2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing

Collocating CPU-only jobs with GPU-assisted jobs on GPU-assisted HPC

Jiadong Wu and Bo Hong
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, USA
jwu65, bohong@gatech.edu

***Abstract*—In recent years, GPU has evolved rapidly and exhibited great potential in accelerating scientific applications. Massive GPU-assisted HPC systems have been deployed. However, as a heterogeneous system, GPU-assisted HPC is harder to be programmed and utilized than conventional CPU-only system. Statistics of the Keeneland system indicate that the effective utilization rate of computational resources is only about 40% when the system runs in normal condition with enough jobs in its queue. Our theoretical model shows that the lack of overlap between CPU/GPU computation is a major obstacle in the efficient utilization of heterogeneous system. In this paper, we evaluate the possibility of collocating CPU-only job with GPU-assisted job on the same node to increase overlap between CPU/GPU computation, thus achieving better utilization. Several performance compromising factors, such as resource isolation, CPU load, and GPU memory demands, are studied based on workload from popular MPI/CUDA benchmarks. The results indicate that, when those factors are managed properly, the collocated CPU-only job can efficiently scavenge the underutilized CPU resource without affecting the performance of both collocated jobs. Based on this insight, an experimental system with collocation-aware job scheduler and resource manager is proposed. With our experiment workload pool of mixed CPU and GPU jobs, the system demonstrates 15% gain in throughput and 10% gain in both CPU and GPU utilization.**

I. Introduction

GPU has already proved its exceptional capability in accelerating HPC applications. In the Top500 list of supercomputers, the percentage of GPU-quipped system is growing considerably and this trend is expected to continue fin the foreseeable future.

One major concern of the GPU-assisted system lies with the utilization of computational resources. The utilization of CPU is usually not a problem for conventional HPC system, in which CPU is the only processing unit. In such systems CPU is busy for most the allocated time, despite the occasional waiting for I/O or network. However, in GPU-assisted systems the utilization problem arises if CPU and GPU can not be utilized simultaneously by the jobs. To make full use of such heterogeneous system, the fundamental idea is making the CPU/GPU workload ratio of the jobs match with the CPU/GPU device ratio of the system. Since the device ratio is typically fixed after system deployment, a match can only be achieved either by manipulating the workload ratio in every single job or by collocating jobs with complimentary workload ratio.

There are various parallel programming models available (e.g. MPI, OpenMP, CUDA, OpenCL, PGAS, OpenACC) for harnessing the immense power of GPU-assisted HPC, whereas the combined model of MPI and CUDA/OpenCL are most widely supported. However, to make the heterogeneous resources effectively shared and utilized within a MPI+CUDA/OpenCL based application is not easy.

Firstly, MPI applications are usually designed with implicit assumption that each software process runs with a dedicated resource. However, since there are usually more CPU cores then GPU devices equipped on each computing node, it is common in existing GPU-assisted HPC systems that one GPU device may be shared by several CPU processes. Secondly, the CUDA/OpenCL model gives GPU the abstraction of an auxiliary processor to the CPU. Despite the fact that such hierarchical abstraction reflects the control logic in GPU-assisted systems, it is not so programmer-friendly if they want to make CPU and GPU work in parallel with specific ratio. Although asynchronous GPU functions are provided in the model, designing an algorithm that properly overlaps CPU and GPU computation can be tricky or even impossible for some applications.

In this paper we evaluate the feasibility of collocating GPU-assisted jobs and CPU-only jobs, i.e. running a GPU-assisted job and another CPU-only job simultaneously on one node to better utilize both computational resources. Although CPU resources will be oversubscribed when collocating jobs, our experiment results indicate that the execution time of most participating jobs won't be prolonged by more than 5% if the following factors are properly managed: 1) CPU resources isolation, 2) combined CPU load and 3) GPU memory copy bandwidth.

Based on the evaluation results, an experimental multi-node system with collocation-aware job scheduler and resource manager is then proposed. Our system shows that job collocation can significantly improve the throughput and utilization when executing mixed CPU-only jobs and GPU-assisted jobs.

The rest of the paper is organized as follows. In Section II, we provide the background information on the utilization problem and survey the related works. In Section III, we

978-0-7695-4996-5/13 $26.00 © 2013 IEEE
DOI 10.1109/CCGrid.2013.19

Node State	Sample		Utilization		
	Volume	Ratio	CPU	GPU Cyc.	GPU Mem.
$ppn = 1$	96871	5.7%	11.5%	15.6%	4.8%
$ppn = 3$	485582	28.7%	25.2%	34.9%	13.1%
$ppn = 6$	126683	7.5%	43.8%	58.7%	1.7%
$ppn = 12$	414254	24.5%	83.3%	17.9%	4.1%
Idle	417708	24.7%	0.0%	0.0%	0.0%
Offline	128770	7.6%	n/a	n/a	n/a

