SPMario: Scale Up MapReduce with I/O-Oriented Scheduling for the GPU

Yang Liu, Hung-Wei Tseng*, Steven Swanson
Department of Computer Science and Engineering
University of California, San Diego, La Jolla, CA, U.S.A.
{yal036, h1tseng, swanson}@cs.ucsd.edu

***Abstract*—The popularity of GPUs in general purpose computation has prompted efforts to scale up MapReduce systems with GPUs, but lack of efficient I/O handling results in underutilization of shared system resources in existing systems. This paper presents SPMario, a scale-up GPU MapReduce framework to speed up job execution and boost utilization of system resources with the new I/O Oriented Scheduling. The evaluation on a set of representative benchmarks against a highly-optimized baseline system shows that for the single job cases, SPMario can speedup job execution by up to 2.28×, and boost GPU utilization by 2.12× and 2.51× for I/O utilization. When scheduling two jobs together, I/O Oriented Scheduling outperforms round-robin scheduling by up to 13.54% in total execution time, and by up to 12.27% and 14.92% in GPU and I/O utilization, respectively.**

I. Introduction

The popularity of GPUs in general purpose computation has prompted efforts to scale up MapReduce systems with GPUs that have the massive parallelism and high inner-bandwidth [16]. However, lack of efficient I/O handling and scheduling in existing systems may lead to resource contention [15] among processes/tasks, resulting in underutilization of system resources. Simply increasing the number of processes/tasks cannot solve this problem. In fact, our profiling of a state-of-the-art GPU MapReduce framework reveals that, even with multiple processes, the GPU utilization can be lower than 20% while the I/O device is idle for almost 50% of the time. This inefficiency will undermine the potential performance gains from scaling up MapReduce with GPUs.

In this paper,we present *SPMario*, a GPU framework that scales up MapReduce with optimized I/O handling and task scheduling to address the above problems. SPMario runtime includes a scheduler that dispatches tasks to containers, and multiple containers that provide the execution environment for the tasks and enforce the execution order. SPMario proposes *I/O Oriented Scheduling* to coordinate task execution in a way that minimizes idle time of the resources, and pipelines the I/O operations, the data transfers between the host and the GPU, and the GPU kernel execution.

Our experiment results show that for the single job cases, SPMario can achieve up to a 2.28× speedup over the baseline in job execution time, and yield up to 2.12× GPU utilization and 2.51× I/O utilization. When scheduling two jobs together, SPMario with I/O Oriented Scheduling outperforms with round-robin by up to 13.54% in execution time, and 12.27% and 14.92% in GPU and I/O utilization, respectively.

*Current Affilication is Department of Computer Science, North Carolina State University.

This work was supported in part by the Center for Future Architectures Research (C-FAR), one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

II. Background and Motivation

This section provides a brief overview of the GPU and MapReduce, and discusses the problem of I/O handling and resource underutilization.

A. The GPU and MapReduce

The GPU is a processor comprised of multiple SIMD streaming multiprocessors, or SMs. It organizes threads into thread blocks that execute functions called kernels, and the current hardware cannot preempt a running kernel.

MapReduce [9] represents another form of parallelism to process data on a large scale. To write a MapReduce application (a ”job”), users just need to provide two primitive functions called *Map* and *Reduce*. The MapReduce framework will apply the Map function on input data blocks (”chunks”), and run the Reduce function to produce the results.

B. I/O Handling and Resource Underutilization

There are two important problems not well addressed by the existing GPU MapReduce systems: the efficient I/O handling and the resource utilization. Most existing GPU MapReduce frameworks either assume the entire datasets loaded into the host DRAM already, or leave the hassle of handling I/O to the operating system. In addition, few GPU MapReduce solutions aim to pursue better utilization of system resources. Thus, many systems leave individual GPUs underutilized[1].

