Cluster Comput (2015) 18:369–383
DOI 10.1007/s10586-014-0400-1

Scaling up MapReduce-based Big Data Processing on Multi-GPU systems

Hai Jiang · Yi Chen · Zhi Qiao · Tien-Hsiung Weng · Kuan-Ching Li

Received: 23 August 2013 / Revised: 21 May 2014 / Accepted: 8 August 2014 / Published online: 22 August 2014

Abstract MapReduce is a popular data-parallel processing model encompassed with recent advances in computing technology and has been widely exploited for large-scale data analysis. The high demand on MapReduce has stimulated the investigation of MapReduce implementations with different architectural models and computing paradigms, such as multi-core clusters, Clouds, Cubieboards and GPUs. Particularly, current GPU-based MapReduce approaches mainly focus on single-GPU algorithms and cannot handle large data sets, due to the limited GPU memory capacity. Based on the previous multi-GPU MapReduce version MGMR, this paper proposes an upgrade version MGMR++ to eliminate GPU memory limitation and a pipelined version, PMGMR, to handle the Big Data challenge through both CPU memory and hard disks. MGMR++ is extended from MGMR with flexible C++ templates and CPU memory utilization, while PMGMR fine-tuned the performance through the latest GPU features such as streams and Hyper-Q as well as hard disk utilization. Compared to MGMR (Jiang et al., Cluster Computing 2013), the proposed schemes achieve about 2.5-fold performance improvement, increase system scalability, and allow programmers to write straightforward MapReduce code for Big Data.

H. Jiang · Y. Chen · Z. Qiao
Department of Computer Science, Arkansas State University, Jonesboro, AR, USA
e-mail: hjiang@astate.edu

T.-H. Weng · K.-C. Li (✉)
Department of Computer Science and Information Engineering, Providence University, Taichung 43301, Taiwan
e-mail: kuancli@pu.edu.tw

T.-H. Weng
e-mail: thweng@pu.edu.tw

Keywords GPU · Multi-GPU · MapReduce · Pipeline · Big Data · Parallel processing

1 Introduction

The need to process complex and large amount of data has increased dramatically in recent years. Representative applications include data-mining, geoanalysis, predictive analytics, statistical and experimental data analysis and visualization, commerce, and engineering among several others. Traditional methods and techniques have become incapable of processing large datasets within tolerable time periods.

Large-scale data processing is tougher than ever before since data size is increasing too fast for hardware resources to catch up with. The ability to process vast amount of data remains a critical challenge and hard mission for traditional data processing applications and relational database management systems. A number of different architectural designs and programming models have been proposed and designed to speed up the execution of applications with massive data sets. Such platforms range from multi-core clusters, hybrid clusters, clouds, mobile systems, Graphics Processing Units (GPUs) and Cubieboards [2].

GPUs have been considered an excellent alternative to CPUs for High Performance and High Throughput Computing applications. They can exploit data parallelism, increase computational efficiency, and save energy by orders of magnitude [3].

MapReduce has been widely adopted to process Big Data problems [4]. It was originally proposed by Google to provide a simple and flexible parallel programming paradigm. With MapReduce, users only need to write *Map* and *Reduce* functions for parallel computing. The underlying programming details, such as how to handle communication among data nodes, are transparent to users. Data affinity across the network and fault tolerance among multiple nodes can be achieved automatically.

Our previous work, MGMR, has already migrated the MapReduce framework into GPU environment and successfully utilized Multiple GPUs concurrently [1,5]. As GPU technology advances, powerful GPUs such as Nvidia Tesla Kepler have appeared. As to leverage the most advanced features of Kepler GPU such as Asynchronous Dual-channel Data Transfer and Hyper Q, pipelined workflow scheduling module is implemented and added to MGMR for further performance gains.

For large-scale data processing, increasing the system computability alone is not enough. The storage hierarchy including both CPU memory and hard disks is employed to enhance the storage and huge amount of data processing capability as well as efficient data transfer between disks and processing units. For such, it is proposed in this paper MGMR++ and PMGMR to redesign C-based MGMR for scalability. Both new systems are written in C++ with flexible C++ templates and Nvidia Thrust library for some built-in functions. MGMR++ utilizes CPU memory to eliminate the limitation of GPU memory size, whereas the pipelined version PMGMR extends MGMR++ further by adopting a flexible scheduling strategy and hard disks for even larger data capacity. This paper presents the following contributions:

- Multiple GPUs are utilized to accelerate MapReduce operations,
- CPU memory and hard disks are used to continue MapReduce operations, as data size exceeds the aggregate GPU memory,
- Pipelined workflow scheduler maximizes the usage of GPU by overlapping communication and computation to hide the traffic behind computation and minimize the overall communication overhead for larger throughput,
- A job scheduler is employed to manage memory and dataflow so that input size is no longer bounded by both GPU memory and CPU memory capacity,
- A configuration optimizer is adopted to collect the intermediate results and to dynamically adjust the environment settings for balanced loads,
- All above application-level schedulers and optimizer are implemented based on advanced Nvidia GPU features such as streams and Hyper-Q as well as Thrust library.

The remainder of this paper is organized as follows. Section 2 introduces GPU architecture and MapReduce framework, as also some related work, while Sect. 3 will detail the design of MGMR++ and PMGMR system and their architectural designs. We present experimental results in Sect. 4 and compare different versions. Finally, the conclusion and future work are given in Sect. 5.

2 Background and related work

2.1 Big Data

Big Data can be defined as a collection of large and complex data sets, which are difficult to be processed by relational database management systems or traditional data processing applications. Typical size of a problem in this class is found in Terabytes scale, and constantly growing.

The explosion of new computing and architectural models boosted the need for Big Data processing. Companies are facing difficulties in managing and processing vast amount of data as data size grows exponentially. They are unable to manage, manipulate, process, share and retrieve such large data through traditional software tools without turning data processing into a costly and time-consuming process. Companies such as Amazon, Google, Twitter, and Facebook have different goals to keep track users, analyze their information and browsing habits to improve the customer experience, and search keywords to identify emerging trends [6]. Therefore, their top priority job is not just to maintain a huge storage space but also to process the data as quickly as possible since real time response becomes vital.

2.2 MapReduce programming model

MapReduce is a programming model for processing large data sets, widely used in domains such as data mining [7], machine learning [9], and medical imaging [8]. The MapReduce model has been adopted by many computing platforms, such as multi-core systems [10,11], desktop grids [12,13], mobile platforms, clustered systems, clouds, and most recently, Cubieboards [2]. The design goals of MapReduce include programmability, robustness, portability and scalability.

Parallel Programming has been proven to be very challenging over years. MapReduce provides users and developers an easier way to generate parallel and distributed programs without worrying about the architecture of computing platforms. They only write simpler *Map* and *Reduce* functions, mainly focusing on the logic of the specific problem. The MapReduce system automatically takes care of the underlying execution details such as parallelism, communication, fault tolerance, and load balancing.

2.3 Multi-GPU architecture

Each GPU is consisted of multiple streaming-multiprocessors (SMs) that execute thousands of lightweight hardware threads concurrently. CUDA is the mechanism to exploit data parallelism on GPU cores. Up to 512 threads are grouped into thread blocks that are assigned to SMs where every 32 parallel threads are grouped into a warp for scheduling. Extremely fast context switch between warps can help tolerate memory access latency. All threads within the same block can access the common shared memory. This helps synchronize threads within the same block, and facilitate extensive reuse of on-chip data in order to greatly reduce off-chip traffic.

For servers with multiple Nvidia Fermi GPUs, GPUDirect is a technique used to handle inter-GPU communication via the PCIe bus directly, without CPU side data buffering [19]. High-speed DMA (Direct Memory Access) engines enable such inter-GPU communication, directly read/write data from/to another GPU's memory within same server. Therefore, it is possible to eliminate unnecessary copies through system memory on CPU side and achieve significant performance improvement in data transfer. Asynchronous bidirectional memory copy is another advanced feature that not only doubles the data transfer bandwidth, but also helps achieve the overlapping of computation and communication.

For recently released Tesla Kepler GPU, it is possible that GPUDirect transfers between third party devices such as network cards. With this feature, direct communication between multiple Kepler GPUs among different machines turns possible.