Table I
UTILIZATION STATISTICS ON KEENELAND

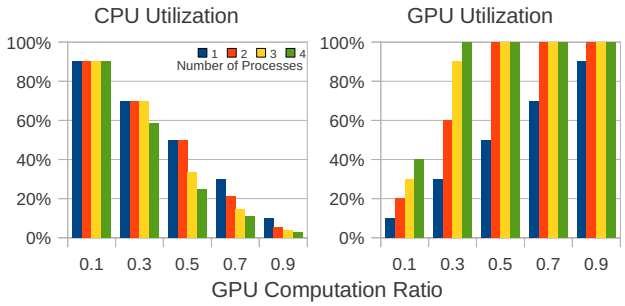

Figure 1. The relationship between between system utilization and GPU computation ratio

evaluate the performance of collocating jobs with experiments on a large variety of job combinations. In Section IV, we use a multi-node experimental system to demonstrate how the system utilization and throughput can be improved by collocating jobs. In Section V the conclusions and discussions are drawn.

II. BACKGROUND AND RELATED WORKS

A. Utilization Statistics on Keeneland

Keeneland is a GPU-assisted HPC developed by Georgia Tech and ORNL. Its initial delivery system (KIDS) consists of 120 HP SL-390 computing nodes with 2 Intel Westmere hex-core CPUs and 3 NVIDIA C2090 Fermi GPUs equipped on each node.

We monitored Keeneland's PBS system for a period of ten days, and gathered a total of 1689360 samples, in which each sample represents the state of one computation node in one minute. According to Keeneland's node allocation policy, there are 3 major states: offline, idle, and exclusively executing certain job. The four most popular ppn (processes per node) settings on Keeneland are $ppn = 1, 3, 6, 12$, since the CPU to GPU ratio is $3 : 12$. The utilization statistics are listed in Table.I. In average, the system-wide CPU utilization is only 42.0%. Moreover, since by default CUDA/OpenCL setting the processes will actively spin on the CPU while waiting for results from GPU, the effective CPU utilization should be even lower than the percentage value we monitored. Although the exclusive node allocation policy can provide better resource isolation between jobs, it intensifies the under-utilization issue of GPU-assisted HPC system. The statistics indicate that the feasibility of collocating multiple jobs should be evaluated.

B. Modeling the Utilization

Parallelism in accessing different resources is the most important factor that determines the utilization of systems with heterogeneous resources. The necessity of overlapping CPU/GPU computation can be demonstrated with the following generic model.

The abstraction of existing GPU-assisted HPC system we considered is that a group of identical computing nodes are exclusively allocated to a GPU-assisted job. For such a job, assuming that 1) every process gets a dedicated CPU core, 2) every p processes get a shared GPU device, 3) the processes do not synchronize with their peers, 4) the processes have similar workload in which the ratio of GPU kernel computation time to the entire execution timespan is r_g, and 5) in any process CPU computation and GPU computation do not overlap. The utilization of CPU U_c and GPU U_g can thus be derived as:

$$U_c = \begin{cases} 1 - r_g & \text{if } r_g < 1/p \\ \frac{1-r_g}{p \times r_g} & \text{if } r_g \geq 1/p \end{cases}$$

$$U_g = \begin{cases} p \times r_g & \text{if } r_g < 1/p \\ 1 & \text{if } r_g \geq 1/p \end{cases}$$

The relation between resource utilization and the GPU computation ratio r_g is illustrated in Fig.1. It can be clearly observed that the utilization of the two resources are actually conflicting to each other, indicating that the overall utilization of the system can not be improved without overlapping GPU computation with CPU computation. The figure also shows that GPUs saturate faster if the user application launches enough processes per node. As a result, for this research we are focusing on collocating a CPU-only job with the GPU-assistied job to scavenge the underutilized CPU resource.

C. Related works on overlapping computation

There are two levels of CPU/GPU computation overlap: overlap within one job and overlap among multiple jobs. Overlap within one job can be manually implemented in the application codes or be transparently enabled by high-level programming models. Popular techniques for optimizing utilization of individual application include workload assignment and pipelining. To design a scheme that efficiently pipelines the workload on to CPU and GPU can be hard and usually platform dependent. High-level programming models, such as GPUSs[1] and MapCG[2], are thus developed to make such tuning work transparent to the users. These models give unified abstraction to CPU and GPU, and their compilers are capable of generating separate executables for the two resources from same piece of user code. The

workload will be dynamically divided and fed into both CPU and GPU if program dependency is satisfied. However, these models have strong assumption on the behavior of the application, which makes them less versatile than the basic MPI+CUDA/OpenCL model.