To improve the resource utilization and performance, we design and implement SPMario based on three observations:

1) Simple multi-tasking might marginally improve resource utilization and the performance, but could result in serious resource contention [15], which will in turn prevent further improvement.
2) A task kernel cannot proceed before its input chunk is ready on the GPU.
3) For typical GPU MapReduce tasks, a single chunk with the size of 64MB or 128MB is enough to saturate the I/O bandwidth, even for a PCIe SSD.

[1]We define the GPU utilization as: *utilization* = $\frac{active\_time}{total\_time}$, where *active_time* measures the accumulated time when SMs are running, and *total_time* is the total time span of the job execution. We also apply this definition to describing the I/O utilization in this paper.

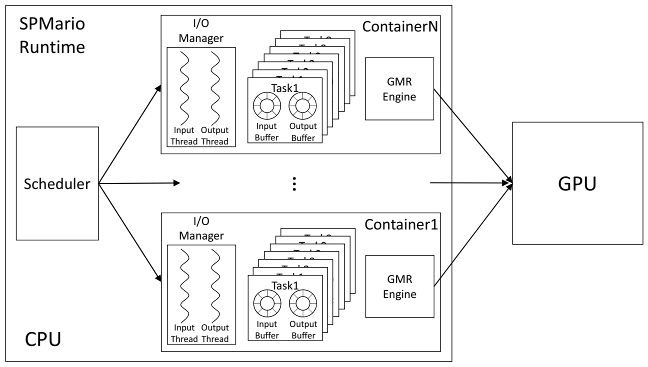

Fig. 1: **System Architecture.**

III. Design and Implementation

As depicted in Figure 1, SPMario includes a global *scheduler*, and one or more *containers*, each with a *GPU MapReduce Engine*. We explain the details below.

A. Scheduler

The centralized scheduler accepts jobs from users and dispatches tasks to containers. It can implement different scheduling policies like priority-based and round robin by controlling the number of containers a job can have and the I/O quota for each container. SPMario iteratively processes tasks, thus keeps one mapper and/or one reducer for each task. The scheduler picks at most one container to run at any time.

B. Container

A container, analogous to a node in the scale-out configuration, accepts a list of tasks from the scheduler and provides the task execution environment. SPMario reads inputs from SSDs into fixed-size chunks (default is 128MB and configurable). Unlike other MapReduce systems, in SPMario it is the dedicated input thread inside the I/O manager that performs I/Os. The I/O thread allows scheduled tasks to take turns enjoying the full SSD bandwidth within the assigned I/O quota while avoid the I/O contention. By choosing the order of supplying input data, the container is able to enforce the execution order of tasks. The I/O manager maintains two *chunk buffers* allocated on the pinned host DRAM for each *running* task. SPMario optimizes data transfer by overlapping I/O operations with asynchronous GPU operations, and assigns a separate CUDA stream for each chunk to allow multiple streams to run in parallel.

C. GPU MapReduce Engine

After a task gets a chunk, it continues to invoke the GPU MapReduce Engine to process the chunk. We implement the GPU MapReduce Engine ("GMR Engine") based on GPMR [16]. GMR Engine has four stages: *Map*, *Bin*, *Sort* and *Reduce*. The mapper runs map kernels to emit key-value pairs. GMR Engine then partitions the key-value pairs into buckets, and sends them to the right reducer in the Bin stage, where a binner thread will collect and distribute the pairs. The sorter sorts the pairs, and the reducer produces outputs.

