Using rCUDA to Reduce GPU Resource-assignment Fragmentation caused by Job Scheduler

Pak Markthub*, Akihiro Nomura† and Satoshi Matsuoka‡
Tokyo Institute of Technology
2-12-1 Ookayama, Meguro-ku, Tokyo, 152-8550 JAPAN
*Email: { *markthub.p.aa, †nomura.a.ac }@m.titech.ac.jp, ‡matsu@is.titech.ac.jp*

***Abstract*—In heterogeneous supercomputers such as TSUBAME2.5, GPUs on some nodes in GPU batch queues are left idle even though there are jobs waiting in the queues; this is caused by GPU resource-assignment fragmentation problem. For example, in the case that each node has three GPUs like TSUBAME2.5's, if a node has already been assigned to a job requesting two GPUs per node, that node cannot be assigned to another job requesting more than one GPU per node until the ongoing job finishes; hence, one GPU is left idle on that node. We examine this problem on TSUBAME2.5's GPU batch-queue system and present a scheduling algorithm that assigns rCUDA (a remote CUDA execution technology) to some processes of some jobs. Because rCUDA allows jobs to utilize the idle GPUs, the proposed scheduling algorithm can alleviate the problem. Using a job pattern obtained from a scheduler log of a TSUBAME2.5's GPU queue, our simulation shows that the proposed algorithm can decrease jobs' lifetime (from the time when a job arrives until finishes) by about 5% on average. Moreover, it can reduce the average number of idle GPUs by about 15%. Also, even reducing the number of nodes serving jobs by around 4%, the proposed algorithm can maintain the average jobs' lifetime around the same as the scheduling algorithm currently used in the TSUBAME2.5's GPU queue.**

***Keywords*-GPU queue; GPU execution; rCUDA; remote GPU execution; scheduling algorithm;**

I. Introduction

A. Resource sharing in TSUBAME2.5 job queues

Jobs submitted to a heterogeneous supercomputer, such as TSUBAME2.5 at Tokyo Institute of Technology, can be roughly categorized into three categories. The first category is GPU-intensive jobs, which need one or more GPUs per node for processing. They normally use CPUs for management – distributing data to GPUs, handling communication between nodes (e.g. MPI), and collecting results from GPUs – but not for calculation; hence, only a small number of CPUs per node are enough to serve this kind of jobs. The second category is CPU-only jobs. In contrast with GPU-intensive jobs, they only use CPUs for processing and do not use any GPUs; even if they execute on nodes that have GPUs, the GPUs are left idle. The third category is jobs that use both CPUs and GPUs intensively. Unlike GPU-intensive jobs, this kind of jobs use both CPUs and GPUs mainly for processing; thus, they usually require a large number of CPUs, and one or more GPUs per node.

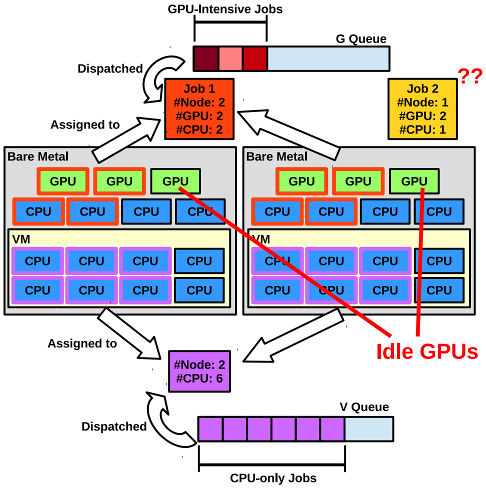

Figure 1. Overview of G queue and V queue architecture with GPU resource-assignment fragmentation problem on G queue

TSUBAME2.5 has *G queue* and *V queue* serving GPU-intensive jobs and CPU-only jobs respectively. G queue and V queue share 480 nodes with each other. GPU-intensive jobs in G queue get exclusive access to three GPUs and four CPUs per node. On the other hand, CPU-only jobs in V queue share virtual machines (VMs) with some other CPU-only jobs. The VMs are hosted on the same group of nodes that the G queue uses. Out of 12 CPUs equipped in each node, each VM gets 8 CPUs to serve CPU-only jobs, and 4 CPUs are left for host OS to serve GPU-only jobs. Fig. 1 shows an overview of G queue, V queue, and the nodes shared between those two queues. As GPU-intensive jobs and CPU-only jobs use different resources (GPUs vs. CPUs) for processing, sharing a node increases overall resource utilization of the system. In this paper, we focus on GPU-only jobs executed on the G queue.