Overlapping CPU/GPU computation among multiple jobs is implicitly supported by the OS. However, due to the possible contention created by collocated jobs, such level of overlap should be facilitated by specially designed software infrastructures to avoid performance degradation. Middlewares[3], [4] such as rCUDA enable the execution of kernel on remote GPU devices and are thus capable of improving the system-wide utilization by shifting workload from busy GPUs to idle GPUs. Scheduling mechanism[5] has been developed for MapCG based system, in which CPU and GPU are dedicated to separated processes. On the other hand, previous research[6] also showed that throughput of conventional HPC systems could be improved by collocating jobs to oversubscribe CPU resources. However, to the best of our knowledge, our work is the first systematic study of collocating CPU-only jobs with GPU-assisted jobs.

III. Performance Studies of Collocating Jobs

Intuitively, real HPC jobs tend to perform worse when collocated with another job competing the resources. Whether job collocation can improve the throughput of the entire system depends on the trade off between the CPU cycles scavenged and the overhead incurred. In this section, we evaluate and model how the jobs are slowed down in collocated environment than in dedicated environment. The experiments focus on those key factors that compromises the performance of collocated jobs, which are resource isolation, combined CPU load, and cudaMemcpy bandwidth.

A. Experimental Setup

1) System Platform: The experimental platform we used is a 4-node cluster with two Intel Xeon X5675 hex-core CPUs, three Nvidia M2070 GPUs, and 24GB of memory installed on each node. All the nodes are connected with a Gigabit Ethernet switch. To minimize the impact of CPU's specific micro architecture, the Intel SMT and Turbo Boost features are turned off. Our software environment consists of Linux kernel 2.6.32-279.el6.x86_64, GCC 4.4.6, Openmpi 1.6, CUDA 4.2.9, Torque 2.5.12, and Maui 3.3.1.

2) Workload Characteristics: Our study on collocating performance involves jobs of two different categories of workload: CPU-only and GPU-assisted. A group of 15 MPI benchmarks are selected from the NAS Parallel Benchmark (NPB)[7] 3.3.1 as our CPU-only workload, and a group of 32 MPI+CUDA benchmarks are selected from the Scalable HeterOgeneous Computing (SHOC) Benchmark[8] 1.1.4 as our GPU-assisted workload. By joining the two groups, a total of 480 different job combinations are constructed. For conciseness, the two categories of workload are referred to as CPU workload and GPU workload in the rest of the paper.

Name	Class Opt.	NProcs Opt.	Execution Time (s)
CG	B	2	48.07
		4	27.06
		8	14.67
EP	B	2	46.56
		4	23.11
		8	11.51
FT	B	2	49.77
		4	26.656
		8	15.48
IS	C	2	8.16
		4	4.31
		8	2.72
MG	C	2	84.50
		4	44.48
		8	22.69

Table II
THE CHARACTERISTICS OF CPU WORKLOAD

The MPI benchmarks we used can be broken down into five basic types respectively: 1) Conjugate Gradient (CG), irregular memory access and communication; 2) Embarrassingly Parallel (EP), minimal communication; 3) Discrete 3D fast Fourier Transform (FT), all-to-all communication; 4) Multi-Grid (MG) on a sequence of meshes, memory intensive; and 5) Integer Sort (IS), random memory access. All these NPB benchmarks are configured with different $Class$ options (which identify problem size) and $NProcs$ options (which identify number of processes). Their baseline execution time on our experimental platform without collocated workload ranges from 2.72 to 84.5 seconds, as listed in Table.II.

Our 32 MPI-CUDA benchmarks can be broken down into eight types: 1) 1D Fast Fourier Transform (FFT), 2) N-body pairwise computation from molecular dynamics (MD), 3) Reduction, 4) Exclusive parallel prefix sum (Scan), 5) General matrix multiplication (SGEMM), 6) Radix sort on integer key-value pair (Sort), 7) Sparse matrix vector multiplication (Spmv), and 8) Streamed Triad. All these benchmarks are configured as 12 processes and single precision (if not otherwise stated). The benchmarks are also configured as embarrassingly parallel on MPI level and as true parallel on CUDA level, such that the MPI processes are executing independently over the same copy of code. Three baseline characteristics (execution time, CPU load, cudaMemcpy bandwidth) are profiled on our platform. The CPU load represents the actual CPU utilization of the benchmark with its 12 processes running concurrently. If the CPUs are fully utilized, the value of CPU load should be greater than or equal to 12. The cudaMemcpy bandwidth is the per process average bandwidth of bi-directional cudaMemcpy traffic, which represents the GPU