Algorithm 1: Container Scheduling.

```
Given the current scheduling policy $Pl$:
for container $c \in C$ do
    $pr \leftarrow$ getPriority($c, Pl$) ;
    $quota \leftarrow$ getQuota($c, pr$) ;
    $status \leftarrow$ scheduleContainer($c, quota$) ;
    updateStatus($status$) ;
```

Algorithm 2: Task Scheduling inside a Container.

```
Define:
$Rt = (kernelTime + dataTransferTime)/IOTime.$
$Qt$, the queue for tasks not scheduled.
$Qr$, the queue for tasks scheduled to run.
while true do
    $quota \leftarrow$getQuota();
    $deposit \leftarrow 0$;
    while $quota$ do
        if $Qt$ is empty then
            $quota \leftarrow 0$;
            return the control to the scheduler;
        else
            if first chunk in the current round then
                //Select the task $T$ with
                //the highest priority and max $Rt$
                $(T, Rt, cquota)$ ←getTaskRt($Qt$);
            else
                $deposit \leftarrow deposit - 1$;
                if $deposit \geq 1$ then
                    //Select the task $T$
                    //not being scheduled yet.
                    $(T, Rt, cquota)$ ←getTaskFair($Qt$);
                else
                    //Select the task $T$ with the
                    //highest priority and max $Rt$
                    $(T, Rt, cquota)$ ←getTaskRt($Qt$);
            $deposit \leftarrow deposit + Rt$;
            $pushQr(T, cquota)$;
            $quota \leftarrow quota - cquota$;
```

IV. I/O Oriented Scheduling

There are three core aspects of *I/O Oriented Scheduling*: Keeping I/O devices (SSDs) busy for maximum I/O bandwidth; Enforcing the task execution order by controlling the I/O order to reduce resource contention; Scheduling tasks with different I/O and kernel patterns together to improve the overall resource utilization.

The concrete scheduling is twofold: the scheduler first chooses a container to monopolize the full I/O bandwidth within its assigned quota as shown in Algorithm 1. The scheduler loops through all containers to make sure no container is starving. SPMario is able to support priority-based scheduling policy, by changing the I/O quota assigned to each container. Then the scheduled container executes the chunk-based task scheduling Algorithm 2 to determine the task execution order. This is essentially a budget-based scheduling, and for the convenience of the description, we assume the fixed-size chunks for now. To support various-sized chunks, we just need to modify the values of Rt based on the smallest chunk size accordingly.

V. Evaluation

We evaluate SPMario and show the experimental results in this section.

A. Experimental Platform

The experiments run on a server with a 4-core Intel Xeon E52609V2 CPU and 32GB DRAM. It contains an NVIDIA Tesla K20 GPU with 5GB GDDR5 memory connected with 16 lane PCIe 2.0 with 8GB/s bandwidth. The storage is a 768GB PICe SSD and can sustain 2.2GB/s for read and write. We run Ubuntu Linux kernel 3.16.3 with CUDA Toolkit 7.0 [1].

B. Benchmarks

We run the experiments with three representative GPU MapReduce applications. Matrix Multiplication (MM) has longer I/O time than the kernel, but the I/O and the kernel time are comparable. K-Means Clustering (KMC) is a compute-intensive application with longer kernel than I/O. Linear Regression (LR) has much shorter kernel than I/O. We use cuBLAS library [2] to implement MM for best performance, and extend KMC and LR from GPMR to support big datasets.

	MM16K	MM32K	KMC2K	KMC4K	LR2K	LR4K
Total Input Size (GB)	17.4	135.2	32	64	16	32
Chunk Size (MB)	64	128	128	128	128	128

TABLE I: **Dataset Sizes and Chunk Sizes of the Benchmarks.**

We test each application on two sizes of datasets, as shown in Table I, with 1, 2, and 4 processes on the baseline, while with 1, 2, and 3 on SPMario, because the SPMario scheduler occupies one CPU core. We also run four pairs of these applications to evaluate SPMario's scheduling policy.

C. Intra-Job I/O Oriented Scheduling

We evaluate SPMario's I/O Oriented Scheduling for individual jobs against optimized GPMR [16] as the baseline to be fair. Figure 2 and Figure 3 depict the results of GPU and I/O utilization, and Figure 4 shows average per-chunk I/O time for each set of the experiments.