B. GPU resource-assignment fragmentation problem

Theoretically, it is possible to assign a node to more than one GPU-intensive job at a time providing that the

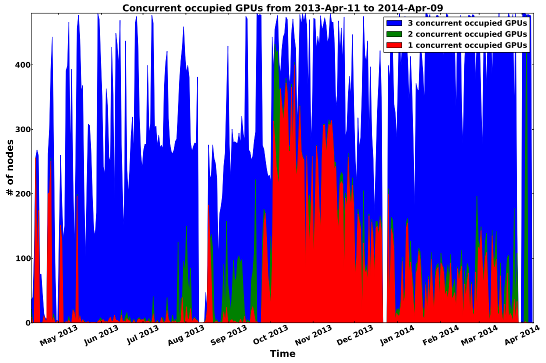

Figure 2. The number of nodes in G queue that had some number of GPUs being occupied during April 2013 to April 2014

available resources (GPUs, CPUs, memory, etc.) are enough. However, the current scheduling algorithm of the G queue allows a node to be assigned to only one job at a time. This results in GPU resource-assignment fragmentation. Fig. 1 shows this kind of situation. Job 1 and Job 2 arrived at the G queue. Job 1 got dispatched and two nodes have been assigned to the job as requested. At this time, there are two idle GPUs, each from a different node. Unfortunately, Job 2 cannot use these GPUs because it requests a single node that has two GPUs. Therefore, these two GPUs have to be left idle and Job 2 cannot run. This is a loss for both user and provider.

Fig. 2 shows the GPU usage pattern of the G queue from April 2013 to April 2014. This information was captured from the Utilization Monitoring System of TSUBAME2.5. Remind that each node has three GPUs, the graph shows how many GPUs on assigned nodes were left idle as illustrated by the red and the green areas. The nodes that fell into these areas have the fragmented GPU resources that could have been assigned to some other jobs.

The paper is organized as follows. We first examine the effect of using rCUDA, a GPU virtualization middleware, on the system and on the performance of applications. We present a mathematical model of the execution time of applications that use rCUDA, and propose a scheduling algorithm that deals with the GPU resource-assignment fragmentation problem. We evaluate the algorithm by simulating jobs whose parameters (resources requirements, arrival pattern, etc.) were obtained from TSUBAME2.5's G queue's scheduler and highlight the benefits of using the proposed scheduling algorithm over the current G queue's scheduling algorithm of TSUBAME2.5.

II. GPU VIRTUALIZATION USING RCUDA

rCUDA [2], [3] is a CUDA-compatible GPU virtualization middleware that we used to alleviate the GPU assignment-fragmentation problem by allowing an application to use GPUs remotely. rCUDA library intercepts all CUDA calls

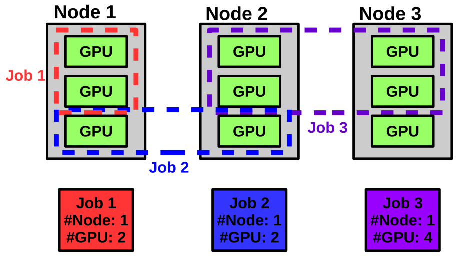

Figure 3. Example of GPUs assignment enabling by rCUDA

of applications that use it and forwards all information regarding the calls to rCUDA servers running on remote hosts. The rCUDA servers execute the calls on the GPUs of nodes that they are running on, and return all results of the executions back to the caller. Moreover, programs that are able to execute using native CUDA can also run using rCUDA without modification because rCUDA provides the same API as native CUDA. More details about how rCUDA works and its performance analysis can be found in [2], [3], [4], [5], [6], [7].

Because rCUDA enables applications executing on one node to use GPUs on other nodes, it removes the node boundary for GPU assignment from the viewpoint of a scheduler. For example, as shown in Fig. 3, a job can get two GPUs from the same node as usual like Job 1; or it can get two GPUs, each from different nodes as Job 2. Therefore, it could be used to solve the GPU assignment-fragmentation problem. Furthermore, it is possible for a job to request the number of GPUs per node more than the number of GPUs physically exists on a node, like Job 3. This opens an opportunity for jobs, which currently cannot run on the G queue due to the number of GPUs, to run without changing hardware. However, because rCUDA uses network to send and receive data about intercepted CUDA calls, the network may affect the performance of the GPU-intensive jobs.

The effect of network contention on applications using rCUDA

We conducted experiments on the effect of network contention using cudaMemcpy with rCUDA. We used a small testbed which consists of two compute nodes; each of the nodes had one 6-core Intel i7-3930K, one Nvidia Tesla K20c connected via PCI-E Gen3 8x, and 48 GB memory; the nodes were connected to each other via FDR InfiniBand. The speed measured by *ib_read_bw* was 6.38 GB/s and the latency measured from *ib_read_lat* was 5.97 μs. We used ib_read_bw apps to generate the network contention on the experiment system. Since we configured the InfiniBand to use fair share, the average bandwidth for cudaMemcpy is given by (1). We also built a model for estimating cudaMemcpy data transfer time. The model is expressed by