Name	Size Opt.	Iteration Opt.	Execution Time (s)	CPU Load	cudaMemcpy bandwidth (MB/s)
FFT	4	1	6.50	2.22	39.43
		120	18.47	1.19	13.86
		240	30.60	0.72	8.36
		360	42.64	0.54	6.00
MD	1	2000	7.91	9.87	0.81
		4000	10.92	7.69	0.58
		10000	20.77	4.83	0.31
		30000	55.68	2.82	0.11
Reduction	3	1	0.90	1.17	35.41
		10	4.51	0.70	70.98
		20	8.86	0.61	72.26
		30	13.15	0.59	73.03
Scan	2	1	0.44	1.04	36.72
		4	1.42	0.79	28.10
		8	2.79	0.65	25.80
		16	5.52	0.57	24.62
SGEMM	3	1	1.20	2.98	53.19
		4	4.37	1.26	36.63
		8	8.58	0.75	33.56
		20	21.21	0.35	31.68
Sort	4	1	2.74	1.43	140.00
		8	8.81	1.57	348.81
		16	16.29	1.54	377.08
		24	23.77	1.54	387.77
Spmv	4	1	6.69	5.99	13.50
		4	18.37	2.60	4.96
		8	33.19	1.50	2.78
		16	62.10	0.86	1.53
Triad	4	1	0.75	5.05	576.64
		10	7.07	4.94	611.14
		20	14.30	4.92	604.20
		30	21.65	4.99	598.48

Table III
THE CHARACTERISTICS OF GPU WORKLOAD

memory demands of the benchmark. The profiling results, listed in Table.III, show that these benchmarks cover a wide spectrum of GPU-assisted workload patterns. The detailed $Size$ and $Iterations$ options that we used to configure each benchmark can also be found in the table.

In order to facilitates our experiments, following modifications are applied to the SHOC benchmark code : 1) Assigning the ith GPU device to the host processes whose rank mod GPU count equals i; 2) Explicitly setting CUDA device flag and event flag as BlockingSync to make sure that host CPU process does not busy spin when waiting for GPU; 3) Wrapping the RunBenchmark() function into for-loops to enable repeat running of benchmark without reinitialize GPU devices.

3) Performance Metric: In the experiments, we focused on the performance metric of Slowdown Ratio, which is defined as:

$$\frac{(CollocatedExecutionTime - BaselineExecutionTime)}{BaselineExecutionTime}$$

. If both collocated CPU and GPU jobs have small slowdown ratio (e.g. less than 5%), it means that this job combination is able to effectively utilize the system resources and leverage

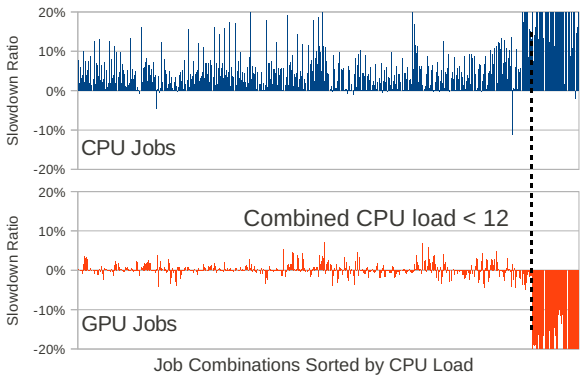

Figure 2. The slowdown of shared mode over partitioned mode

the system throughput. The slowdown ratio can be affected by many conditions. In the following sections, different performance compromising factors are evaluated against this ratio.

B. Performance Compromising Factors

1) Resource Isolation: When a CPU job and a GPU job are collocated on the same node, the CPU cores on that node can be managed in two ways: shared or partitioned. For the shared mode, both jobs can access any of the cores, and the contentions are handled by Linux's default process scheduling algorithm. For the partitioned mode, either job will have its dedicated subset of cores, which are enforced explicitly with CPU affinity settings. Since in typical CPU jobs every process is able to fully utilize the CPU resource, we will dedicate equal amount of cores to these CPU processes and the remaining cores to GPU job.

We separately tested the 480 job combinations with these two modes and calculated the difference in execution time. The results are listed in Fig.2. Each blue bar in the figure represents the CPU job slowdown ratio of shared mode over partitioned mode in one specific combination, and each red bar represents the GPU job slowdown ratio in the same combination. To make the figure clearer, the bars are sorted respectively by job combination's CPU load. The figure shows that the CPU job of most job combinations takes longer to finish in shared mode. For the GPU jobs, if the combined CPU load is less than 12, both modes have some losses and gains. In general, collocated jobs perform better if the resources are partitioned. The following experiments are also focused on this mode.