2.4 Related work

MapReduce has been implemented on many different platforms such as computer clusters, multi-core, shared memory, Cubieboards and CPU-GPU coupled architecture systems. Hadoop MapReduce [14] developed by Apache Software is designed for better programmability in processing vast amount of data in clusters. Hadoop was developed in Java, and Hadoop Streaming permits users to customize their own *Map* and *Reduce* functions in other programming languages such as C and Python.

Mars [20] is the first GPU-based MapReduce system, that makes use of an atomic-free output-handling scheme on GPUs. Unfortunately, it contains several drawbacks. It has a two-step output process to calculate the data allocation and avoid race conditions among threads. Also, it utilizes bitonic sorts on the output data generated by the Map stage to group the intermediate data, which has been proven to be inefficient.

StreamMR [21] is a GPU MapReduce framework for AMD GPUs, in which data chunks are partitioned into wavefronts, similar to the warp concept in NVidia's CUDA. A wavefront-wide atomic-free algorithm was developed to substitute global atomic operations for better performance. However, such an algorithm is driven by a pointer-based linked list that is not supported well in GPUs. Also, the per-wavefront buffer design causes inefficient memory utilization during data transfer, which is limited by the memory controller inside GPUs.

GPMR [11] extends GPU MapReduce to GPU cluster level. It splits the large input of Map and Reduce functions into chunks and uses partial reductions and accumulation to reduce network operations. GPMR utilized GPU clusters to achieve the speedup over other MapReduce libraries. However, it only focused on single GPU per node and the communication between multiple GPUs on the same node was not considered.

The Coupled CPU-GPU MapReduce [15] utilized two schemes as well as both CPU and GPU for data processing. In Map-Dividing scheme, the data of Map and Reduce stage is processed in both CPU and GPU, whereas in Pipelining Scheme, each device is only responsible for performing one stage of the MapReduce. To balance the load between GPU and CPU well, its runtime system reduces the block size of each worker to synchronize all workers. Unfortunately, this technique could also decrease GPU utilization when the input size is too small.

The motivation and contributions of MGMR++ and PMGMR systems are relevant and originated from the weaknesses found in some implementations of the aforementioned MapReduce systems.

3 Multi-GPU MapReduce design: MGMR++ and PMGMR

3.1 Challenges of Big Data

Big data has already become a challenge for MapReduce, just as in other data processing systems [16]. A common GPU-based MapReduce application tends to read input from hard disks to CPU memory and then GPU memory. Under the current storage hierarchy, the capacity of hard drive is much bigger than CPU memory, and this gap also exists between CPU memory and GPU memory [17,18]. Thus, large input can easily cause overflow with GPU or even CPU memory.

3.2 MGMR++ system design

MGMR++ extends MGMR with flexible C++ templates and its target platform is Nvidia Fermi family of GPUs. It is designed to be extensible and customizable while maintaining high occupancy of multiple GPUs. Load balancing across multiple GPUs is achieved at runtime based on hardware performance and job sizes. Every stage of the MapReduce pipeline can be re-defined by users although the default Shuffler and Sorter are provided.

When data size is smaller than accumulated GPU memory size, MGMR++ adopts single-round mode. Otherwise, multi-round mode will be applied, assuming CPU memory is big enough to contain all data.

The overview of single-round MGMR++ is shown in Fig. 1. The input of Map stage is partitioned into sets of key-value pairs assigned to workers in different GPUs simultaneously. Next, the intermediate data generated from Map stage are shuffled among workers across GPUs without going through CPU memory. The Shuffle stage incurs all-to-all communication among workers. For workers within one GPU, the communication is accomplished through commonly shared GPU global memory. On the other hand, for workers in different GPUs, GPUDirect enables remote GPU memory access without going through CPU memory for performance gains. Finally, all outputs of Reduce stage on multiple GPUs are copied back to CPU memory.

Through iterative GPU activations, MGMR++ can handle any large data set that exceeds the sum of multiple GPU memory unit sizes (but smaller than CPU memory) in multi-round mode. As shown in Fig. 2, a self-scheduling strategy is used to assign data sets to GPUs for processing. If data cannot be processed all together by GPUs, the data pool will be used as a buffer in page-locked memory on the CPU side for intermediate data. In this multi-round mode, input data is loaded to GPUs multiple times/rounds for *Map* function to generate temporary data saved back to the data pool on CPU side. Then, Shuffle stage starts. The sorted temporary data is loaded to GPUs next multiple times/rounds for *Reduce* function to generate the final results, which are sent back to CPU memory and hard disks.

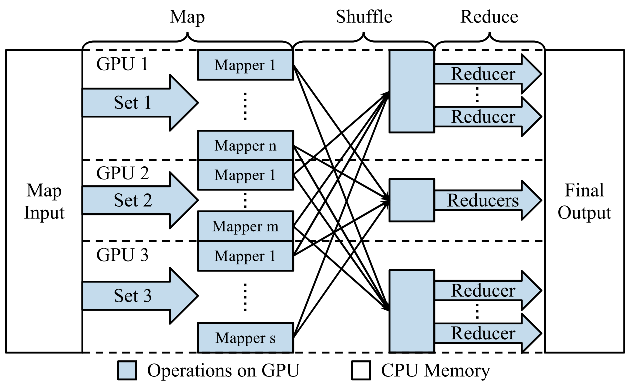


Fig. 1 MGMR++ in single-round mode with multiple GPUs

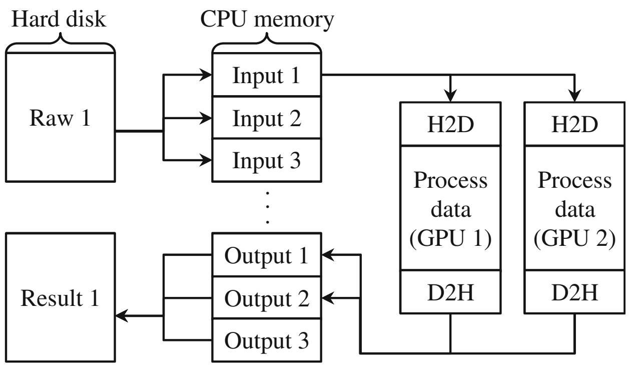


Fig. 2 MGMR++ in multi-round mode for Big Data applications (H2D: Host-to-Device Communication; D2H: Device-to-Host Communication)

In *Map* stage, the input key-value pairs are partitioned into various sets with different sizes in the data pool. When the CPU program detects an idle GPU, it will activate one *Map* function and assign one data set over. For multi-round mode, once Map workers finish the work, the intermediate results will be sent back to the data pool and another group of data sets will be loaded for processing until the Map work is done. Next, in Shuffle stage, two modes work differently. In one-round mode, input/output data is placed in GPU global memory for the shuffling among multiple GPUs and Reduce workers can continue easily, although in multi-round mode data pool is used to contain all intermediate data and shuffling takes place in CPU memory. Finally, in *Reduce* stage, data will be loaded from the data pool into GPU global memory and Reduce workers will work in a self-scheduling manner. Different from the *Map* stage, each reducer is indivisible.

3.2.1 Map stage

The *Map* stage consists of several sub-stages: Output Size Estimation, Key-Value Processing, and Partial Folder. In Output Size Estimation sub-stage, the MGMR++ estimates output size in advance to avoid memory overflow in GPU. Unlike CPU, GPU cannot dynamically allocate global memory inside kernel functions. In Key-Value Processing sub-stage, *Map* workers fetch input data from the data pool in a self-scheduling manner and execute the user-defined *Map* function. Asynchronous memory copy such as cudaMemcpyAsync() is used to overlap communication and computation, i.e., both GPUs and PCIe bus will be busy at the same time. In multi-round mode, the output data sets need to be copied back to the data pool since GPU global memory is not big enough to contain all intermediate results. Then, MGMR++ takes full advantage of bidirectional memory copies, given that two DMA engines work in opposite directions [22]. Therefore, data transfer bandwidth is doubled. Communication operations are overlapped as well. Finally, in Partial Folder sub-stage, the intermediate output data will be folded to reduce its size. This feature gives the user a way to reduce data transfer and computation overheads. I/O bound applications can use this sub-stage to reduce data transfer cost for performance gains.