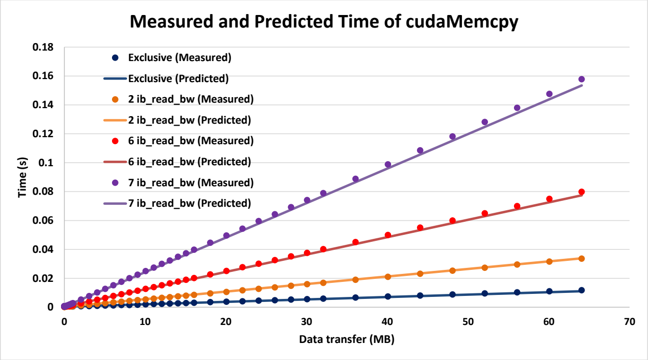

Figure 4. The effect of network contention on cudaMemcpy using rCUDA

(2).

$$bw_{\text{eff}} = \frac{net\_bw}{\#ib\_read\_bw + 1} \tag{1}$$

$$\begin{aligned} time_{\text{rcuda}} = {} & (rcuda\_lat + net\_lat)(gpu\_call\_count) \\ & + \frac{datasize_{\text{app}}}{bw_{\text{eff}}} overhead_{\text{rcuda}} \end{aligned} \tag{2}$$

where bw_{eff} is the bandwidth of the application; net_bw is the total bandwidth of the network; $time_{\text{rcuda}}$ is the time per CUDA kernel call when using rCUDA; $rcuda\_lat$ is the additional latency when using rCUDA; net_lat is the latency of the network; gpu_call_count is the number of CUDA calls; $datasize_{\text{app}}$ is the data transfer size of the call; and $overhead_{\text{rcuda}}$ is the additional bandwidth overhead when uses rCUDA. We first conducted an experiment using cudaMemcpy with rCUDA without running ib_read_bw. By using the regression analysis, we obtains the value of $overhead_{\text{rcuda}} = 1.09$ and $rcuda\_lat = 3.47\text{ms}$.

As shown in Fig. 4, the data transfer time using cudaMemcpy fits to our model quite well; the fitness of the model (R^2) is 0.986. As rCUDA always uses network for transferring data between local memory and remote GPU, this model can be applied to all types of GPU data transfer. Nevertheless, because the latency of rCUDA is more than 500 times greater than that of network latency (InfiniBand) and GPU latency, we ignore the network latency and GPU latency in the model.

According to the model, applications using rCUDA might suffer from the network contention because of the decrease in network bandwidth. The host-to-device and device-to-host data transfer may not be affected as much as device-to-device data transfer due to the fact that the total network bandwidth and host-and-device transfer bandwidth are approximately the same (7 GB/s for InfiniBand FDR and K20c GPU). But in the case of device-to-device data transfer, the native bandwidth of GPU is around 170 GB/s, which is a lot higher than that of the network. Also, depending on the same-host GPU-to-GPU and inter-host communication pattern, the amount of slow down of each application is not the same. Fortunately, many jobs executing on the G queue are nearly embarrassingly parallel; thus, the contention on the network should not strongly affect inter-host communication. However, the contention of the network might directly affect the applications' performance due to same-host GPU-to-GPU communication – programmers do not expect same-host GPU-to-GPU communication to go over network, but rCUDA changes its nature.

Another thing that we need to be concerned about is the latency. Since the latency of using rCUDA is a lot greater than that of network and GPU transfers, applications that use rCUDA should avoid frequent GPU data transfer as much as possible; hence, it would be better if applications transfer data to and from GPU in large chunks instead of frequently transferring small chunks.

III. R Queue: rCUDA-integrated GPU queue

Even though rCUDA enables a scheduler to assign GPUs to a job regardless of the physical boundary, jobs that use rCUDA may suffer from the network contention; therefore, the scheduler needs to be careful when assigning the jobs to use rCUDA. We propose *R queue*, a scheduling algorithm that allows the use of rCUDA to share GPU resources, as the replacement of the current G queue. Fig. 5 and Fig. 6 show the simplified version of the algorithm of G queue and the algorithm of R queue respectively. In the simplified version of the G queue's scheduling algorithm, the scheduler dispatches a job in the first come first serve basis. The first job in the queue gets dispatched if there are enough resources to satisfy the requirements of the job. Even though the full version of the algorithm also takes other properties into consideration, such as job's dependency, priority, etc., we simplified it in order to make the comparison with the R queue algorithm easier to understand. For R queue, the algorithm first tries to find as many suitable nodes as possible to execute the jobs by using the simplified G queue's algorithm. If the number of suitable nodes is not enough, the algorithm tries to find more nodes using 2-step resource finding. First, it tries to find a node that has enough resources excluding GPUs (e.g. CPUs, memory) to serve a process of the job. Then, the algorithm tries to find some nodes that have idle GPUs and ask that process to use rCUDA servers running on those nodes. One thing in common about these two steps is the node that has fewer processes running is selected first. By asking only a subset of processes of jobs to use rCUDA and selecting the nodes that have fewer processes running first, the effect of network contention is minimized while enabling more jobs to execute at the same time.