2) Combined CPU Load: Fig.3 lists the CPU job slowdowns and GPU job slowdowns of the 480 job combinations over their baseline execution time. Intuitively, both jobs take longer to finish as the combined CPU load increases. However, the performance of GPU jobs is less vulnerable than which of CPU jobs when collocated. There two reasons for the such difference: 1) the host processes of GPU jobs

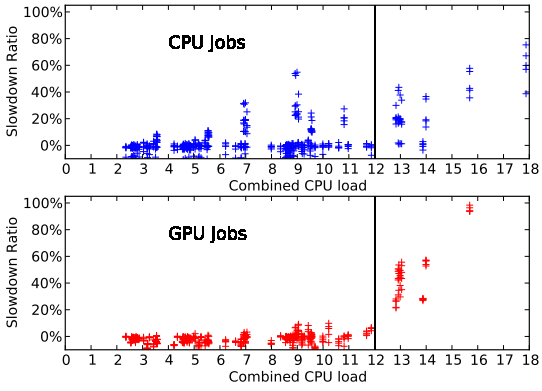

Figure 3. The slowdown of jobs against combined CPU load

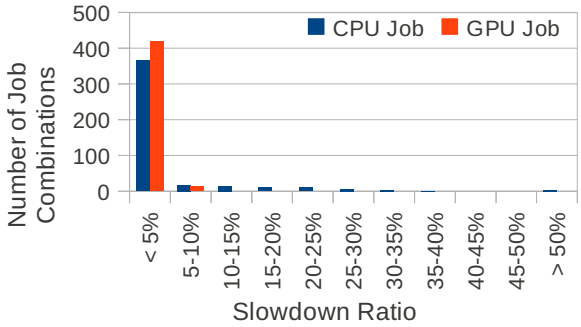

Figure 4. The distribution in of jobs over slowdown ratio

in most SHOC benchmarks are embarrassingly parallel, and 2) the performance of GPU kernels is independent of the performance of its host process. It can also be noticed in the figure that the slowdown in both jobs are much less significant if the combined load is less than 12 (the number of CPU cores per node). When the CPUs become over-utilized, the performance of both collocated jobs decreases dramatically.

The distribution of performance with combined load ranging from 0 to 12 is listed in Fig.4. It shows that the slowdown of all GPU jobs and of 88% of the CPU jobs are less than 10%. According to this result, if the CPUs are not persistently over utilized, collocating CPU job can be served as an efficient way to scavenge the CPU resources.

3) cudaMemcpy Bandwidth: As shown in Fig.3, CPU jobs in the majority of the combinations have small slowdown ratio if combined load is less than 12. However, there are some incidences when CPU jobs are slowed down even if CPU resources are abundant. In the Fig.5, the same groups of results are plotted against CPU load and cudaMemcpy bandwidth respectively, and those results related to the same type of CPU benchmark are marked with unique color. This figure demonstrates that the slowdown ratio has clear correlation with both the GPU job's cudaMemcpy bandwidth and the CPU job's type. In general, the cudaMemcpy bandwidth exhibits a strong exponential impact over the performance, while different types of CPU jobs have different sensitivity to this impact.

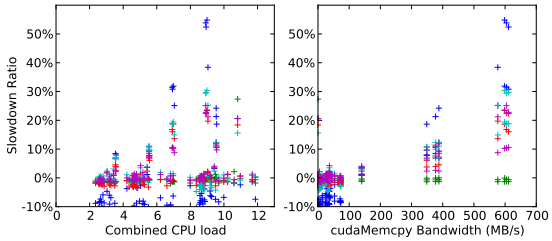

Figure 5. The slowdown of CPU jobs against combined CPU load and cudaMemcpy bandwidth

Job Type	CG	EP	FT	IS	MG
Coeff. a	0.00761	0.0001	0.00563	0.00420	0.00457
Coeff. b	0.00151	0.007	0.00230	0.00641	0.00330

Table IV
REGRESSION ANALYSIS RESULTS

We developed the following regression model to link the slowdown ratio with multiple parameters of the impact.

$$S = P \times b \times e^{a \times B}.$$

In this model, S is the slowdown ratio, P is the process number of the CPU job, B is the cudaMemcpy bandwidth of the GPU job, a and b are the two coefficients. The regression procedures are illustrated in Fig.6, in which the experiment results related to CG type CPU benchmark are used as samples. The three colors used in this figure represent three process number (2, 4, and 8) of CPU job. The first step is to remove all the results with a small bandwidth (i.e. less than 100 MB/s) since the bandwidth is a minor performance compromising factor in those cases. Then the slowdown ratios are divided by the process number. The divided values are used as samples to fit an exponential curve. Finally, multiply the process number back into coefficients to get the final curves. With these procedures, we can get a pair of coefficient for each type of CPU jobs, which are listed in Table.IV. The quality of this model is illustrated in Fig.7. The coefficients indicate that synchronization intensive CPU jobs such as CG are more sensitive to the bandwidth impact.

