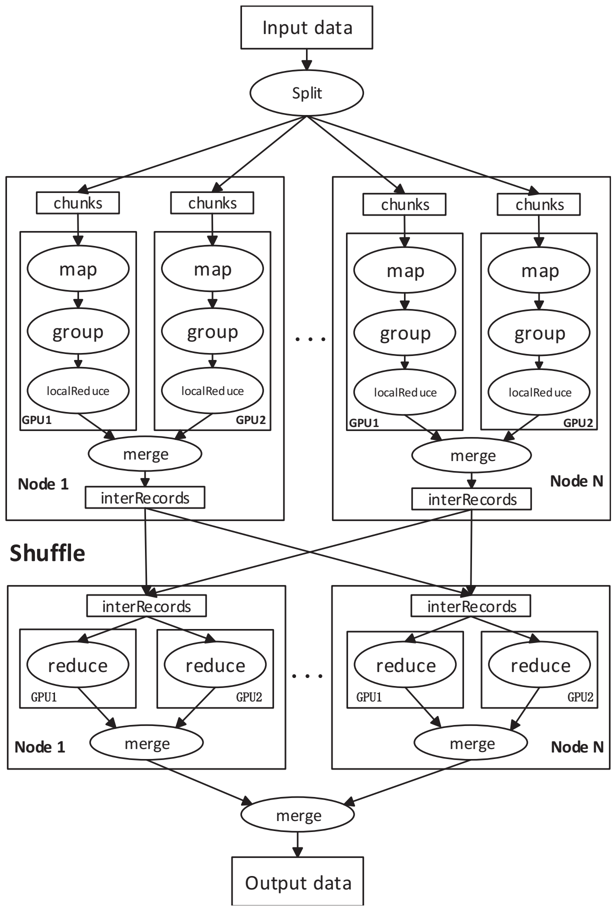


Fig. 1. Our GPU cluster-based MapReduce framework.

On the *inter node level*, we first split input data into chunks and then distribute these chunks evenly to all compute nodes. In the map and localReduce stage, local data chunks are transformed into $(key, value)$ pairs and sent to GPU devices. Then each GPU device does the map procedure in a fine-grained parallel fashion. Next, the produced intermediate records with the same key will be grouped together. In order to improve the computing efficiency, the localReduce step is used to post-process the previously grouped intermediate records into a smaller dataset. In the shuffle stage, each node first divides local intermediate records into several batches based on their keys. The batches with the same range of keys are then sent to corresponding nodes. Thus records with the same key are sent to the same node and merged as new intermediate records for further processing. Finally in the reduce stage, the intermediate records with the same key are sent to GPU device for emitting the final results. Using our framework, users can express the computation tasks as the *map*, *local_reduce*, and *reduce* functions which are defined as follows:

$$\begin{array}{lll} \text{map} & (k1, v1) & \rightarrow \text{list}(k2, v2) \\ \text{local\_reduce} & (k2, \text{list}(v2)) & \rightarrow \text{list}(k2, v2) \\ \text{reduce} & (k2, \text{list}(v2)) & \rightarrow \text{list}(k3, v3) \end{array}$$

The *map* function takes a set of input $(key, value)$ pairs in the form of $< k1, v1 >$ and generates a set of intermediate $(key, value)$ pairs $< k2, v2 >$. The *local_reduce* function reduces these intermediate pairs to generate the partial results on each GPU. It receives an intermediate key with the type $k2$ and a list of intermediate values in the form of $v2$. Then it outputs a set of partial $(key, value)$ pairs $< k2, v2 >$. Finally the *reduce* function handles partial pairs with the same key and outputs the final result pairs $< k3, v3 >$.

