Vol.2016-HPC-153 No.27
2016/3/3

Serving More GPU Jobs in Multi-GPU Batch-Queue Systems using Remote GPU Execution and Migration (Unrefereed Workshop Manuscript)

Pak Markthub[1,2,a)] Akihiro Nomura[1,2] Satoshi Matsuoka[1,2]

Abstract:
Despite having many jobs waiting, a multi-GPU batch-queue node-sharing system normally holds noticeable number of idle GPUs. In our previous work, we presented a scheduling algorithm called RQ that uses remote GPU execution to get rid of the scattered idle-GPU problem, the main cause of the idle-GPU phenomenon, however the nature of RQ creates performance problems for GPU jobs. We examine the severity of these performance problems in various situations using simulation and statistical analysis, and propose MRQ scheduling algorithm, the improved version of RQ that combines remote GPU migration to solve the performance problems caused by the nature of RQ. By ways of simulation and statistical analysis, we found that MRQ outperforms RQ in every situation, and can further reduce a job's lifetime (waiting time + execution time) as well as can tolerate more GPU communication intensity as much as five and ten folds respectively.

1. Introduction

Normally, there is a GPU job waiting to use a multi-GPU batch-queue node-sharing system even though the idle resources of that system are sufficient to serve the job. The scattered idle-GPU problem is mainly responsible for this phenomenon. For example, a system composed of two nodes, each has three GPUs, cannot concurrently serve two jobs requesting two GPUs per node when one job requests one node and the other requests two nodes. Although there are adequate resources to serve both jobs, the locations of the unoccupied GPUs make the system incapable of fulfilling both jobs at the same time. **Figure 1** depicts the problem and our previous solution we are going to discuss in the following paragraph.

In our previous work [1, 2], we proposed a solution to the scattered idle-GPU problem. The main concept of our proposed solution is using remote GPU execution to virtually move an unoccupied GPU from one node to another node. By virtually consolidating unoccupied GPUs into a node, that node could become usable for serving a waiting job. Although this method seems to be able to naturally solve the problem, the jobs using remote GPU execution have to pay the price with extra execution time, due to the overhead of remote GPU execution. Moreover, other jobs in the system may suffer from the increasing in network traffic produced by remote GPU communication. To reduce this negative effect, we proposed a method for extending an existing scheduling algorithm that minimizes the use of remote

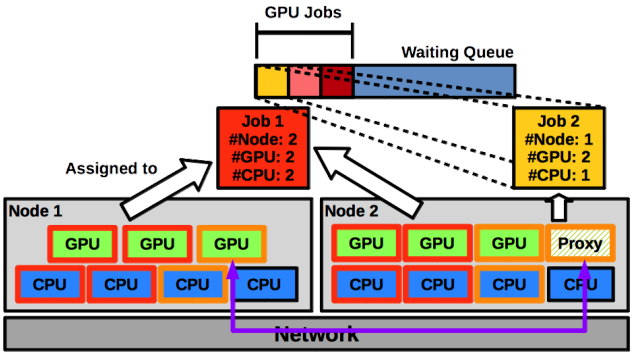


Fig. 1: An example of how to use remote GPU execution to solve the scattered idle-GPU problem in a multi-GPU system. Without using remote GPU execution to virtually move a GPU from Node 1 to Node 2 (by creating a proxy), the system cannot concurrently serve Job 1 and Job 2.

GPU execution while solving the scattered idle-GPU problem. Our implementation used rCUDA [3, 4], the state-of-the-art remote GPU execution middleware, to create a GPU proxy linking to an unoccupied GPU on another node; hence, the number of unoccupied GPUs of a node can be virtually increased. Additionally, we created **RQ scheduling algorithm** to show how to extend the first-come-first-serve (FCFS) scheduling algorithm to solve the scattered idle-GPU problem using our proposed method. Figure 1 illustrates an example of our solution.

Although our proposed solution for solving the scattered idle-GPU problem tries to minimize the negative effect of using remote GPU execution, the solution is still not optimal in some

[1] Tokyo Institute of Technology
[2] JST CREST
[a)] markthub.p.aa@m.titech.ac.jp

situations. The overhead of remote GPU execution, especially rCUDA, depends on GPU communication [1, 2]. High number of GPU invocations and large GPU data transfer can increase the job's execution time more than five folds [5]. This implies that using our solution may not produce good results (sometimes worse than using the original scheduling algorithm) if the job set composes mostly of GPU-communication-intensive jobs. Furthermore, we found that even minority of jobs get their execution time increased up to less than 2%, the gain from having shorter waiting time could be cut down more than two folds as a consequence of the domino effect – longer execution time of a running job prevents a waiting job to start earlier, which in turn causes another job to wait longer.