Although the coefficients varies across different types of CPU workload, a profiling-based threshold over cudaMemcpy bandwidth can be setup in production system to conservatively prevent cases of serious slowdown. In our experiment for example, slowdown can be capped within 10% by preventing GPU jobs with bandwidth above 300MB/s from collocating with any CPU job. In the following research, we will investigate the possibility of implementing a smarter

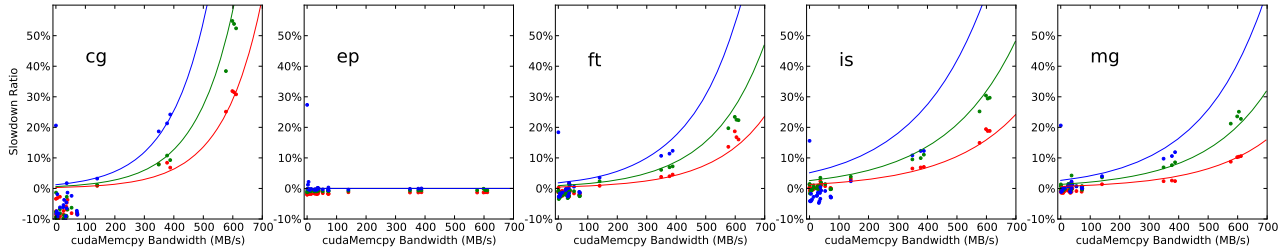

Figure 7. Regression quality

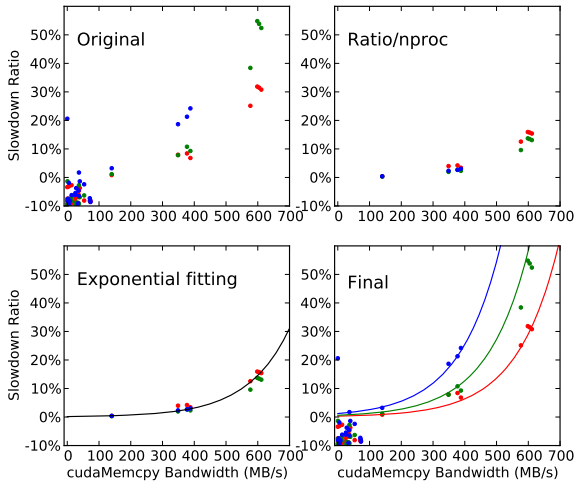

Figure 6. Regression analysis of slowdown over cudaMemcpy bandwidth

Job Type	CG	EP	FT	IS	MG
CPU Job Slowdown	2.16%	15.1%	3.88%	4.89%	11.3%

Table V
AVERAGE SLOWDOWN OF COLLOCATING MULTI-NODE CPU JOBS WITH TRIAD GPU JOBS

threshold with estimation of the job-specific coefficients.

4) Multi-node Jobs: When job collocation involves more than one node, the performance will be affected by other conditions such as network bandwidth. In this section, we scaled up the experiments to 4 nodes. The CPU jobs are configured with 32 processes ($ppn = 8$), and the GPU jobs are configured with 48 processes ($ppn = 12$).

Table.V lists the CPU job slowdown of the extreme condition —- collocating with Triad GPU jobs which incur the largest cudaMemcpy bandwidth and also a combined CPU load higher than 12. Even under such intensive pressure, the multi-node CPU jobs exhibit only a slight degradation in performance. For those communication intensive CPU jobs (CG FT and IS), the impact of job collocation is mostly shadowed by the impact of network due to the fact that the network bandwidth is significantly lower than the memory bandwidth. For those jobs with less communication (EP and MG), the slowdown ratios are relatively higher, since in those jobs the over-utilized CPU becomes the major bottleneck. For the GPU jobs, the performance is identical to their single-node counterparts, because they are embarrassingly parallel on the process level.

C. Insights on the performance of job collocation

Our experimental results indicate that the performance of job collocation can be determined by four major factors: 1) resource isolation, 2) CPU load, 3) GPU memory demands, and 4) inter-process synchronization/communication intensity. Among these factors, resource isolation can be easily managed by the system, CPU load and cudaMemcpy bandwidth can either be monitored by the system or be hinted by the user, and synchronization/communication intensity can only be hinted by the user. With these major factors informed, CPU/GPU jobs can be efficiently collocated to better utilize the heterogeneous resources.

Several other job characteristics (memory footprint, data precision, GPU kernel launch frequency) and system characteristics (I/O rate, hardware/software IRQ frequency) have also been evaluated, although we didn't observe significant correlation between these characteristics and the slowdown in collocated jobs. However, due the incompleteness of our experiment, it is possible that additional factors exist. Another issue involving the study is that the workload didn't cover the possible co-scheduling slowdown when the processes in one job have dependencies among each other. We plan to address these issues in future research by extending the coverage of the workload pool.

IV. COLLOCATING JOBS ON EXPERIMENTAL SYSTEM

In this section, an experimental system with collocation-aware job scheduler/resource manager is proposed to demonstrate the capability of job collocation in improving system utilization and throughput.