In all the experiments, we observe similar results: For the single task case, SPMario can reduce the execution time by overlapping GPU kernels and I/O operations. With the increase in the task number, SPMario can continue to improve the GPU and I/O utilization, by reducing the I/O contention and the idle I/O time through deliberate scheduling. Figure 4 shows the average per-chunk I/O time of the baseline rises rapidly while remains almost unchanged in SPMario, indicating severe I/O contention among different tasks in the baseline, which is one of the main factors causing the inefficiency of the baseline.

Overall, with up to three processes, SPMario is able to achieve a 2.28× speedup over the original single process baseline in job execution time, and yields up to 2.12× GPU utilization and 2.51× I/O utilization.

D. Inter-Job I/O Oriented Scheduling

For scheduling multiple jobs, we run four pairs of jobs, including [MM16K, KMC2K], [MM16K, LR2K], [MM16K, KMC4K], and [KMC2K, LR4K]. The chosen jobs in each pair have close dataset sizes and execution time, and these combinations cover all the three different types of jobs.

Currently there is no job scheduler supporting multi-job scheduling on GPUs, and the GPMR cannot run multiple applications together either. So we use **SPMario with round-robin** as the baseline to compare against **SPMario with I/O Oriented Scheduling**.

It is not easy to further squeeze any improvement out of the highly optimized SPMario jobs, since SPMario has already removed most idle I/O time for individual jobs, as we see in section V-C. However, SPMario's **I/O Oriented Scheduling is still able to find optimization opportunities** to improve the utilization and performance, as summarized in Table II. Limited by NVIDIA K20's device memory (5GB), SPMario is only tested to schedule two jobs together, but it can schedule more with bigger device memory (e.g., 12GB in K40).

	Exec. Time ↘ (%)	GPU Util. ↗ (%)	I/O Util. ↗ (%)
[MM16K,KMC2K]	7.92	10.63	9.17
[MM16K,LR2K]	12.59	1.80	11.65
[MM16K,KMC4K]	6.53	7.18	7.03
[KMC2K,LR4K]	13.54	12.27	14.92

TABLE II: **The Percentage Improvement of SPMario with I/O Oriented Scheduling over Round-robin for Co-Scheduling.** When scheduling two jobs together, SPMario with I/O Oriented Scheduling can reduce the total execution time by up to 13.54 %, and improve GPU and I/O utilization by up to 12.27 % and 14.92 %, respectively

VI. Related Work

Mars [11] is the first attempt to port the MapReduce model to GPUs. Catanzaro et al. [4] build a framework with a constrained MapReduce model. MapCG [12] avoids some inefficiency of Mars with atomic operations and provides source code portability between CPU and GPU. Some efforts utilize the shared memory in a GPU [13], [5]. Grex [3] takes a step further by using parallel input data split and load-balance data partitioning. Chen et al. [6] explore different schemes to run tasks on an integrated architecture. These early studies mainly focus on offloading computation to the GPU, and can only handle datasets smaller than GPU memory. On the contrary, SPMario can efficiently handle large datasets.

GPMR [16] is a GPU MapReduce library for both single node and GPU cluster. It utilizes chunking to handle datasets bigger than GPU memory, but does not consider I/O accesses. PMGMR [7] improves its single-node predecessor MGMR [8] for bigger datasets, but using slow hard drive disks will underutilize the multiple GPUs. Moim [17] implements a multi-GPU MapReduce framework for both CPUs and GPUs and supports load balancing. MATE-CG [14] provides generalized reduce API to run jobs on both CPUs and GPUs. Glasswing [10] uses a 5-stage pipeline to improve the performance of a hybrid MapReduce system with OpenCL. But its ”vertical scalability” merely means getting better performance with a better device (e.g., ”scale” from a CPU to a GPU), while still running a single task on each device. In SPMario we explore to scale up GPU MapReduce with running multiple concurrent tasks and even jobs on a single GPU.