Someone might also be concerned about security and resource restriction of GPU-intensive jobs if they share nodes; in this case, some sandbox mechanisms such as cgroup or LXC can be employed. Also, such mechanisms

```
def meet_standard_req(server, job):
 return server.cpus >= job.requested.cpus and server.
     memory >= job.requested.memory

def find_GQ_suitable_servers(job):
 suitable_servers = list()
 for server in get_all_servers():
  if len(suitable_servers) == job.requested.node_count:
   break
  if not server.has_job() and meet_standard_req(server,
      job) and server.gpus >= job.requested.gpus:
   suitable_servers.append(server)
 return suitable_servers

def G_queue_main():
 for job in G_queue:
  suitable_servers = find_GQ_suitable_servers(job)
  if len(suitable_servers) == job.requested.node_count:
   server.add(job)
   server.start(job)
  else:
   wait_until_state_change()
```

Figure 5. Simplified version of the scheduling algorithm of G queue

have almost no impact on the performance of applications as indicated in [1], making the sharing more attractive.

IV. Evaluation and Discussion

A. Simulated jobs and simulation method

The data obtained from the scheduler's log of the G queue were composed of many attributes. The relevant attributes used in this simulation are shown in Table I. Despite having a lot of data, some attributes from the log were omitted (e.g. job dependency) due to privacy concern, and some attributes were not recorded at all (e.g. GPU-to-GPU total data transfer). We had to estimate the value of some attributes in order to simulate this jobs set. The estimating methods of each attribute are also shown in the table. The "net_data_size", "gpu_gpu_data_size", and "gpu_cpu_data_size" were set equal to the amount of memory requested. Since applications need local memory to buffer inter-node data transfer, this number can roughly represent the data transferred between nodes (e.g. MPI communication, cudaMemcpy with rCUDA). For "net_conn_count" and "gpu_call_count", we used a uniform random function ranging from 100 to 100,000 to represent the values. The number 100 and 100,000 represents the two extremes of application communication pattern (infrequently transferring data, and frequently transferring data). Since those values were not recorded directly and application names were not disclosed, we used the randomization to mix jobs that had frequent communication and infrequent ones together. This served as a rough approximation of the jobs on the G queue. The closeness of the estimation might still be a concern to some readers; however, we experimented with many values of these estimated data and found out that the results (which we will discuss later) were not significantly different. The main reason is because the R queue algorithm tries to minimize the number of jobs that use rCUDA; therefore, these estimated values affected only a small subset of the jobs.

```
def get_least_occupied_server(server_list):
 return sorted(server_list, key = lambda server: server.
     num_running_jobs)[0]

def R_queue_main():
 for job in R_queue:
  suitable_servers = find_GQ_suitable_servers(job)
  for server in suitable_servers:
   server.add(job)
  revert = False

  # Find more nodes
  while not revert and len(suitable_servers) < job.
      requested.node_count:

   # 1st step: find a node to run a process of the job
   server_list = get_all_servers()
   proc = None
   while len(server_list) > 0:
    server = get_least_occupied_server(server_list)
    if not meet_standard_req(server, job):
     remove_from_list(server_list, server)
    else:
     proc = server.add_base(job)
     suitable_servers.append(server)
     break
   if proc is None:
    revert = True
    break

   # 2nd step: find rCUDA-server nodes for the process
   server_list = [server for server in get_all_servers()
       if server.idle_gpus > 0]
   num_assigned_gpus = 0
   while len(server_list) > 0 and num_assigned_gpus < job.
       requested.gpus:
    server = get_least_occupied_server(server_list)
    proc.assign_rcuda_server(server)
    num_assigned_gpus += 1
    if server.idle_gpus == 0:
     remove_from_list(server_list, server)
   if num_assigned_gpus < job.requested.gpus:
    revert = True
    break

  if len(suitable_servers) < job.requested.node_count or
      revert:
   # Remove the assignment if not enough resources
   for server in suitable_servers:
    server.remove(job)
  else:
   for server in suitable_servers:
    server.start(job)
```

Figure 6. Scheduling algorithm of R queue

Table II shows the parameters of each simulated nodes. All of the values were obtained from the real nodes of the G queue. Our simulator generated sets of the simulated nodes using these parameters and simulated resource occupancy of each node by the simulated jobs. The number of nodes that were generated for each experiment is given in the relevant subsections of those experiments.