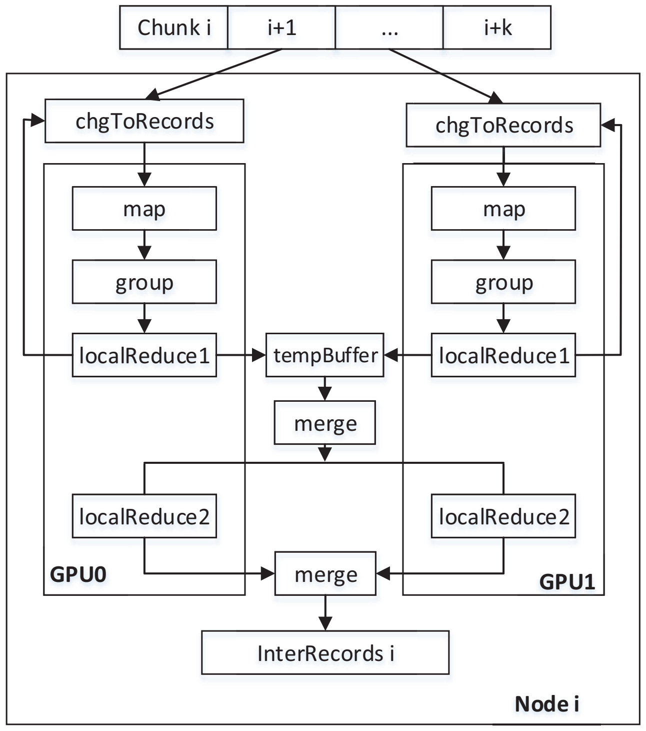


Fig. 2. Illustration of our method to process local data chunks using multiple GPUs on each node. Here we assume that there are two GPUs on each node.

Fig. 2 further illustrates how we implement GCMR on the *intra node level*. From Fig. 2 we can see that there are two localReduce functions in the work flow. LocalReduce1 starts working after the group function. It is responsible for handling the map results which have

TABLE I
THE API FUNCTIONS IN GCMR.

Functions defined by users
void map(void key, void* val, unsigned int key_size, unsigned int val_size)* The map function. Each map task processes one input $< key, value >$ pair.
void local_reduce(void key, void* val, unsigned int key_size, unsigned int val_count)* The local reduce function processes the local intermediate records with the same key.
void reduce(void key, void* val, unsigned int key_size, unsigned int val_count)* The reduce function deals with the records with the same key from all GPU nodes.
int compareKey(void key1, unsigned int size1, void* key2, unsigned int size2)* This function is used to compare two records based on their keys.
Functions provided by runtime
*void setConf(Spec_t * spec)* This function initializes the basic configuration and controls the general workflow.
void chgToRecords(Spec_t spec, void* key, void* val, unsigned int key_size, unsigned int val_size)* This function transforms input file to $< key, value >$ pairs.
void startMapReduce(Spec_t spec)* This function starts the main procedure of MapReduce.
void finalize (Spec_t spec))* This function de-initializes the program.
void EMIT_FUNC(void key, void* val, unsigned int key_size, unsigned int val_size)* This function emits an output record in map or reduce functions.

been sorted. And it copies outputs to a temp buffer in the host memory. Once all local data chunks have been processed by the map and localReduce1 functions, the temp buffer will merge temp outputs.

In order to decrease the size of intermediate results, localReduce2 will be further used to merge the output records generated from different local data chunks.

B. APIs

As shown in TABLE I, our framework mainly provides two kinds of APIs: the user defined APIs and runtime APIs. The user defined APIs, which are coded using C/C++, are implemented by users. They are used to implement the main work of map/reduce procedure for each record. Runtime APIs are provided by our framework. They are used to help users complete the main workflow of particular applications.

The following pseudo code shows how we implement the Word Count (WC) application using GCMR. In our WC implementation, the map function scans a line of the text document each time. Then it emits each word as the key and a temp count 1 as the value. On each node, the local_reduce function counts the intermediate records which come from each data chunk and emits the unique word as the key and the partial count k as the value. The reduce function collects all the temp results and add calculates the final results.

```
//input: a document
//intermediate results: $< word, 1 >$
map(key, val, keySize, valSize):
    for each word w in the input:
        EMIT_FUNC(w, 1);

//intermediate results: $< word, 1 >$
//local output results: $< word, k >$
local_reduce(key, vals, keySize, valCount):
    EMIT_FUNC (key, valCount);

//local output results: $< word, k >$
//output: $< word, n >$
reduce(key, vals, keySize, valCount):
    int totalCount = 0;
    for each v in vals:
        totalCount += v;
    EMIT_FUNC(word, totalCount);
```

C. Multi-threading Overlap Scheme

Data transfer between the CPU and GPU is a known bottleneck for many GPU-based applications and, therefore, should be minimized. This bottleneck is caused by

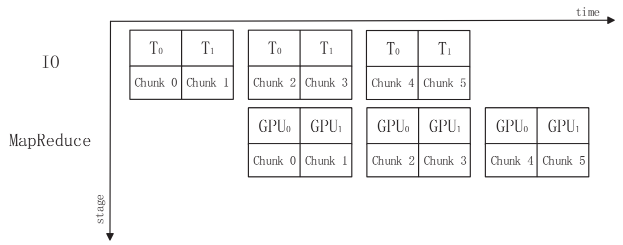


Fig. 3. Illustration of our multi-threading overlap scheme.