A. Job Scheduling and Resource Management

In the proposed system, the jobs are scheduled based on their priority. GPU jobs are assigned higher priority than CPU jobs, since the system can be relatively better utilized

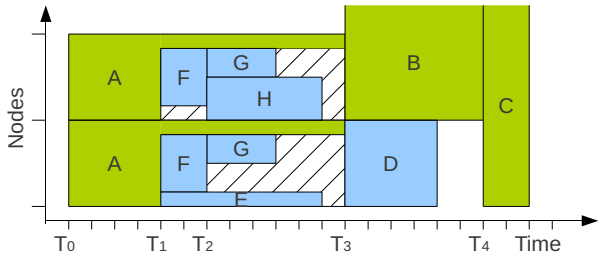

Figure 8. Scheduling example of mixed jobs

by GPU jobs. In either type of jobs, the ones with higher ppn value have higher priority. Backfilling is also enabled, so that some of the low priority jobs (most likely the small CPU jobs) may be started earlier if they are not delaying high priority jobs. The objective of this scheduling policy is to optimize the overall system utilization.

The CPU resources are managed in an elastic way. The CPU jobs are given 100% dedicated CPU resources during their requested execution time, but the GPU jobs are given 100% dedicated CPU resources only during the monitoring period of their requested execution time. After that, the underutilized CPU resources may be collected if there are feasible CPU jobs. Specifically, when a GPU job gets executed, the system begins monitoring the CPU load of its allocated nodes at 1-minute resolution. The monitoring period ends when the job reaches portion P_M of its requested execution time, and then an estimation of the underutilized CPU resources is made based on the CPU load assuming that the job's future behavior will be consistent with the behavior we monitored. The underutilized CPU will be marked as available until the GPU job finished, allowing feasible CPU jobs to be filled into these temporal resources.

Fig.8 gives a scheduling example. Job A-C are GPU jobs, and job D-H are CPU jobs. The portion of monitoring period is set as $P_M = 1/3$. At the beginning, job A-D were scheduled to time T_0 T_3 and T_4 based on their priorities. At time T_1, the system finished monitoring job A and also marked the underutilized CPU resources available on the related nodes. Feasible small CPU jobs, E-H, were then scheduled to time T_1 and T_2 on the temporary resources.

Since the estimation of temporary underutilized resources may be inaccurate due to irregular behavior of some GPU jobs, a upper bond P_B can be set on the maximum collectible CPU resources. For example, with $P_B = 8$, at most 8 CPU processes are allowed to be scheduled on the temporary resources, even if the monitored CPU load is only 1.5 for the existing GPU job. Such limit can create a buffer zone for the GPU job to accommodate unpredicted contention.

The combination of P_M and P_B determines the resource collection policy. A moderate policy of longer monitoring period and smaller collectible CPU resources limit can lower the chance of jobs being killed due to serious delay.

B. Software Modification

In order to support the new feature of job collocation, modifications are needed in Torque[9] and Maui[10]. Based on the idea that job collocation is more of a scheduler feature than a resource manager feature, we implemented the most of the new functionalities in Maui, while only keeping resource monitoring and binding duties in Torque.

1) Torque: Torque is a very popular open-source HPC resource manager based on the original PBS project. Torque has three components: a) pbs_server, a management deamon running on central node; b) pbs_sched, a basic scheduler deamon running on central node; and c) pbs_mom, a monitor deamon running on each computation node. Instead of relying on its basic scheduler, Torque is usually deployed with another standalone scheduler such as Maui to achieve better performance.

The major modification we made on Torque 2.5.12 is recording PID of MPI processes and manipulating CPU binding. Although binding is natively supported in Torque code, the affinity can not be altered dynamically. Therefore, we modified the its scheduler API to record and report the MPI processes' PID of each job. Maui will make the binding decision and inform Torque's pbs_mom to manipulate the binding dynamically. To better monitor the CPU load, we also inserted a time stamp on each CPU load value captured.

2) Maui: Maui is a widely used open-source batch job scheduler, and it supports a variety of scheduling features. However, it has not been optimized for GPU-assisted HPC system. We have implemented three new features on top of Maui 3.3.1: a) GPU-aware node allocation, b) Elastic CPU resource management, c) Collocation-aware dynamic CPU set binding.

With these new features, the scheduler can prioritize qsub request with specific GPU requirement. It will monitor and collect underutilized CPU resource on those nodes occupying by GPU jobs. When collocated jobs are detected, the scheduler is also capable of binding them to partitioned CPU resources.

C. Experimental Result

In this experiment, the system is presented with a pool of mixed jobs, in which the 32 GPU jobs and 300 CPU jobs are implemented with the workloads that are studied in previous sections. The ppn is configured as 2-8 for the CPU jobs and 12 for the GPU jobs. The execution time is configured as 0.5-11 minutes for CPU jobs and 7-13 minutes for the GPU jobs. When submitting each job, the time request is set to 115% of its expected execution time. The system is configured with a moderate resource collection policy: the upper bond of collectible CPU resources P_B is set to 9 out of 12 cores, and the portion of monitor period $P_M$is set to $1/3$.