To improve the performance of our solution, we propose a new solution that combines the use of remote GPU migration to our previous solution. We also give a proof by ways of simulation and statistical analysis that the new solution can effectively solve the performance problems remote GPU execution causes, and can largely lower the extra execution time. The contributions of this work are as follows: 1) analysis regarding how the characteristics of jobs affect the performance of our previous solution, especially of RQ scheduling algorithm, 2) a way to combine a remote GPU migration technique to the previous solution to solve the performance problems, 3) MRQ scheduling algorithm, a realization of the new solution which improves RQ scheduling algorithm, and 4) the performance comparison of FCFS, RQ, and MRQ on various job characteristics. Our simulation shows that MRQ performs better and can tolerate more GPU communication intensity than RQ by 2.5 and 10 folds respectively for the recorded job set from TSUBAME2.5's G queue system.

2. RQ Performance Problem Analysis

Knowledge of how our previous solution works is crucial for understanding the performance problems we are trying to solve. This section provides a review of remote GPU execution (the core technique for eliminating the scattered idle-GPU problem used in our previous solution), a summary of RQ scheduling algorithm (an extension of FCFS scheduling algorithm that realizes our previous solution), and comprehensive discussion on the performance problems caused by the nature of our previous solution in the context of RQ. Readers are encouraged to read our previous work [1, 2] for more information.

2.1 Remote GPU Execution and RQ Scheduling Algorithm

Remote GPU execution is a technique for a process to execute its GPU code in a GPU physically residing on another node. Its general concept is capturing any GPU-related work (GPU execution code, GPU communication commands, etc.) and sending them to a daemon process to execute the captured work on the remote node's GPU. **Figure 2** shows that concept. There are many remote GPU execution implementations such as rCUDA [3, 4], VOCL [6], GridCuda [7], etc. In our previous work, we chose rCUDA because it had very low overhead compared with many other implementations [8, 9] and supported up-to-date GPU technologies especially CUDA.

Our previous solution uses the concept of remote GPU

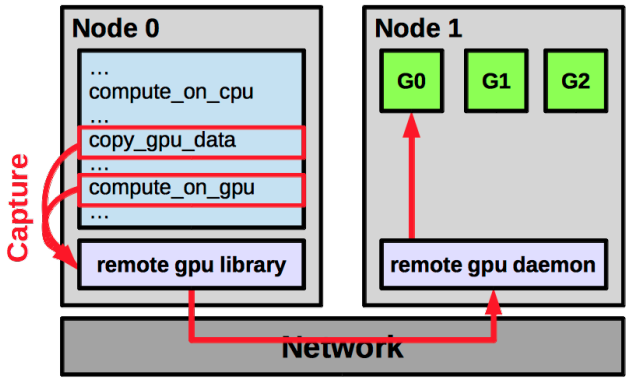


Fig. 2: How remote GPU execution works

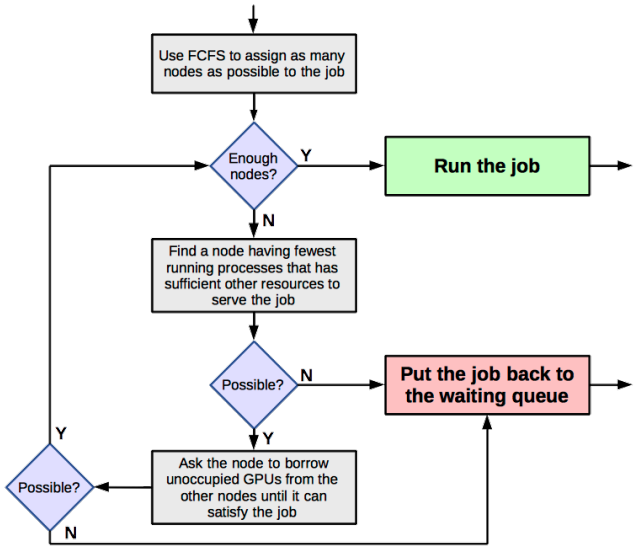


Fig. 3: How RQ scheduling algorithm assigns resources to serve a waiting job

execution to solve the scattered idle-GPU problem. Consolidating as small number of unoccupied GPUs as possible into a node to make that node suitable to serve a waiting job is its main concept. Because remote GPU execution changes the nature of GPU communication from intra-node to inter-node communication and adds extra processing layers to any processes that use it, the more processes use remote GPU execution the congest the network becomes and more jobs get their execution time increased. In other words, our previous solution suggests that the system should use a based scheduling algorithm to schedule as many waiting jobs as possible, and only use remote GPU execution to virtually consolidate unoccupied GPUs to serve more waiting jobs when the based scheduling algorithm cannot find a way to do so. **Figure 3** shows the overview of how RQ scheduling algorithm, which extends FCFS scheduling algorithm, works.

2.2 Remote GPU Execution Overhead Problem

In our previous work [1, 5], we thoroughly discussed about the overhead of remote GPU execution, especially of rCUDA. However, since the knowledge regarding this overhead is

necessary for understanding the rest of the paper, we summarize the important aspects and implications of our rCUDA overhead analysis in this section.