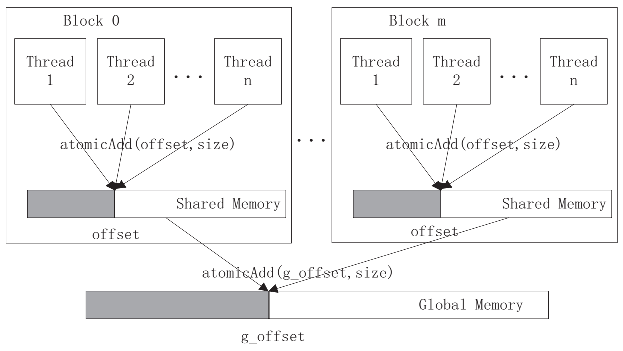


Fig. 4. Illustration of our two level atomic operation scheme.

the slow file I/O, relatively low PCI Express bus bandwidth, as well as the overhead associated with initializing each transfer. In order to increase the efficiency of our framework, we have used a multi-threading pipeline to overlap the data transfer and computation between the host CPU and GPU device. Fig. 3 gives an illustration of our method. In Fig. 3, we assume that there are two GPUs on a compute node. In this case, we first launch two threads T_0 and T_1 to load two local data chunks (Chunk 0 and Chunk 1) into host memory and transform them to input $(key, value)$ pairs in parallel. Then we send them to two GPU devices (GPU_0, GPU_1) for MapReduce computation. At the same time, T_0 and T_1 will load next two local data chunks into host memory. This pipeline will continue until all local data chunks are processed. Experiments show that our multi-threading pipeline can improve the performance of our framework greatly.

D. Two-level Atomic Operation Scheme

Because of lacking of dynamic memory allocator on GPU, Mars takes a two-step scheme to write results to global memory. First, it counts the size of output records from each GPU thread and computes their prefix sum as write positions. Then each GPU thread puts the output records to the corresponding write location. Since this method needs additional counting phases, its performance is not good.

In our framework, we implement the dynamic memory allocation by using a two-level atomic operation scheme on GPUs. Fig. 4 illustrates our method. In our method, we first store the intermediate results of each thread block into a high-performance shared memory array. Before a thread writes results into shared memory, it first gets the writing position using the atomic operation. After all threads in a thread block finish the writing operation, results in the shared memory array will be written into a global memory array. Similar with the previous step, each thread block needs to get the writing position using the atomic operation before writing results into global memory. Our method is similar, but not identical to the method used in MapCG [7]. In MapCG, it copies results in the shared memory into global memory when a thread warp completes the writing to shared memory. Our method decreases the copy frequency by waiting for the whole thread block finishes the local writing operation and thus improves the performance.

IV. Performance Evaluation

Our framework is implemented using CUDA, multi-threading, and MPI programming models. We have evaluated our implementation on a GPU cluster which is composed of 4 nodes and connected by the high-speed InfiniBand switch. Each node contains a Quad-Core AMD Opteron(tm) Processor with 2.4 GHz clock rate, two NVIDIA Tesla c1060 GPUs, and 8 GB host memory. And each Tesla c1060 GPU has 240 CUDA cores (30 MP$\times$8 cores/MP) with 1.3 GHz GPU clock rate, 4 GB global memory, 16KB shared memory on each SM. The operating system is the 64-bit CentOS 6.3 with the kernel version 2.6.32.

We have conducted our tests using three commonly used applications: Word Count (WC), String Match (SM), and Matrix Multiplication (MM). The WC application is used to count the appearance frequency of each word in a text file. In our test, we first divide the text file into smaller sized chunks and distribute them evenly to all compute nodes. Then each Map thread emits a (word, 1) record. Next, the Reduce threads collect all the records with the same key within the cluster and output the frequency for each word. The SM application is used in web documents for searching a given string. On GPU clusters, each node is responsible for scanning part of the documents. The MM application is used to calculate the result of two matrix multiplication. In our