When the jobs are handled by conventional Torque/Maui system, it takes a makespan of 9345 seconds to finish. In

the entire 448,560 CPU core seconds, 195,162 are spent on CPU jobs, 237,936 are spent on GPU jobs, and 15,462 are idle due to scheduling. The weighted average CPU load for GPU jobs is 2.5, which means about 188,000 core seconds are underutilized in GPU jobs. The effective CPU utilization during entire makespan is 54.5%.

When handled by our collocation-enabled Torque/Maui system, the makespan is 7,857 seconds, reduced by 15.9%. The CPU time spent on either type of jobs remains roughly unchanged since the scheduler is aware of the potential slowdown when collocating jobs. 8,118 idle core seconds are created by scheduler. The results indicates that the collocated CPU jobs have scavenged more than 60,000 or 32% underutilized core seconds from GPU jobs. The effective CPU utilization is improved to 64.0%, and the GPU utilization is also improved form 53.0% to 63.1%. In average, only 2.3 out of 332 job are killed due to timeout, and they are short CPU jobs.

In the 188,000 core seconds of underutilized CPU resources, 33.3% resides in the monitoring period, 10.3% is due to the bond of collectible resources, and the rest 56.4% is the real collectible resources. If the moderate collection policy is strengthened to $P_M = 1/5P_B = 2$, the real collectible resources will increase to 75.1% even though the timeout cases are expected to increase.

V. Conclusion

In this paper we evaluated the possibility of collocating CPU job and GPU job on GPU-assisted HPC system to improve the resource utilization. We demonstrated the severeness of the utilization issue with system statistics and theoretical model. We constructed a MPI/CUDA-based workload pool with NPB and SHOC benchmarks, and then studied several system-related and job-related factors over the collocation performance, such as resource isolation, combined CPU load, and cudaMemcpy bandwidth. We also implemented an experimental multi-node system based on Torque and Maui to demonstrate how job collocation can benefit the system's utilization as well as throughput.

Our results indicate that collocating CPU job and GPU job is a feasible way of improving system utilization. With those factors properly informed and managed, the collocated CPU job can efficiently scavenge the underutilized resources without incurring significant performance degradation in either of the collocated jobs. Our experimental 4-node system achieved 15.9% gain in throughput and 10% gain in both CPU and GPU utilization with a moderate resource collection policy. With a stronger policy, the system is capable of collecting 75% of the underutilized CPU cycles.

In the future, we plan to capture more performance factors and to design a comprehensive model for the collocated jobs. Such a model would help the system reliably estimate the underutilized resources. We also plan to design adaptive resource collection policies, so that the system can maintain higher ratio of collectible resources while accommodating irregularity over jobs.

Acknowledgment

This work is supported by the US National Science Foundation under award number CNS-0845583.

References

[1] E. Ayguadé, R. M. Badia, F. D. Igual, J. Labarta, R. Mayo, and E. S. Quintana-Ortí, "An extension of the starss programming model for platforms with multiple gpus," in *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, ser. Euro-Par '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 851–862.

[2] 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*, ser. PACT '10. New York, NY, USA: ACM, 2010, pp. 217–226.

[3] J. Duato, A. Pena, F. Silla, R. Mayo, and E. Quintana-Ortí, "rcuda: Reducing the number of gpu-based accelerators in high performance clusters," in *High Performance Computing and Simulation (HPCS), 2010 International Conference on*. IEEE, 2010, pp. 224–231.

[4] J. Wu, W. Shi, and B. Hong, "Dynamic kernel/device mapping strategies for gpu-assisted hpc systems," in *Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, W. Cirne, N. Desai, E. Frachtenberg, and U. Schwiegelshohn, Eds., vol. 7698. Springer Berlin Heidelberg, 2013, pp. 96–113.

[5] V. Ravi, M. Becchi, W. Jiang, G. Agrawal, and S. Chakradhar, "Scheduling concurrent applications on a cluster of cpu-gpu nodes," in *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, may 2012, pp. 140 –147.

[6] C. Iancu, S. Hofmeyr, F. Blagojevic, and Y. Zheng, "Oversubscription on multicore processors," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, april 2010, pp. 1 –11.

[7] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber *et al.*, "The nas parallel benchmarks summary and preliminary results," in *Supercomputing, 1991. Supercomputing'91. Proceedings of the 1991 ACM/IEEE Conference on*. IEEE, 1991, pp. 158–165.

[8] A. Danalis, G. Marin, C. McCurdy, J. Meredith, P. Roth, K. Spafford, V. Tipparaju, and J. Vetter, "The scalable heterogeneous computing (shoc) benchmark suite," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. ACM, 2010, pp. 63–74.

[9] G. Staples, "Torque resource manager," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM, 2006, p. 8.

[10] D. Jackson, Q. Snell, and M. Clement, "Core algorithms of the maui scheduler," in *Job Scheduling Strategies for Parallel Processing*. Springer, 2001, pp. 87–102.