The simulator uses time-step-wise numerical method to simulate scheduling pattern. This method concerns only points in time where there is a change in the system such as when a job arrives in the system, or when a job finishes processing. There is no actual job running in the simulation;

Table I
JOB'S ATTRIBUTES, DESCRIPTIONS, AND HOW TO OBTAIN THE DATA

Field Name	Obtaining Method	Description
requested.node_count	Scheduler log	Number of nodes required
atime	Scheduler log	Job arrival time
etime	Scheduler log	Job end time
requested.cpus	Scheduler log	Requested # CPUs per node
requested.gpus	Scheduler log	Requested # GPUs per node
requested.memory	Scheduler log	Requested # memory per node
used.walltime	Scheduler log	Actual execution time
net_data_size	Eq requested.memory	Network transfer size per node
net_conn_count	Rand(100, 100000)	# connection opening per node
gpu_gpu_data_size	Eq requested.memory	GPU-to-GPU transfer size per node
gpu_cpu_data_size	Eq requested.memory	GPU-to-CPU transfer size per node
gpu_call_count	Rand(100, 100000)	# CUDA calls per node

Table II
SOME PARAMETERS OF EACH SIMULATED NODE

Field Name	Value		Description
cpus	8		# CPUs (including hyper-threading)
gpus	3		# GPUs
memory	22	GB	Amount of memory
net_bw	7	GB/s	Network bandwidth
net_lat	1.2	μs	Network latency
gpu_gpu_bw	171	GB/s	GPU-to-GPU bandwidth
gpu_cpu_bw	7	GB/s	GPU-to-CPU bandwidth
gpu_lat	0.99	ms	CUDA call's latency

instead, between each time step, the simulation assumes consistent parameters, for example, available network bandwidth for each application, etc. By using these parameters, the simulator can compute the execution time of a job as follow:

$$time_{\text{exec}} = time_{\text{net}} + time_{\text{gpu\_comm}} + time_{\text{other}} \tag{3}$$

$$\begin{aligned} time_{\text{net}} = {} & (net\_lat)(net\_conn\_count) \\ & + \frac{net\_data\_size \times \#job}{net\_bw} \end{aligned} \tag{4}$$

if the job uses rCUDA,

$$time_{\text{gpu\_comm}} = time_{\text{rcuda}} \tag{5}$$

otherwise,

$$\begin{aligned} time_{\text{gpu\_comm}} = {} & (gpu\_call\_count)(gpu\_lat) \\ & + \frac{gpu\_gpu\_data\_size}{gpu\_gpu\_bw} \\ & + \frac{gpu\_cpu\_data\_size}{gpu\_cpu\_bw} \end{aligned} \tag{6}$$

where $time_{\text{exec}}$ is the execution time of a job; $time_{\text{net}}$ is the time the job spends on network transfer; $time_{\text{gpu\_comm}}$ is the time the job spends on GPU data transfer – both GPU to GPU and GPU to CPU; $time_{\text{other}}$ is the time the job spends doing other things except what is stated above, such as GPU/CPU computation, local disk access, etc; $\#job$ is the number of jobs running on a node assigned to this job. Also, we define a job's lifetime as follow:

$$time_{\text{life}} = time_{\text{wait}} + time_{\text{exec}} \tag{7}$$

where $time_{\text{life}}$ is the lifetime of a job. $time_{\text{wait}}$ is the wait time of the job. Even though the above model is simple, it can be used for estimating the actual execution time of a job. Moreover, we are able to calculate the next time-change of the system and simulate the execution of the job set using the simplified version of the G queue and the proposed R queue thanks to this model.

The disadvantage of using rCUDA compared to native CUDA is quite clear when taking a look at the (3) to (7). A job that uses rCUDA has longer execution time due to the nature of rCUDA that changes GPU communication to network communication; thus, $time_{\text{gpu\_comm}}$ gets longer. Nevertheless, the R queue's algorithm allows two or more jobs to co-exist on a node; this causes the network bandwidth that each job can use to be reduced. The reduction of network bandwidth affects not only jobs that use rCUDA but also jobs that do not. Despite having that negative effect, R queue's algorithm has the advantage that a job can start earlier. Unfortunately, the advantage is not as obvious as the disadvantage. Instead of analyzing the advantage mathematically, we use the following experiments to see the outcome of using R queue compared with G queue.

B. GPU occupancy pattern

In the first experiment, we wanted to know the GPU occupancy pattern of the G queue and the R queue. We simulated five servers and fed 50 jobs, whose parameters (arrival time, requested resources, etc.) were obtained from the G queue's log during a busy period (September 1 - 10, 2013), to our scheduler. We used this subset of all jobs and five nodes of the system in order to more easily visualize the occupancy pattern, which is the purpose of this experiment.

Fig. 7 and Fig. 8 show an example of the GPU occupancy pattern of the G queue and the R queue respectively. The x-axis shows the time since the start of the scheduler. The y-axis shows the GPUs grouped by node. Because each node has three GPUs, each group is divided into three lines indicating GPU0, GPU1, and GPU2 of each node respectively. A block indicates which GPU was being occupied by a job. Blocks having the same color represent the same job. The width of the blocks shows how long the jobs use the GPUs. Note that we used the same simulated job set for both queues. On G queue, each GPU-intensive job is executed on the exclusively assigned nodes; there are no