3.2.2 Shuffle stage

In single-round mode, if the intermediate data (output of the *Map* stage) is small enough to put in one GPU's global memory, these key-value pairs will be sorted through radix sort in Nvidia Thrust library [23] during this stage. If they are

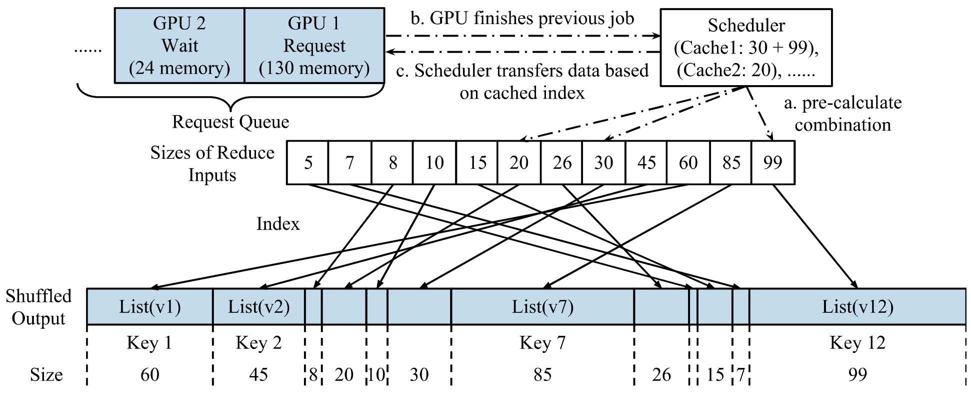

GPU 2 Wait (24 memory)
GPU 1 Request (130 memory)
......
Request Queue
b. GPU finishes previous job
c. Scheduler transfers data based on cached index
Scheduler (Cache1: 30 + 99), (Cache2: 20), ...
a. pre-calculate combination
Sizes of Reduce Inputs
5 7 8 10 15 20 26 30 45 60 85 99
Index
Shuffled Output
List(v1) List(v2) List(v7) List(v12)
Key 1 Key 2 Key 7 Key 12
Size 60 45 8 20 10 30 85 26 15 7 99

Fig. 3 The interaction between Partition Scheduler and GPUs

distributed across multiple GPUs, Parallel Sorting by Regular Sampling (PSRS) [24], also called Sample Sort, is applied to incur all-to-all broadcast through the GPUDirect technique, and all data will be redistributed among GPUs. In fact, for scalability, hashing might be a better choice. However, current GPU hashing algorithms only work well with integers and real numbers, not with text streams. For generality purposes, this work picks PSRS and leaves discussions on hashing strategy as future work.

When data is too big for aggregate GPU memory and multi-round mode has to be used, the input and output of Shuffle will be placed in the data pool (on the CPU side). A CPU partition scheduler will use PSRS to reorder key-value pairs and build indices in the data pool. GPUDirect is not necessary since the data exchange does not happen among GPUs.

3.2.3 Reduce stage

Partition Scheduler is used only if the input of Reduce stage exceeds the sum of all GPUs' memory sizes as in multi-round mode. Figure 3 shows details about how Partition Scheduler works. After Shuffle stage, Partition Scheduler maintains all value list partitions in the data pool on the CPU side. The indices of these value lists have been built in advance. As GPU is idle and its Reduce workers come to ask for more Reduce work, the Partition Scheduler will assign several value lists as a combination with the consideration of load balancing and transfer it to the designated GPU.

An approximate algorithm of the subset sum problem [25] is used while the sum is GPU memory and the subset is the sizes of different inputs. As shown in Fig. 3, while each GPU works on current reducers, multiple CPU threads concurrently calculate the possible combinations of inputs for different GPUs. So when one GPU has finished the current job, it can get another job directly from Partition Scheduler. All GPUs keep busy until all Reduce inputs are processed, and load balancing is achieved. Przydatek's algorithm [25] works fast, and therefore, it is possible that one GPU finishes its job before Partition Scheduler gets the most approximate sum. If this happens, Partition Scheduler transfers a combination of inputs based on the current result immediately to GPU instead of blocking GPU from continuous computation.

Finally, in the Key-Value Processing sub-stage, a user-defined *Reducer* function is executed in hardware threads on GPUs. Similar to the situation in *Map* stage, multiple GPUs' computation and PCIe's bidirectional communication capacities are exploited thoroughly, by utilizing advanced features in Nvidia CUDA.

3.3 PMGMR system design

PMGMR mainly consists of three functional units: a job scheduler, a GPU operation scheduler, and a configuration optimizer. They are designed to improve MGMR [5] and MGMR++ in the following aspects: capability of Big Data processing, efficient GPU utilization, and fine-tuned overall system performance.

Figure 4 shows the overview of PMGMR workflow. The job scheduler launches *Map* and *Reduce* jobs based on settings. Particularly in GPU Fermi architecture, GPU operations inside these jobs are emitted to GPU operation scheduler and issued in an optimized order. After each job finishes its execution, the recorded timeline is sent to the configuration optimizer, and the job setting is adjusted at runtime based on the analyzed results. PMGMR is designed to process massive amount of data. When the processed data is too large for GPU and CPU memory, PMGMR employs a job scheduler to handle the data transfer among hard drive, CPU memory, and multiple GPUs. The job scheduler ensures that no memory overflow will occur due to large inputs. This enables users to import any size of data without defining extra functionalities if hard disks are big enough for the data and its outputs.

The data scheduler of PMGMR uses a divide-and-conquer strategy, which separates input data into smaller pieces and merges their outputs back later. This strategy is applied to both *Map* and *Reduce* stages, since the input of *Map* stage can be partitioned into chunks of any size, whereas the input

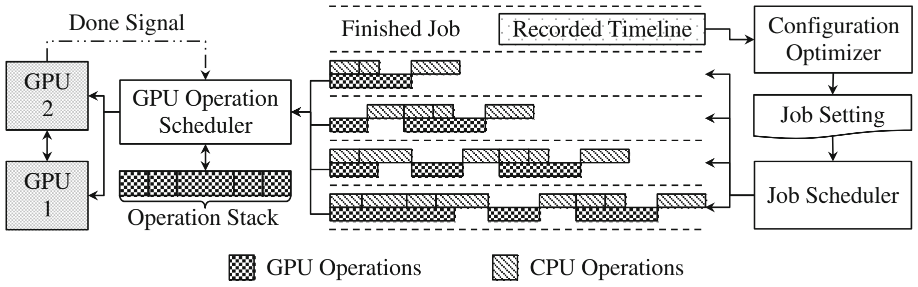


Fig. 4 The overview of PMGMR workflow in Fermi architecture

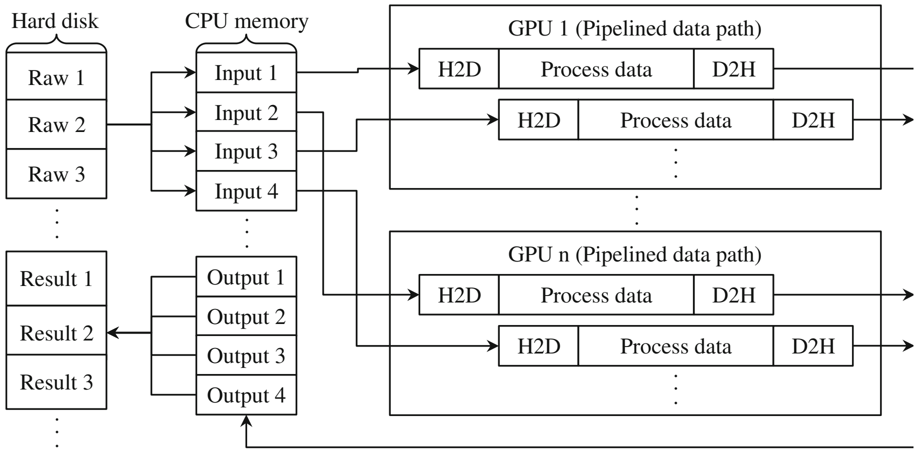


Fig. 5 The data-flow overview in PMGMR

of *Reduce* stage is only indivisible for values with the same key. The implementation of Shuffle stage is inspired by the idea of external sorting. However, instead of using CPU, the data is transferred and sorted in multiple GPUs directly.

The dataflow overview of data scheduling is depicted in Fig. 5. A part of data is loaded from hard disks into CPU memory, and then this partition is further divided into smaller chunks, whose size is small enough for GPU memory. The scheduler keeps GPUs busy with input chunks and transfers their outputs back to CPU memory. Once all chunks are processed, the outputs of the current partition are combined and written into hard disks. This process is repeated until all parts of the input in the hard drive are processed.