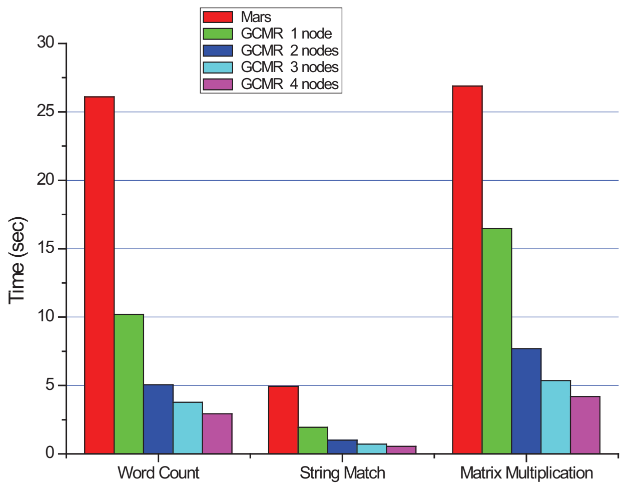
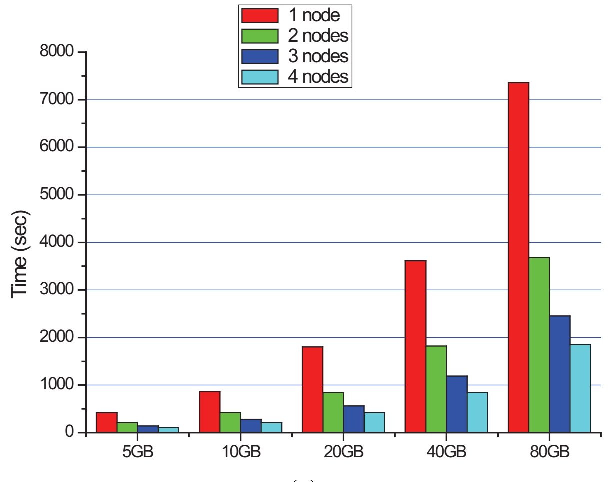


Fig. 5. The run time of three applications on Mars and GCMR.

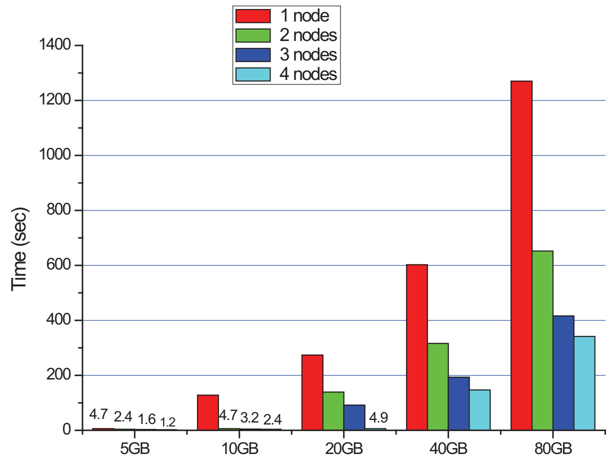
test, we divide the MM task into a set of sub matrix multiplications. And each GPU node works on these sub matrix multiplications.

We have compared the performance of GCMR to Mars [6]. Fig. 5 shows the runtime of Mars and GCMR for processing WC, SM, and MM. Because Mars only supports a single GPU and small-scale datasets, in our tests we have used three small-scale datasets for each applications: WC: 150MB, SM: 2GB, MM: 4000×4000 and 4000×4000. And we have run Mars on an Nvidia Tesla C1060 GPU. As to GCMR, we have run it on a compute node equipped with two Nvidia Tesla C1060 GPUs. Fig. 5 shows the runtime comparison between Mars and GCMR. From it we can see that GCMR achieves speedups up to 2.6 on a single compute node and up to 9.1 on 4 compute nodes of a GPU cluster system. Also, GCMR shows very good scalability for processing these small-scale datasets as the number of compute nodes increases.

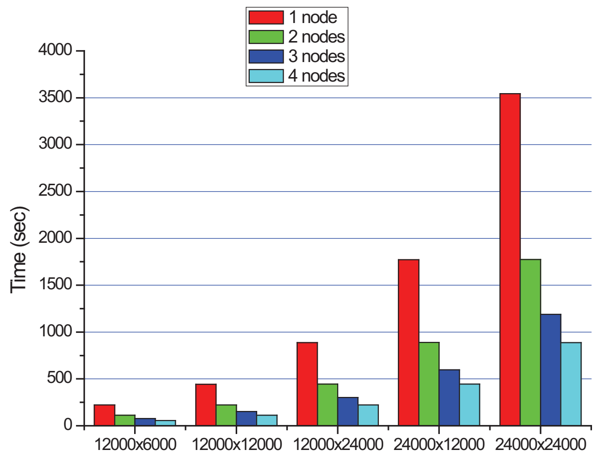
We have also evaluated the scalability of GCMR for handling large-scale datasets. In our tests, we have used various large-scale datasets for WC (5GB to 80 GB text files), SM (5GB to 80 GB text files), and MM (the matrix size varies from 12000×6000 to 24000×24000). Fig. 6 shows the runtime of GCMR for processing these datasets. From it we can see that GCMR achieves linear scalabilities for processing these large-scale datasets as the number of compute nodes increases. As to SM, its runtime is very short when data sizes are 5GB, 10GB and 20GB. This is because SM is an I/O dominant application. When the data size is less than the local