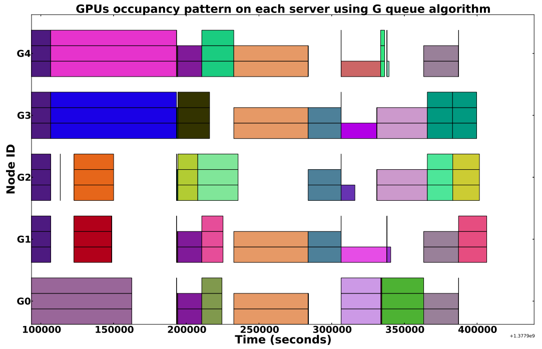

Figure 7. GPUs occupancy pattern on each server using G queue algorithm

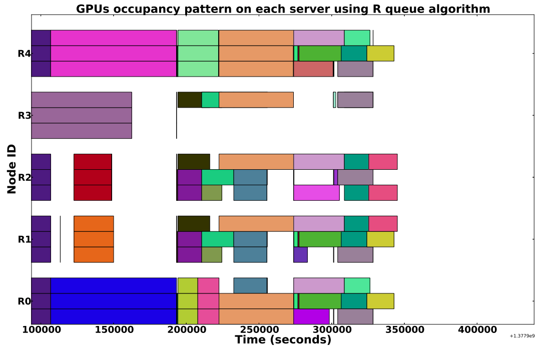

Figure 8. GPUs occupancy pattern on each server using R queue algorithm

shared nodes. Whenever the scheduler cannot find enough nodes to satisfy the request of incoming jobs, these jobs will have to wait. In contrast with the R queue, some nodes are shared among jobs. In most of the time interval, the R queue can process more jobs than the G queue. However, the R queue also makes some jobs execute longer due to the network contention. Despite that, the benefit of the ability to execute more jobs overshadows its disadvantage when there are a lot of jobs in the queue and not all the jobs request the maximum number of GPUs per nodes.

From mathematics' point of view, what R queue's algorithm tried to do is reducing the wait time of each job as much as possible. The judgment of the algorithm is based on information up to the present since it cannot know what kind of jobs may come in the queue in the future. By doing its best on starting currently waiting jobs as soon as possible, it opens more opportunity for later submitted jobs to start early. This reflects on the results shown in Fig. 7 and Fig. 8

C. *The reduction of jobs lifetime on R queue*

In the second experiment, we wanted to know how using the R queue affects the wait time, the execution time, and the lifetime of jobs. We used all 63,484 jobs from the job set obtained from the G queue's log. We generated 480 nodes, which is the same number of nodes in TSUBAME2.5's G queue, and simulated the running of both the simplified G queue and the R queue with these data.

Fig. 9, Fig. 10, and Fig. 11 show the jobs' wait time decrease, execution time decrease, and lifetime decrease, respectively, between the simulated G queue and the R queue. The "time decrease" means that the time on the G queue minus the same time on the R queue; for example, the "wait time decrease" of a job is the wait time of the job in G queue minus the wait time of that job on R queue.

As shown in Fig. 9, the wait time of a job running on the R queue decreased about 25.24% on average compared to the same job running on the G queue. This was due to the R queue allowing some jobs to use rCUDA. These jobs were able to use the idle GPUs on the nodes already assigned to the other jobs, thus they were able to start earlier. Note that some jobs had their wait time increased due to some other jobs taking longer to execute, which was due to the network contention. However, only 0.81% of the jobs had their wait time increased and the average increase was 3.51 minutes.

Regarding the execution time, many jobs got their execution time increased as shown in Fig. 10; however, the average time increase was only about 0.03%. This shows that the R queue's algorithm did well in keeping the effect of the network contention low. Note that the positive value of the x-axis was due to the error from using the step-wise numerical method for the simulation.

Fig. 11 shows the lifetime decrease of the jobs set. As shown, most of the jobs got their lifetime decreased by about 5.06% on average. This means the users got their jobs done earlier when using R queue. Moreover, the average number of idle GPUs was reduced from 78.02 on G queue to 66.56 on R queue, a 14.69% reduction.

D. *Effect of reducing total number of nodes on R queue*

Since the R queue can reduce the lifetime of jobs compared with executing them on the G queue, we can apply this benefit in another way. By reducing the number of nodes serving the R queue, we can still achieve an average jobs' lifetime similar to that of the G queue. We used the same simulator but reduced the number of simulated nodes from 480 to 450 by 10-node step. The job set was the same as the previous section. As shown in Table III, the average lifetime of the jobs running on the R queue increased as the number of nodes decreased; the reason is because there are fewer resources for processing incoming jobs. According to the simulation, when reducing the number of nodes by 20 we still got the average lifetime of the jobs executing on the R queue almost equal to those executing on the G queue.