None of the above GPU MapReduce frameworks or systems

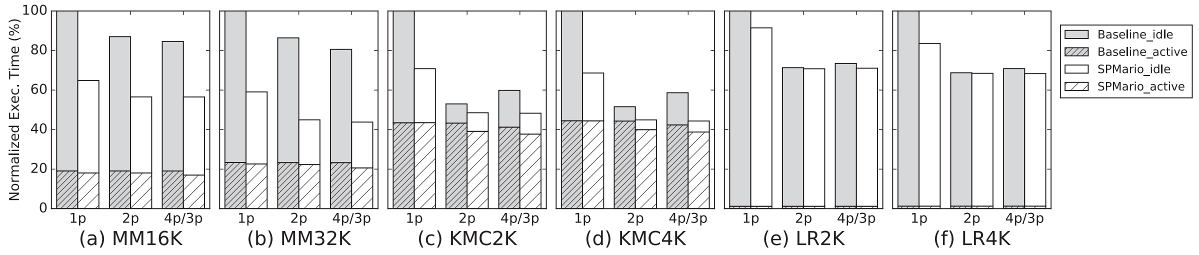

Fig. 2: **GPU Utilization Comparison of the Benchmarks with Different Numbers of Processes.** All data are normalized to the execution time of the corresponding baseline's single process case (1p) for comparison. Note we use 4p/3p to denote the comparison of 4p for baseline and 3p for SPMario. Both the baseline and SPMario decrease the execution time and improve the GPU utilization by shrinking the idle time with more processes (baseline from 1p to 4p, SPMario from 1p to 3p), but SPMario outperforms the baseline for all cases.

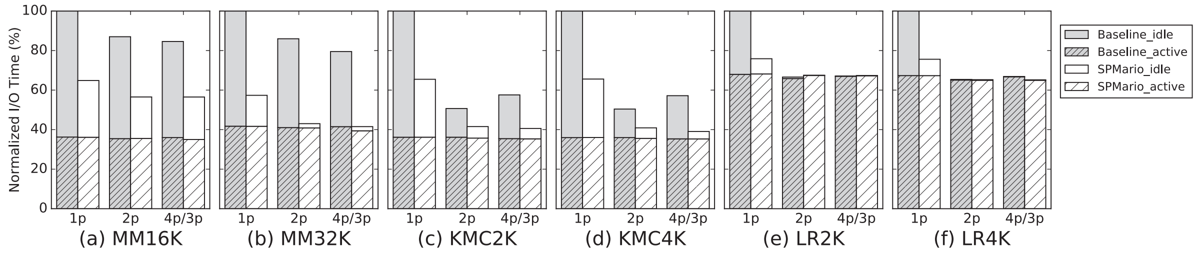

Fig. 3: **I/O Utilization Comparison of the Benchmarks with Different Numbers of Processes.** All data are normalized to the I/O span of the corresponding baseline's single process case (1p) for comparison. Note we use 4p/3p to denote the comparison of 4p for baseline and 3p for SPMario. Both the baseline and SPMario improve the I/O utilization by shrinking the idle time with more processes (baseline from 1p to 4p, SPMario from 1p to 3p), but SPMario outperforms the baseline for all cases except LR2K.

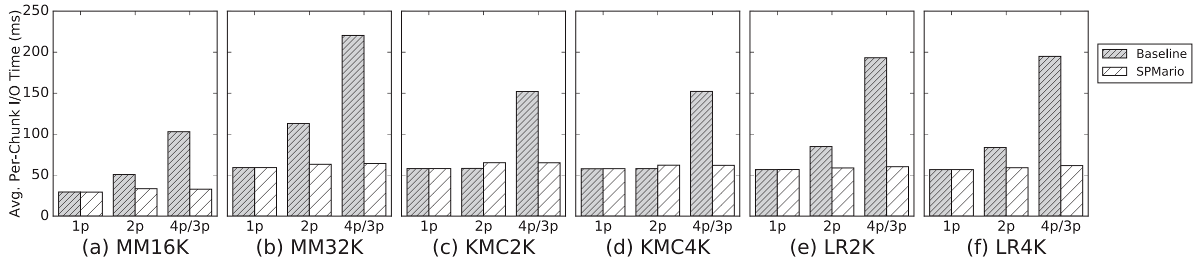

Fig. 4: **Avg. Per-Chunk I/O Time Comparison of the Benchmarks with Different Numbers of Processes.** Note we use 4p/3p to denote the comparison of 4p for baseline and 3p for SPMario. With more processes (baseline from 1p to 4p, SPMario from 1p to 3p), the average per-chunk I/O time drastically increases for the baseline, while remains almost unchanged for SPMario.