3.3.1 Pipelined multi-GPU utilization

Full GPU utilization is hard to achieve. For an ordinary user, designing efficient *Map* and *Reduce* functions in a GPU-based MapReduce system is a difficult task. Even with the well-designed official library by Nvidia, the average utilization of a single kernel is still around 80 percent, which means that roughly 20 percent of the computational power is wasted [23].

In order to extract full computational power from GPU, rather than relying on users and programmers to tune applications and system configuration, a runtime scheduler is developed to improve Nvidia GPU utilization through CUDA streams and Hyper-Q.

The overview of the pipelined data-path is shown in Fig. 5. The data-paths are overlapped in each GPU by using multiple CUDA streams. Consequently, PMGMR takes advantage of concurrent kernel execution since Fermi and Kepler architectures allow kernel functions from different streams in the same context to run simultaneously for idle SM utilization. Furthermore, bidirectional memory copy operations also increase the average bandwidth of data transfers since two DMA engines work in opposite directions. However, maintaining both benefits in multiple GPUs is not an easy task. Both the threshold of job pipelines and the input size need to be adjusted appropriately for better efficiency.

The control flow overview of the job scheduler is shown in Fig. 6. Before the job starts, the job scheduler collects information about available CPU memory and GPU devices. Based on the gathered information and a user-defined structure, the scheduler calculates the maximum feasible input size for CPU and different GPUs. Data then starts to be transferred back and forth between hard disks, CPU memory and GPU memory. The goal of this runtime optimization is to continuously improve the overall utilization of multiple GPUs by approaching to the optimal threshold and input size.

3.3.2 Pipelining scheme in Fermi architecture

Streams and concurrency were features in CUDA 4.0 for Fermi architecture. With GPUs' compute capability no less than 2.0, a programmer is able to launch up to 16

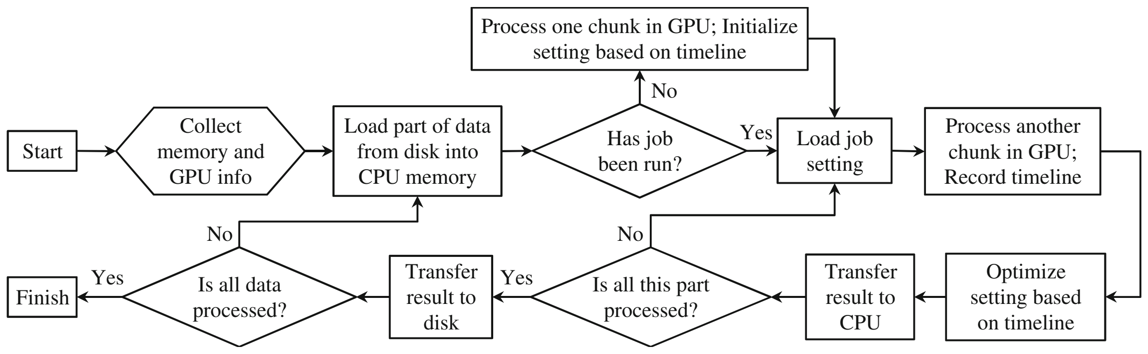

Start
Collect memory and GPU info
Load part of data from disk into CPU memory
Has job been run?
No
Process one chunk in GPU; Initialize setting based on timeline
Yes
Load job setting
Process another chunk in GPU; Record timeline
Optimize setting based on timeline
Transfer result to CPU
Is all this part processed?
No
Yes
Transfer result to disk
Is all data processed?
No
Yes
Finish

Fig. 6 Control flow of the job scheduler

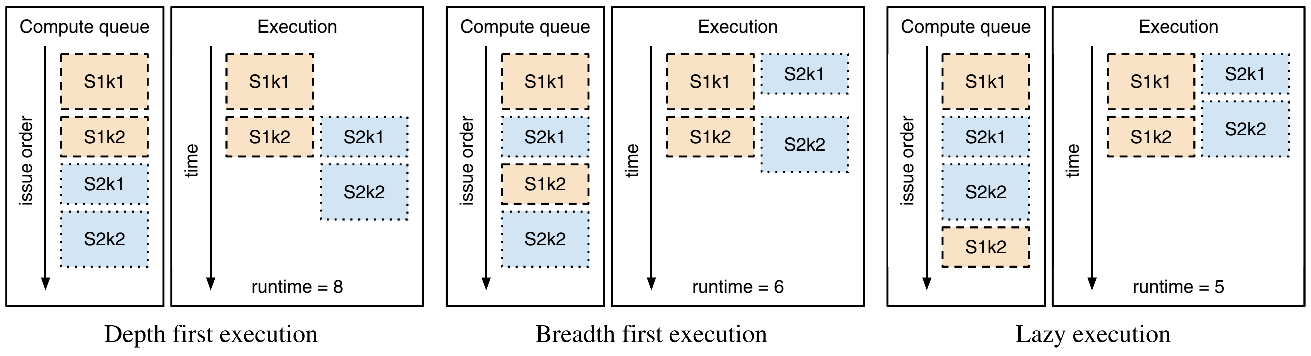


Fig. 7 Different runtime with different execution orders

concurrent CUDA kernels, and two bidirectional data transfer *cudaMemcpyAsync*() calls. As defined in Fermi architecture white paper [26], the Fermi hardware has one compute engine queue and two copy engine queues. Stream dependencies between engine queues are maintained, and become FIFO sequence within an engine queue. A CUDA operation is dispatched from the engine queue either if preceding calls in the same stream have completed or if preceding calls in the same queue are irrelevant and have been dispatched.

By appropriately using streams to achieve operation overlapping and high GPU occupancy, some applications can exhibit several times performance improvement with the same algorithm and only minor code changes. On the other hand, a blocked operation stops all other operations in the queue, even in other streams. For instance, as shown in Fig. 7, when two streams issue four kernels in different ways, the execution time varies.

To take advantage of stream and concurrency, PMGMR processes job pipelines in multiple CPU threads with different GPU streams. In each CPU thread, GPU operations such as memory copy and kernel execution are pre-defined by the user, and their issue order is optimized at compile time by NVCC compiler. However, just like other multi-threaded programs, the execution order of operations among multiple threads cannot be predicted in advance.

3.3.2.1 Reorder-and-fire scheme The goal of our reorder-and-fire scheme is to significantly reduce the number of operation combinations that cause blocking. Instead of issuing GPU operations directly, the execution plan of each operation is first sent to the GPU operation scheduler and placed into a software stack with a limited window size. Operation priorities are estimated by the scheduler based on other operations in GPU. When the pending operations on the stack exceed its window size, GPU operation scheduler sends one GPU operation with the highest priority to the hardware queue and removes the operation from the software stack. Moreover, when an operation finishes, a signal is immediately sent to the scheduler through a callback function in the wrap function.

Table 1 shows a part of the priority table for different combinations of coming operation and GPU operational state when proportions of Device-to-Host, Kernel, and Host-to-Device operations are the same. These priorities need to be adjusted for different operation proportions in different applications, which represent the possibilities of their executions.

Due to the restrictions in Fermi architecture, the possible operation combinations that create concurrency are also limited. Based on feasible and possible combinations, an efficient scheme is designed to calculate the priorities of different combinations that define the chance to increase concurrency and avoid blocking. As shown in Table 1, the foundation

Table 1 An example of the priority table when proportions of Device-to-Host, Kernel, and Host-to-Device operations are the same

Remained	D2H	K	H2D	D2H, H2D	D2H, K	H2D, K	D2H, K, H2D	D2H,D2H, K	. . .
Coming									. . .
D2H	**1**	**2**	**2**	3	3	4	5	4	. . .
K	**2**	**0|2**	**2**	4	2	2	6	4	. . .
H2D	**2**	**2**	**1**	3	4	3	5	6	. . .