V. Related Work

There are a lot of work trying to address heterogeneous cluster scheduling problems; however, few has addressed the GPU resource-assignment fragmentation problem. Yamagiwa and Wada [8] considered the resources usage contention between CPU-based applications and GPU-based

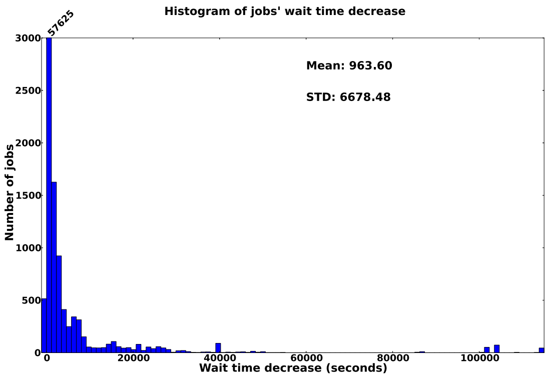

Figure 9. Histogram of the jobs grouped by the difference between the wait time of the jobs executing on the G queue and those executing on the R queue

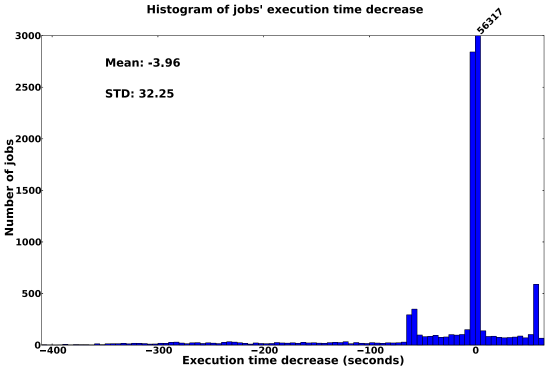

Figure 10. Histogram of the jobs grouped by the difference between the execution time of the jobs executing on the G queue and those executing on the R queue

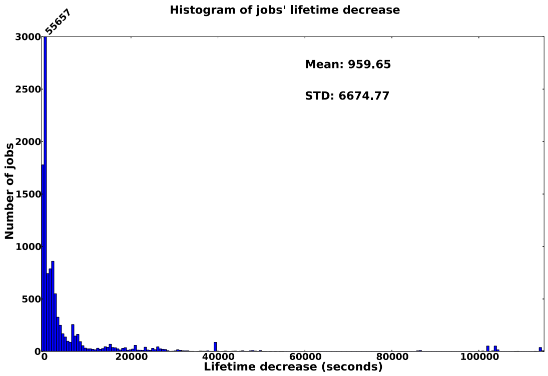

Figure 11. Histogram of the jobs grouped by the difference between the jobs' lifetime (wait time + execution time) of the jobs executing on the G queue and those executing on the R queue

Table III
AVERAGE OF THE WAIT TIME DECREASE (SECONDS), THE EXECUTION TIME DECREASE (SECONDS), AND THE LIFETIME DECREASE (SECONDS) OF THE SIMULATED JOBS EXECUTING ON THE R QUEUE AND THOSE OF THE SAME JOBS EXECUTING ON THE G QUEUE

# Nodes	**Wait time decrease**		**Execution time decrease**		**Lifetime decrease**	
	Avg.	**STD**	**Avg.**	**STD**	**Avg.**	**STD**
450	-901.49	4,236.04	-5.81	39.46	-907.29	4,238.53
460	-32.87	3,654.48	-4.55	35.27	-37.42	3,651.69
470	449.14	3,277.54	-4.31	34.41	444.83	3,272.51
480	963.60	6,678.48	-3.96	32.25	959.65	6,674.77

applications. They showed that two GPU-intensive jobs executing on the same node might suffer from competitive resource access. For example, if the jobs use the same set of GPUs, the execution on GPUs is serialized; therefore, they both will suffer from GPU latency. Fortunately, this is not the case for the R queue. Even if we do share nodes, we do not share GPUs or CPUs between jobs.

Regarding research concerning multi-GPU system, Ravi et al. [12] presented a scheduling method that takes the possibility of a GPU-based job that can be executed on CPUs into consideration. For example, LAMMPS can be configured to either mainly do processing on GPUs, or use only CPUs for processing. This possibility can increase overall utilization of the system. If a job cannot be scheduled to use GPUs due to not enough available resources, it can be scheduled to use CPUs instead. Still, it relies on an assumption that there are a lot of such jobs in the system, which is not true in the TSUBAME2.5's G queue. Also, they did not address the GPU resource-assignment fragmentation problem.

Another interesting one is [13]. They integrated integer-programming optimization technique to SLURM scheduler for GPU queue. The integer-programming optimization allowed the scheduler to schedule jobs more wisely. Therefore, the scheduler minimized the fragmentation of GPU resources. However, their approach cannot remove the physical boundary of nodes in the way we can when using our proposed algorithm.