(a)



(b)



(c)

Fig. 6. .The run time of WC, SM and MM on different number of GPU cluster nodes when handling various sizes of datasets using GCMR. For Matrix Multiplication, if 12000×6000 is the first matrix size, the corresponding second matrix size will be 6000×12000.

cache size, the I/O performance will be very high and thus the total runtime will be very short.

V. Conclusion

In this paper we present GCMR, a new GPU cluster-based MapReduce framework. We have used CUDA and MPI parallel programming models to implement this framework. To derive an efficient mapping onto GPU clusters, we have used a two-level parallelization approach: the inter node level and intra node level parallelization. Furthermore in order to improve the overall MapReduce efficiency, a multi-threading scheme is used to overlap the communication and computation on a multi-GPU node. Experiments show that compared to previously implemented GPU-based MapReduce framework, our framework achieves better performance and much better scalability for processing large-scale datasets. Our results are especially encouraging since performance of many-core architectures grows faster than the performance of standard multicore CPUs. For example, we expect that the already announced 20 series of NVIDIA Kepler GPUs will further improve the performance of MapReduce by at least a factor of 2.

Acknowledgment

This work was supported in part by Key Projects in the National Science & Technology Pillar Program(No.2013BAH39F00), Fraunhofer IDM@NTU and Media Development Authority (MDA) Singapore.

References

[1] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. In *Proceedings of the VLDB Endowment*, pages 922–933, 2009.

[2] R. Chen, H. Chen, and B. Zang. Tiled-MapReduce: Optimizing Resource Usage of Data-parallel Applications on Multicore with Tiling. In *Parallel Architectures and Compilation Techniques*, pages 523–534, 2010.

[3] C. Chu, S. K. Kim, Y. A. Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun. Map-reduce for Machine Learning on Multicore. In *Advances in Neural Information Processing Systems*, pages 281–288, 2007.

[4] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI 2004*, pages 137–150, 2004.

[5] M. Elteir, H. Lin, and W. C. Feng. StreamMR: An Optimized MapReduce Framework for AMD GPUs. In *IEEE 17th International Conference on Parallel and Distributed Systems*, pages 364–371, 2011.

[6] B. He, W. Fang, Q. Luo, N. Govindaraju, and T. Wang. Mars: a MapReduce Framework on Graphics Processors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 260–269, 2008.

[7] C. Hong, D. Chen, W. Chen, W. Zheng, and H. Lin. MapCG: Writing Parallel Program Portable Between CPU and GPU. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 217–226, 2010.

[8] J. Lin and M. Schatz. Design Patterns for Efficient Graph Algorithms in MapReduce. In *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*, pages 78–85, 2010.

[9] A. McKenna, M. Hanna, E. Banks, A. Sivachenko, K. Cibulskis, A. Kernytsky, and M. A. DePristo. The Genome Analysis Toolkit: A MapReduce Framework for Analyzing Next-Generation DNA Sequencing Data. *Genome research*, pages 1297–1303, 2010.

[10] Nvidia. *NVIDIA CUDA Compute Unified Device Architecture-Programming Guide*. http://developer.download.nvidia.com/compute/cuda, 2007.

[11] M. Rafique, B. Rose, A. Butt, and D. Nikolopoulos. CellMR: A Framework for Supporting MapReduce on Asymmetric Cell-Based Clusters. In *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*, pages 1–12, 2009.

[12] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating Mapreduce for Multicore and Multiprocessor Systems. In *International Symposium on High-Performance Computer Architecture*, pages 13–24, 2007.

[13] Y. Shan, B. Wang, J. Yan, Y. Wang, N. Xu, and H. Yang. FPMR: MapReduce framework on FPGA, A Case Study of RankBoost Acceleration. In *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, pages 93–102, 2010.

[14] J. Stuart and J. Owens. Multi-GPU MapReduce on GPU Clusters. In *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium*, pages 1068–1079, 2011.

[15] H. C. Yang, A. Dasdan, R. L. Hsiao, and D. S. Parker. Map-reduce-merge: Simplified Relational Data Processing on Large Clusters. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1029–1040, 2007.