D2H: Device-to-Host K: Kernel H2D: Host-to-Device

Fig. 8 Pseudo code for operation priority calculation

Algorithm 1 Operation priority formula

```
 1: procedure PRIORITY($coming, remain$)          ▷ Coming and remaining operations
 2:     $total \leftarrow 0$
 3:     $count \leftarrow amount(remain)$
 4:     for $i = 1 \rightarrow count$ do
 5:         $this \leftarrow checkPair(coming, remain[i])$      ▷ Priority of each operation pair
 6:         if $i = count$ then
 7:             if $coming = K$ & $remain[i] = K$ then          ▷ K represents Kernel
 8:                 $this \leftarrow 0$
 9:             end if
10:         end if
11:         $total \leftarrow total + this$
12:     end for
13:     return $total$                                  ▷ Total priority of all pairs
14: end procedure
```

priorities are the nine pairs, which have one coming and one remaining operation. These nine priorities are defined according to these four situations:

(1) *cudaMemcpyAsyncs*() operations in the same direction are serialized (1 credits),
(2) Bidirectional *cudaMemcpyAsyncs*() operations are overlapped (2 credits),
(3) Sequentially issued kernels delay signals and block *cudaMemcpyAsyncs*()(0 credits),
(4) Kernels in different streams can be executed concurrently when resources are available (2 credits)

In situation 1, although memory copies are serialized, kernels and memory copies in the opposite direction can still run concurrently. Both situations 2 and 4 are the perfect cases to increase concurrency. Lastly, situation 3 is a very special case and actually the worst case, which shows that inappropriate concurrent kernel executions can block other operations. The solution is to insert memory copy between the sequential kernels. These situations represent four unique circumstances, and only situation 3 relies on the issue order. Thus, for the GPU scene, which has more than one remaining operations, its priorities can be calculated by adding all priorities of its one-to-one pairs together while distinguishing two different cases of situation 3. The generalized scheme is described as the algorithm shown in Fig. 8.

3.3.2.2 Execution plan queue As a requirement of previous schemes, the runtime scheduler needs to have the capability of reordering and issuing the user's execution plans. To achieve this, two main issues should be considered. Firstly, the definition of the user's kernel function is unknown, and Secondly, user-defined GPU operations from different streams need to be reordered at runtime.

To handle abovementioned issues, PMGMR employs the most recent version of the standard C++ programming language, C++11, which includes several additional features to the core language and extends the C++ standard library. Then, user-defined functions can be stored as an object which can be moved around and copied. In this way, the wrap functions and their operation types are sent to the runtime scheduler. Based on the operation types, the scheduler can now calculate their priorities and issue them in an optimized order.

3.3.3 Pipelining scheme in Kepler architecture

Kepler GPU family has improved concurrency functionality with the new Hyper-Q feature, which increases the total number of connections (work queues) between the host and the CUDA Work Distributor logic in GPU by allowing 32 simultaneous, hardware-managed connections (compared to the single connection available with Fermi). Therefore, up to 32 streams can be totally independently executed in Kepler GPUs.

PMGMR takes great advantage of Hyper-Q to process job pipelines in different CPU threads. Since GPU operations in different GPU streams are actually maintained in different hardware connections, PMGMR no longer needs to determine the cross dependency issue for task serialization, as in Fermi GPU family.

Thence, the utilization of Kepler GPU is always higher and more stable because concurrency and bidirectional data transfer can be achieved easily.

3.3.3.1 Runtime tuning and load balancing For the job scheduler, job pipelines are issued according to two options in job setting: threshold of job pipelines and the input size of each job. These two options can be adjusted by the configuration optimizer based on the timeline records and the detection of memory overflow.

The threshold option represents the number of running CPU threads which form the job pipelines and process input data for user-defined *Map* and *Reduce* functions. If the number is too small, concurrency cannot be high enough to fully utilize multiple GPUs. However, if too many job pipelines are generated, the performance can also decrease due to thread context switch latency. PMGMR adjusts this option based on the processing throughput. Initially, the number of threshold is small, assuming it is N. After N job pipelines are launched, the configuration optimizer increases this number and keeps track the processing throughput for any necessary adjustment.

The input size option can be carefully configured to maximize the usage of GPU memory and avoid freezing due to memory overflow. Initially, the size of input/output equals to *GPUMemory/PipelineNumber*, and no memory overflow may occur in this situation. For the purpose of maximizing GPU occupancy, the configuration optimizer gradually increases the input size with fixed threshold. Then the possibility of memory overflow also increases. Therefore, before any GPU operation is issued, the wrap function checks the GPU memory usage to determine whether this operation can be issued or not. If one operation may cause GPU memory overflow, its corresponding job pipeline will be blocked until it is safe to issue this operation.

Performance may degrade due to the blocked CPU operations and low GPU utilization. Therefore, when job scheduler detects a memory overflow operation, the setting of input size is rolled back to the previous state, and the maximum GPU utilization is reached for this job.

4 Experimental results

4.1 Results on MGMR++ scheme

Experiments for this system design were conducted on a server containing two Intel Xeon X5660 (2.80GHz, totally 24 cores) with 24 GB RAM and two Nvidia GPUs: Quadro 6000 and Tesla C2070 (both have 1.15 GHz, 5,375 MB global memory, 64KB L1-cache/SM) for Fermi GPU testing.

The server is running the GNU/Linux operating system with kernel version 2.6.32. Testing applications are implemented with CUDA 5.0 and compiled with NVCC compiler in CUDA Toolkit 5.0. CPU versions are implemented with OpenMP using 24 threads to utilize all 24 CPU cores for full capacity.

Traditional MapReduce is designed for data-intensive applications and High Throughput Computing (HTC) since data can be easily distributed in computer clusters. However, GPU computing is good for compute-intensive applications and High Performance Computing (HPC) due to the data transfer slowdown on PCIe bus. Multi-GPU MapReduce systems, MGMR++ and PMGMR, intends to integrate both cases. Some data-intensive might not be able to achieve better performance in terms of execution time (due to CPU-GPU communication overhead). But HTC effect is always there. In this work, K-Means Clustering (KMC) [27] and Unique Phrase Pattern (UPP) are selected to represent compute-intensive and data-intensive applications.

4.1.1 K-Means Clustering

K-Means Clustering (KMC) [27] is used in data mining, which aims to partition n observations into k clusters where all observations in a cluster are close to the nearest mean. The testing data is randomly generated from a 10 k $\times$ 10 k square area with floating-point coordinates. Map stages finds the cluster for each point based on means and emit <index (cluster), point>. Since KMC is NP-hard, we set the test to 3 rounds for performance measuring. Also, the cluster number k is set to 24 for fair comparison between CPU and GPUs since there are totally 24 CPU cores. As shown in the left part of Fig. 9, we generated up to 520 millions points for the experiments. KMC is quite computationally intensive since each point needs to compare with 24 means to find the closest one.

The Multi-GPU version can declare clear computability advantage. In the right part of Fig. 9, the double-GPU version achieves 91.7 times speed-up over the CPU version and 1.7 times speed-up over the single-GPU version. As shown in Fig. 10, the Shuffle stage takes a small percentage of execution time because of the optimization of Partial Folder substage. For the same reason, the Reduce stage is also very light-weighted.

4.1.2 Unique-Phrase Pattern

Unique Phrase Pattern (UPP) can detect the most frequently used phrase patterns. For simplicity (buffer management), only two to three-word phrases are acceptable for our tests. The input data is randomly generated from a forty-thousand-

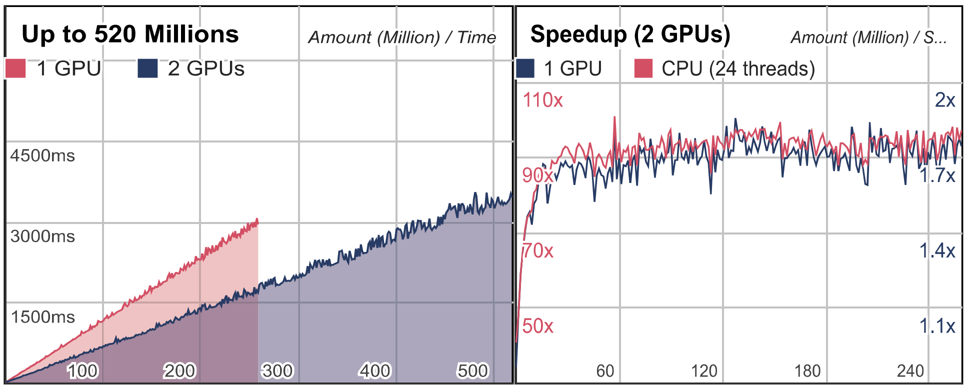


Fig. 9 Experimental results of KMC: execution time and the speedups of the double-GPU version over the 24-thread CPU version and the single-GPU version