VI. CONCLUSION

This paper has considered the GPU resource-assignment fragmentation problem on heterogeneous supercomputers caused by job schedulers. The fragmentation prohibits the efficient distribution of all GPU resources among multiple jobs on a given node, resulting in some GPUs being left idle even though there are jobs waiting. We have shown how this problem persisted in the TSUBAME2.5's GPU queue (G queue) and proposed the use of rCUDA, which allowed using GPUs remotely, to alleviate this problem. Due to the fact that rCUDA causes applications to be more affected by network contention, we have presented a model to predict the increase in applications' runtime when using rCUDA. The model shows that the overhead of using rCUDA will vary with effective network bandwidth. This suggests that we should avoid having a lot of jobs using rCUDA running at the same time. We also have proposed the R queue, which is a job scheduler that supports rCUDA, as a replacement for the G queue. We have shown, by way of simulation, how the

R queue allows a small subset of jobs to remotely use GPUs on nodes that have been assigned to other jobs; this reduces the amount of idle GPUs in the system, thus increases overall resource utilization. We have extracted an actual job set from the TSUBAME2.5's G queue's scheduler's log and executed it on the simulated G queue and the R queue. Our results indicate that the jobs, after submission, complete on average 5% faster on the R queue than the G queue. The simulated R queue has about 15% less idle GPUs than the G queue. The algorithm of the R queue and the results also tell us that by allowing a small subset of jobs to start early, the overall utilization increases significantly. Besides this, we also show another benefit of the R queue; that is it allows us the reduce the number of nodes in the queue by approximately 4% (20 nodes) and still maintain the same jobs' lifetime as with the G queue; thus, using the R queue, we have more capacity left for serving more jobs or for other usage.

VII. Future Work

Even though the simulation results show a good improvement of the R queue over the G queue, the algorithm of the R queue is still a heuristic algorithm. For the future, we plan to analyze the scheduling pattern using online optimization technique. It will allow us to model the queue mathematically and compare the utilization of our algorithm with the optimal one. Moreover, the R queue still does not take scheduling based on network, such as network distance between nodes, into account. Since rCUDA changes all GPU-relevant communication into network communication, R queue will cause some issues from the network's point of view. However, integrating multiple objectives into one scheduler will complicate the algorithm; we plan to include a technique of using multiple schedulers to achieve multiple objectives, like in [9], [10], [11], into our work instead. This will allow us to achieve higher utilization and mitigate the network issue while keeping our algorithm simple.

Acknowledgment

This research was supported by JST, CREST (Research Area: Advanced Core Technologies for Big Data Integration). Also, we are grateful to Global Scientific Information and Computing Center, Tokyo Institute of Technology for providing anonymized job execution history of TSUBAME2.5 supercomputer.

References

[1] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. a. F. De Rose, "Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments," *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pp. 233–240, Feb. 2013.

[2] J. Duato, F. Igual, and R. Mayo, "An efficient implementation of GPU virtualization in high performance clusters," *Euro-Par 2009 Parallel Processing Workshops*, pp. 385–394, 2010.

[3] J. Duato, A. Pena, and F. Silla, "rCUDA: Reducing the number of GPU-based accelerators in high performance clusters," *Proceedings of the 2010 International Conference on High Performance Computing and Simulation (HPCS 2010)*, 2010.

[4] J. Duato, A. Pena, and F. Silla, "Enabling CUDA acceleration within virtual machines using rCUDA," *Proceedings of the 2011 International Conference on High Performance Computing (HiPC 2011)*, 2011.

[5] J. Duato, A. J. Pena, F. Silla, R. Mayo, and E. S. Quintana-Orti, "Performance of CUDA Virtualized Remote GPUs in High Performance Clusters," *2011 International Conference on Parallel Processing*, pp. 365–374, Sep. 2011.

[6] C. Reaño, A. Pea, and F. Silla, "Cu2rcu: Towards the complete rcuda remote gpu virtualization and sharing solution," *Proceedings of the 2012 International Conference on High Performance Computing (HiPC 2012)*, 2012.

[7] C. Reaño and R. Mayo, "Influence of InfiniBand FDR on the performance of remote GPU virtualization," *Proceedings of the IEEE Cluster 2013 Conference*, 2013.

[8] S. Yamagiwa and K. Wada, "Performance study of interference on gpu and cpu resources with multiple applications," *IEEE International Symposium on Parallel & Distributed Processing, 2009. IPDPS 2009.*, 2009.

[9] R. Curry, C. Kiddle, and R. Simmonds, "Policy Based Job Analysis," *21st International Symposium on High Performance Computing Systems and Applications (HPCS'07)*, pp. 19–19, 2007.

[10] C. Reiss, A. Tumanov, and G. Ganger, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *SoCC '12 Proceedings of the Third ACM Symposium on Cloud Computing*, 2012.

[11] M. Schwarzkopf and A. Konwinski, "Omega: flexible, scalable schedulers for large compute clusters," in *EuroSys '13 Proceedings of the 8th ACM European Conference on Computer Systems*, 2013, pp. 351–364.

[12] V. T. Ravi, M. Becchi, W. Jiang, G. Agrawal, and S. Chakradhar, "Scheduling Concurrent Applications on a Cluster of CPU-GPU Nodes," *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pp. 140–147, May 2012.

[13] S. Soner and C. Ozturan, "Integer Programming Based Heterogeneous CPU-GPU Cluster Scheduler for SLURM Resource Manager," *2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems*, pp. 418–424, Jun. 2012.