explore the potential of sharing a single GPU among multiple jobs, or improving the system utilization. On the contrary, SPMario employs I/O Oriented Scheduling to efficiently overlap I/O, data transfer between the host and the GPU, and GPU kernel execution, and to schedule multiple jobs to run together to better utilize the I/O and GPU resources.

VII. Conclusion

GPU MapReduce frameworks require efficient I/O handling and task scheduling for both performance and resource utilization. In this paper, We present SPMario, a GPU MapReduce framework to improve the utilization of both GPU and I/O resources, and share the GPU among concurrent tasks. Experiments on three representative applications show SPMario is able to accelerate job execution and boost resource utilization by removing idle I/O time with I/O Oriented Scheduling.

References

[1] https://developer.nvidia.com/cuda-toolkit.
[2] http://docs.nvidia.com/cuda/cublas/#axzz3yw7ERpUo.
[3] C. Basaran and K.-D. Kang. Grex: An Efficient MapReduce Framework for Graphics Processing Units. *J. Parallel Distrib. Comput.*, 73(4), Apr. 2013.
[4] B. Catanzaro, N. Sundaram, and K. Keutzer. A Map Reduce Framework for Programming Graphics Processors. In *Workshop on Software Tools for MultiCore Systems*, 2008.
[5] L. Chen and G. Agrawal. Optimizing MapReduce for GPUs with Effective Shared Memory Usage. In *HPDC '12*, 2012.
[6] L. Chen, X. Huo, and G. Agrawal. Accelerating MapReduce on a Coupled CPU-GPU Architecture. In *SC '12*, 2012.
[7] Y. Chen, Z. Qiao, S. Davis, H. Jiang, and K.-C. Li. Pipelined multi-GPU MapReduce for big-data processing. In *Computer and Information Science*, pages 231–246. Springer, 2013.
[8] Y. Chen, Z. Qiao, H. Jiang, K.-C. Li, and W. W. Ro. Mgmr: Multi-gpu based mapreduce. In *Grid and Pervasive Computing*, pages 433–442. Springer, 2013.
[9] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation*, OSDI'04, 2004.
[10] I. El-Helw, R. Hofman, and H. E. Bal. Scaling MapReduce Vertically and Horizontally. In *SC '14*, 2014.
[11] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: A MapReduce Framework on Graphics Processors. In *PACT '08*, 2008.
[12] C. Hong, D. Chen, W. Chen, W. Zheng, and H. Lin. MapCG: Writing Parallel Program Portable Between CPU and GPU. In *PACT '10*, 2010.
[13] F. Ji and X. Ma. Using Shared Memory to Accelerate MapReduce on Graphics Processing Units. In *IPDPS '11*, 2011.
[14] W. Jiang and G. Agrawal. MATE-CG: A MapReduce-Like Framework for Accelerating Data-Intensive Computations on Heterogeneous Clusters. In *IPDPS '12*, 2012.
[15] S. Ma, X.-H. Sun, and I. Raicu. I/O throttling and coordination for MapReduce. *Illinois Institute of Technology*, 2012.
[16] J. A. Stuart and J. D. Owens. Multi-GPU MapReduce on GPU Clusters. In *IPDPS '11*, 2011.
[17] M. Xie, K.-D. Kang, and C. Basaran. Moim: A Multi-GPU MapReduce Framework. In *CSE '13*, 2013.