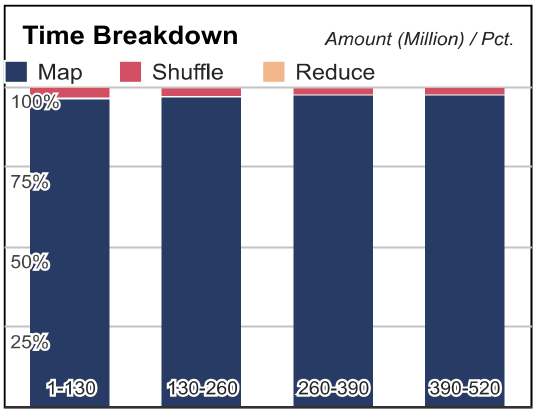


Fig. 10 Experimental results of KMC: runtime breakdown

word dictionary that is pre-hashed. In MGMR++, UPP is developed as a three-pass MapReduce. Therefore, no extra work is needed for allocating different sizes of buffers for different phrases. The first two passes count 2-word and 3-word phrases separately. Key-value pairs are emitted as <list (hash(word$_1$), ...),1>. Both results are used as the input for the third pass in order to sort all phrases in one mapper for their occurrence. Since UPP is originally I/O-bound, the sub-stage Partial Folder in the Shuffle stage is activated in each pass to reduce the data transfer overhead.

In the left one of Fig. 11, performance comparison is given for one- and two-GPU cases. The single-GPU version is only slightly faster than double-GPU version when the input size is very small (less than 45 MB), due to low GPU occupancy and communication overhead in double-GPU version's Shuffle stage. Bigger applications can exploit parallelism more efficiently. In the right one of Fig. 11, although double-GPU version still remains a similar advantage over single-GPU one, the advantage over CPU-version drops. Since UPP is an I/O-bound application, CPU-version takes advantage of this. According to the runtime breakdown as shown in Fig. 12, although *Map* stage is the most time-consuming portion, Shuffle and Reduce stages exhibit larger percentage than those in KMC. Still, application types are the main reason here.

4.2 Results on PMGMR scheme

Experiments for this design scheme were conducted on two servers. The server 1 contains two Intel Xeon E5504 (2.00GHz, total 8 cores) with 24 GB memory and two Nvidia Tesla C2050 cards (1.15 GHz, 448 CUDA Cores, 2,687 MB global memory) for Fermi GPU testing, and server 2 contains four Intel Xeon E5-2620 (2.00GHz, total 24 cores) with 32 GB memory and two Nvidia Tesla K20Xm (0.73 GHz, 2688 CUDA Cores, 5,760 MB global memory) for Kepler GPU testing. Both servers are running the GNU/Linux operating system with kernel version 2.6.32. Testing applications are implemented with C++11 and CUDA 5.0 and compiled with g++ and NVCC compiler in CUDA Toolkit 5.0.

4.2.1 PMGMR vs. MGMR++

The pipelining scheme of PMGMR utilizes multiple GPUs mainly in two ways: concurrent kernel execution and bidirectional memory copy. K-Means Clustering (KMC) is used in data mining, which aims to partition n observations into k clusters where all observations in a cluster are close to the nearest mean. We use KMC to test the pipelining scheme because of its computation-heavy characteristic which can clearly demonstrate the advantage of GPU-based MapReduce schemes. Concurrent kernel execution is originally designed for inefficient kernels. If the performance of KMC can still be improved by pipelining scheme, then this scheme can perform even better in other benchmarks with inefficient kernels which originally waste the computation power of multiple GPUs.

As shown in the left part of Fig. 13, the pipeline scheme has considerable speed-up over the sequential execution when the number of input pairs is small. As the number grows bigger, the speedup becomes smaller in both Kepler and Fermi versions because each kernel has already consumed most of GPU resources. However, the execution time is still reduced by asynchronous memory copy because most of the memory copy operation is overlapped with computations.

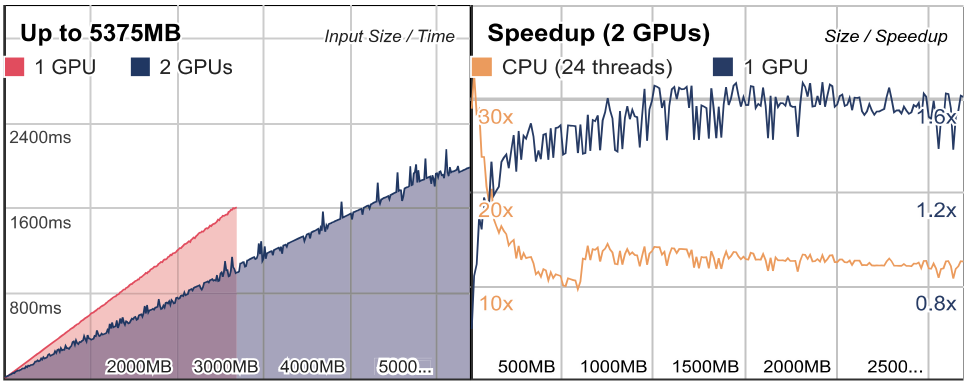


Fig. 11 Experimental results of UPP: execution time and the speedups of the double-GPU version over the 24-thread CPU version and the single-GPU version

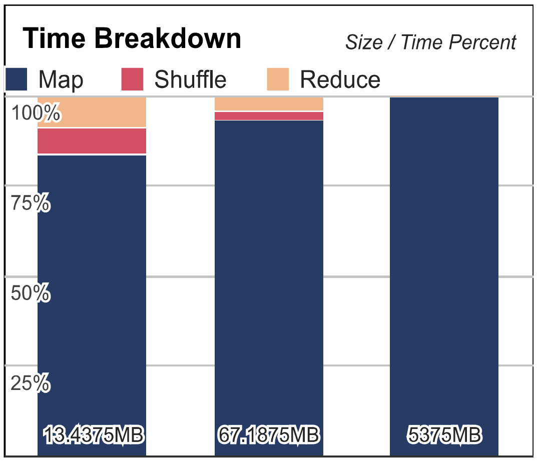


Fig. 12 Experimental results of UPP: runtime breakdown

As shown in the right part of Fig. 13, Kepler GPU family gains more benefits from the pipelining scheme than Fermi does. The reason is that both hardware connections and level of concurrency are improved in Kepler. As a result, when the input and output data sets can fit GPU memory, the Kepler version of PMGMR achieves 1.8 times speedup over Fermi version, whereas the Kepler version of MGMR++ only achieves 1.6 times speedup.

4.2.2 Overall performance

A 60GB binary file, which contains the original coordinate information of KMC, is generated to measure the overall performance of PMGMR. Since KMC is NP-hard, we set the maximum rounds to 3 for performance measuring. In this test, the first of several rounds has relatively low execution time. Though, as the size of input increases, and actually before this size can incur CPU memory overflow, the execution time starts to increase dramatically in the coming rounds. Then, after few rounds, the execution time becomes stable again with a much lower throughput. The main reason of this is that Linux maintains buffer caches in its file system.

Originally, most of data that PMGMR reads are from the caches that reside in CPU memory. After all the cache are flushed out from CPU memory because of the memory operations, PMGMR starts to read data from hard drive which has very slow read/write speed and actually becomes a main bottleneck.

As shown in the left one of Fig. 14, when the input size is roughly less than 3 GB, the execution time is so small and can almost be ignored if we compare it with the following execution time. This phenomenon comes from buffer caches in memory. When data is small, it might be left in buffer caches. Future data might have been pre-fetched and no I/O operation might be incurred. However, as data size increases, buffer caches will not be large enough and I/O operations will be required all the time. The detail is out of the scope of this paper and we only consider the stable larger data cases. After that, the performance of PMGMR is mainly bounded by hard disk speed. Since the read/write combined speed of hard drive is 105 MB/s in Fermi machine and 163 MB/s in Kepler machine, PMGMR shows very low operation overheads while continuously processing input and storing output. The difference of execution time between Fermi and Kepler machines is also caused by different hard drive speeds. Moreover, in the right one of Fig. 14, the runtime breakdown shows that as the input size increases, the proportion of shuffle and reduce stages do not increases. The reason is that a *Partial Reduce* stage is used for each portion to reduce I/O. Thus, only the sum of the *x*–*y* coordinates of each cluster in each portion is written into the output file of the *Map* stages. Since the number of clusters is limited, total size of the output files is always smaller compared to the input of the Map stage. Thus *Shuffle* and Reduce stages become very light-weighted.

4.2.3 Runtime optimization

A GPU operation scheduler and a setting optimizer are employed to improve the stability and throughput in PMGMR. The upper-left one in Fig. 15 shows the execution time comparison between normal multi-threaded Fermi version and scheduled multi-threaded Fermi version. The experimental

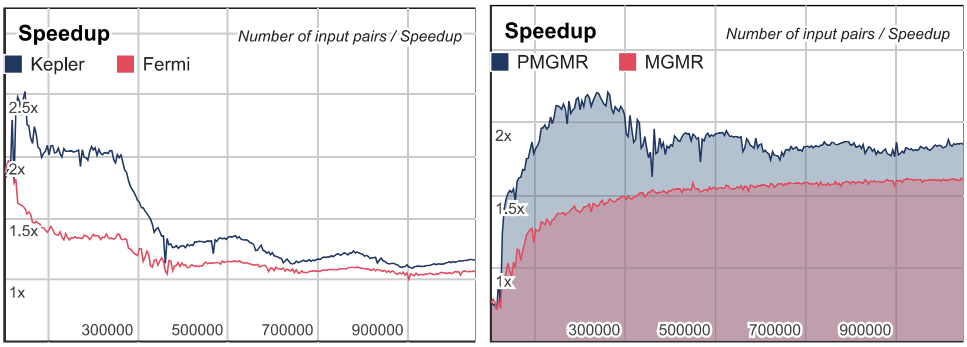


Fig. 13 Two speedups: The left part is PMGMR's runtime speedup over MGMR++; while the right one is the speedup of the Kepler versions of PMGMR and MGMR++ over their corresponding Fermi versions

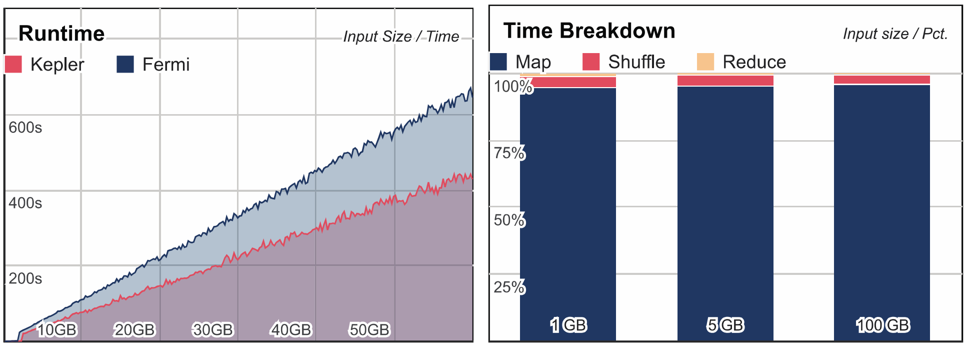


Fig. 14 PMGMR with K-means Clustering: execution time and runtime breakdown

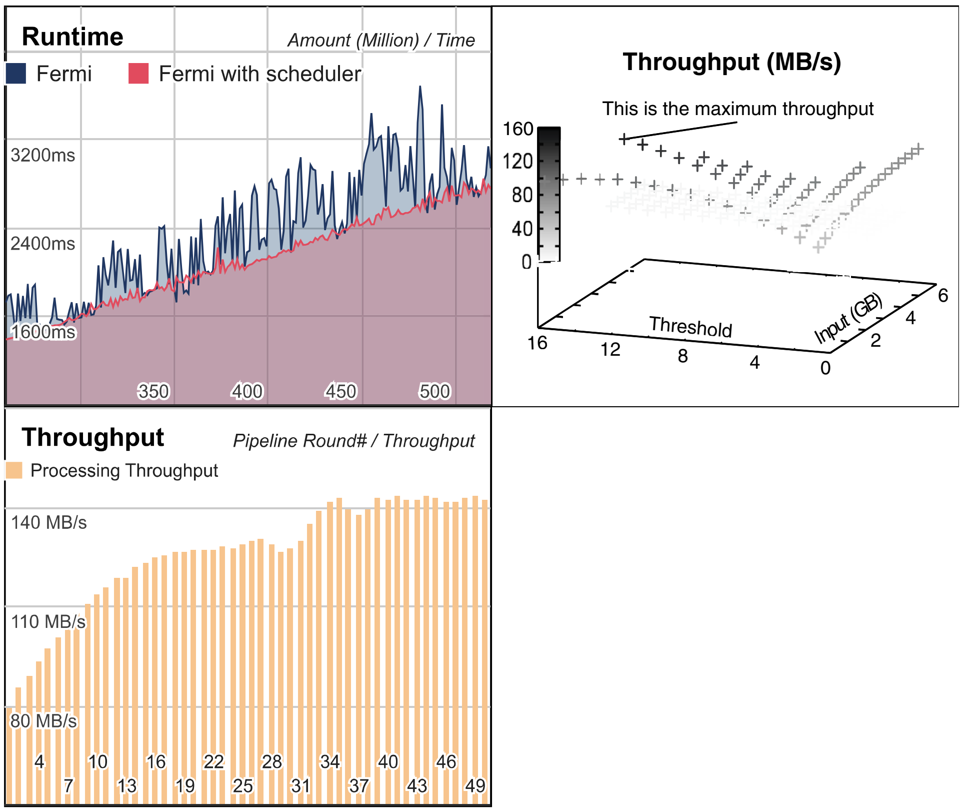


Fig. 15 Runtime optimization results: performance of Fermi scheduler, throughput records from different rounds, and throughput with different sizes of input and threshold

result shows that compared to the normal Fermi version, the scheduled one is much stable. Although sometimes the normal Fermi scene shows shorter execution time, the average performance of the scheduled version is still higher. The performance fluctuation might come from Fermi hardware and system configuration.

The setting optimizer mainly adjusts threshold of job pipelines and data inputs to optimize the setting. Experimental results show that setting optimizer works well as it is designed. As shown in the upper-right one of Fig. 15, threshold of job pipelines and the input size of each job can be adjusted independently for better throughput. After setting them with different values, the maximum throughput can be achieved without any memory overflow. However, this 3-D figure only shows that both factors can be adjusted and the peak position (along *Z*-axis) is detected for the maximum throughput.

PMGMR can approach to a near-perfect setting automatically. As shown in the lower-left portion of Fig. 15, the throughput roughly keeps increasing until the 22nd round. Next, the throughput starts to drop because too many job pipelines are generated. At 29th round, the setting optimizer detects the dropping throughput and rolls back to the previous threshold option. The throughput returns to the previous level and the threshold option is fixed. After that, the size of input is gradually increased every round until 36th round. A memory overflow is detected and the operation blocked by the job scheduler. Therefore, the option of input is also rolled back to the previous state. Finally, the throughput becomes stable, and both options are finalized for the current MapReduce stage.

5 Conclusions and future work

In this paper, two multi-GPU MapReduce systems were designed and further implemented with consideration of both GPU computing power and storage hierarchy for large-scale data processing. Experimental results have demonstrated the effectiveness of these two systems.

MGMR++ and PMGMR system are developed aiming scalability in both computational power and data size aspects. As data size is larger than the aggregate GPU memory, CPU memory is used to continue MapReduce computations. When CPU memory is not big enough, hard disks can be used for large-scaled MapReduce. Several runtime schedulers are developed to help improve overlapping of computation and communication operations. System configuration can also be set automatically for a near-perfect result.

Experimental results have demonstrated MGMR++'s advantages over both CPU and single-GPU MapReduce in both performance and scalability aspects, while PMGMR outperforms MGMR++ in capability, performance, and stability. Both systems are compared on Fermi and Kepler GPU families.

By carefully tuning factors in both architecture design schemes, the overall performance of multi-GPU MapReduce can be improved for well-known benchmarks, and is thus more attractive to many other existing applications. Our results show that it is therefore possible to build a multi-GPU environment aiming at MapReduce computations for elastically scalable and efficient Big Data processing. Although storage bandwidth might turn into a bottleneck in Big Data applications, scaled GPU clusters will help achieve larger aggregated memory. Even for a single GPU node, reduced GPU usage will benefit other running applications since almost all operating systems are multi-user and multi-task ones. These systems can provide nicer sharing environments.

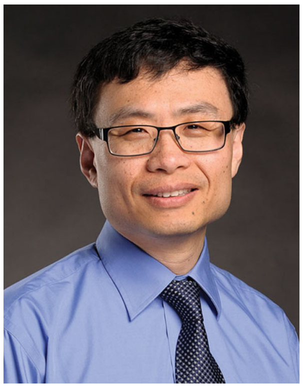
The future work includes extending PMGMR to GPU Clusters by using RDMA and Hyper-Q for further performance scalability, developing generalized hash function for Shuffle stage, integrating PMGMR with distributed file systems and storages for fault tolerance, and improving its easy-to-use aspect with the newest C++11 standard for programmability. More real-world applications will be applied for detailed performance analysis. Hadoop Streaming is one of future branches for testing PMGMR and MGMR++ in GPU-based cluster environments.

Acknowledgments Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies or institutions. This research is based upon work partially supported by National Science Foundation/USA under grant No. 0959124, Ministry of Science and Technology/Taiwan (MOST)/Taiwan under grant MOST 103-2221-E-126-010-, The Providence University research project under grant PU102-11100-A12 and NVIDIA through CUDA Center Awards.

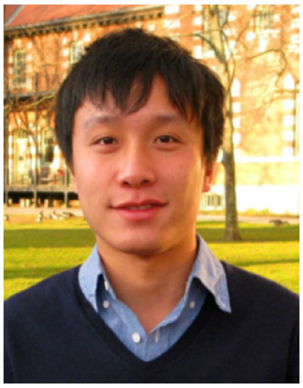
References

1. Jiang, H., Chen, Y., Qiao, Z., Li, K.-C., Ro, W., Gaudiot, J.-C.: Accelerating MapReduce framework on multi-GPU systems. Cluster Computing, pp. 1–9. Springer, Berlin (2013)
2. Cubieboards: an Open ARM Mini PC, http://www.cubieboard.org 2014
3. CUDA Programming Guide 6.0, NVIDIA, 2014
4. Dean, Jeffrey, Ghemawa, Sanjay: MapReduce: simplied data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
5. Chen, Y., Qiao, Z., Jiang, H., Li, K.-C., Ro, W.W.: MGMR: multi-GPU based MapReduce. Grid and Pervasive Computing. Lecture Notes in Computer Science, vol. 7861, pp. 433–442. Springer, Berlin (2013)
6. Bollier, D., Firestone, C.M.: The Promise and Peril of Big Data. Communications and Society Program. Aspen Institute, Washington, DC (2010)
7. Jinno, R., Seki, K., Uehara, K.: Parallel distributed trajectory pattern mining using MapReduce. In: Proceedings of IEEE 4th International Conference on Cloud Computing Technology and Science, pp. 269–273, 2012
8. Lee, D., Dinov, I., Dong, B., Gutman, B., Yanovsky, I., Toga, A.W.: CUDA optimization strategies for compute-and memory-bound neuroimaging algorithms. Comput. Methods Programs Biomed. **106**, 175 (2012)
9. Raina, R., Madhavan, A., Ng, A.D.: Large-scale deep unsupervised learning using graphics processors. In: Proceedings of the 26th International Conference on Machine Learning, Canada, 2009
10. Fadika, z., Dede, E., Hartog, J., Govindaraju, M.: Marla: Mapreduce for heterogeneous clusters. In: Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pp. 49–56, 2012
11. Stuart, J.A., Owens, J.D.: Multi-GPU MapReduce on GPU clusters. In: Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium, pp. 1068–1079, 2011

12. Foster, I., Kesselman, C.: The Grid 2: blueprint for a new computing infrastructure, Morgan Kaufmann, 2003
13. Czajkowski, K., Fitzgerald, S., Foster, I., Kesselman, C.: Grid information services for distributed resource sharing. In: Proceedings of 10th IEEE International Symposium on High Performance Distributed Computing, pp. 181–194, 2001
14. White, T.: Hadoop: The Definitive Guide. O'Reilly Media, Sebastopol (2012)
15. Chen, L., Huo, X., Agrawal, G.: Accelerating MapReduce on a coupled CPU-GPU architecture. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis 2012
16. Nakada, H., Ogawa, H., Kudoh, T.: Stream processing with big data: SSS-MapReduce. In: Proceedings of 2012 IEEE 4th International Conference on Cloud Computing Technology and Science, pp. 618–621, 2012
17. Ji, F., Ma, X.: Using shared memory to accelerate MapReduce on graphics processing units. In: Proceedings of the IEEE International Parallel & Distributed Processing Symposium, pp. 805–816, 2011
18. Chen, L., Agrawal, G.: Optimizing MapReduce for GPUs with effective shared memory usage. In: Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing, pp. 199–210, 2012
19. Shainer, G., Ayoub, A., Lui, P., Liu, T., Kagan, M., Troot, C.R., Scantlen, G., Crozier, P.S.: The development of Mellanox/NVIDIA GPU Direct over InfiniBand new model for GPU to GPU communications. Computer Science-Research and Development, pp. 267–273. Springer, Berlin (2011)
20. Fang, Wenbin, He, Bingsheng, Luo, Qiong, Govindaraju, Naga K.: Mars: Accelerating MapReduce with Graphics Processors. IEEE Trans. Parallel Distrib. Syst. **22**(4), 608–620 (2011)
21. Elteir, M., Lin, H., Feng, W.C., Scogland, T.R.W: StreamMR: an optimized MapReduce framework for AMD GPUs. In: IEEE 17th International Conference on Parallel and Distributed Systems, pp. 364–371, 2011
22. Tuning CUDA Applications for Kepler, http://docs.nvidia.com/cuda/kepler-tuning-guide/
23. Nathan, B., Jared, H.: Thrust: a productivity-oriented library for CUDA. In: GPU Computing Gems: Jade Edition, Morgan Kaufmann, pp. 359–371, 2011
24. Xiaobo, L., Paul, L., Jonathan, S., John, S., Sze, W.P., Hanmao, S.: On the versatility of parallel sorting by regular sampling. Parallel Comput. **19**(10), 1079–1103 (1993)
25. Bartosz, P.: A fast approximation algorithm for the subset-sum problem. Int. Trans. Oper. Res. **9**(4), 437–459 (2002)
26. FERMI Compute Architecture White Paper, Nvidia
27. Shi, Y., Léon-Charles, T., De, M.B., Yves, M.: Optimized data fusion for kernal k-means clustering. IEEE Trans. Pattern Anal. Mach. Intell. **34**(5), 1031–1039 (2012)



Hai Jiang received his B.S. degree from Beijing University of Posts and Telecommunications, China, M.A. and Ph.D. degrees from Wayne State University, Detroit, MI, USA. He is an Associate Professor in the Department of Computer Science at Arkansas State University, USA. Dr. Jiang is a professional member of ACM and IEEE computer society. He has published one book and more than 70 papers in refereed journals and conference proceedings as well as several book chapters. He has been involved in 60 conferences and workshops as a program/workshop/session chair or as a program committee member. His current research interests include parallel & distributed systems, computer & network security, high performance computing and communication, and modeling & simulation.



Yi Chen is a graduate student in the Department of Computer Science at Arkansas State University. He received his Bachelor's degree in Information and Computing Science from South-Central University of Nationality. Yi's current research interests include high performance parallel system, big data and cloud computing. He recently works on multi-GPU MapReduce system using GPU clusters.

Zhi Qiao is a master's student with an emphasis in Distributing System. He received his B.A. from Beijing Normal University, Zhuhai, China. He currently works under Dr. Hai Jiang. His most recent interest including enhance MapReduce performance on GPU environment and handling Big Data on GPU cluster.

Tien-Hsiung Weng is currently an associate professor at the Department of Computer Science and Information Engineering at the Providence University in Taichung, Taiwan. He received the PhD in Computer Science from University of Houston, Texas, USA. From 2010 to 2013, he was a department chair of Computer Science and Information Engineering at the Providence University. His research interests include parallel programming model, performance measurement, and compiler analysis for code improvement.

Kuan-Ching Li is currently a Professor in the Department of Computer Science and Information Engineering at the Providence University, Taiwan. He received the PhD and MS in Electrical Engineering and Licenciatura in Mathematics from University of Sao Paulo, Brazil. He was a chair in 2009 and the Special Assistant to the University President since 2010. His research interests include networked computing, parallel software design, and performance evaluation and benchmarking. He is a senior member of the IEEE and a Fellow of the IET.