The overhead of rCUDA follows the equation below:

$$time_{\text{rcuda}} = (rcuda\_lat + net\_lat)(gpu\_call\_count) + \frac{gpusize}{bw_{\text{eff}}} overhead_{\text{rcuda}} \quad (1)$$

where $time_{\text{rcuda}}$ is the time for communicating with a remote GPU; $rcuda\_lat$ is the additional latency for communicating with the remote GPU per invocation; net_lat is the latency of the network; gpu_call_count is the number of GPU invocations; $gpusize$ is the amount of GPU data transfer; bw_{eff} is the effective bandwidth the process gets, which will be discussed in Equation (2) of Section 2.3; and $overhead_{\text{rcuda}}$ is the additional bandwidth overhead when using the remote GPU. For rCUDA version 5.0, the values of $rcuda\_lat$ and $overhead_{\text{rcuda}}$ are 50.62 μs and 1.03 respectively. The equation predicts that a job which frequently invokes a GPU or transfers large amount of GPU data, later refer to as **GPU-communication-intensive job**, will get its execution time increased largely when using a remote GPU. In our previous work [5], we showed that LAMMPS [10, 11], an application that has high number of GPU invocations, had the execution time increased more than five times when using rCUDA, as the equation predicts.

2.3 Congested Network Problem

Remote GPU execution does not only add extra execution time to processes that use it, but also makes other processes running in the same time period suffer from more congested network traffic. Remote GPU execution changes GPU communication of a process to network communication. Not only this means the overhead of communicating with a remote GPU depends on the quality of the network, as highlighted in Equation (1), but also means the other processes sharing the same network have to be suffered from more network traffic generated by the remote GPU communication. Moreover, by nature our previous solution enables more jobs to use the system concurrently. This implies that communication intensive (both GPU and network) jobs are likely to experience severely longer execution time when using our previous solution.

For better understanding and latter use, we also give a mathematical model expressing the congested network problem in this section. Assuming that each node has only one network interface that has maximum bandwidth capacity bw_{max} and a node can communicate to any other nodes such that the network topology is not the main cause of network contention, if all running processes continuously use the network and the system implements a fair-shared policy, the effective bandwidth bw_{eff} each running process gets is:

$$bw_{\text{eff}} = \frac{bw_{\text{max}}}{num\_processes} \quad (2)$$

where $num\_processes$ is the number of running processes sharing the same node. According to the generic architecture of remote GPU execution illustrated in Fig. 2, one daemon process is needed to handle the communication with a remote GPU. Hence, the number of remote GPUs also reduce the effective bandwidth, in addition to the number of processes sharing the same nodes. Although this model expresses the best case scenario as the network topology is ignored, we can see that the expected effective bandwidth of each job is likely to be lower when the system uses RQ compared with FCFS.

2.4 Cross-Node Remote GPU Assignment Problem

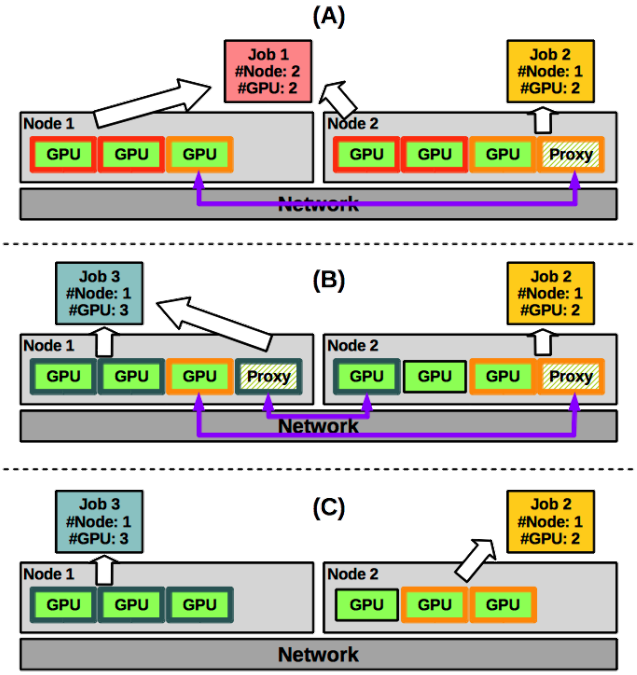


Fig. 4: An example of the cross-node remote-GPU assignment problem

GPU assignment conducted by RQ may end up in a suboptimal state even though RQ makes optimal decisions for every assignment. One weak point of the remote GPU execution technology RQ uses (i.e. rCUDA) is it only allows static assignment. Once a process is assigned with a remote GPU, the process has to continue using that remote GPU until it finishes, even if a local GPU becomes available while the process is running. This technological limitation in RQ leads to suboptimal assignment, which we call cross-node remote GPU assignment problem. **Figure 4** shows an example of this problem on a system composing of two nodes, each has three GPUs. Job 1 requesting two nodes, with two GPUs each, enters the system when there is no job running. We can see that the best way to assign resources to Job 1 is as shown in Fig. 4.A; letting Job 1 uses a remote GPU at this point is not optimized since it will create unnecessary network traffic and overhead as we have already explained. Then, Job 2 requesting a node with two GPUs comes in while Job 1 is still running. With RQ, Job 2 can concurrently use the system with Job 1 by using remote GPU execution as shown in Fig. 4.A. After that, Job 1 finishes and Job 3 requesting a node with three GPUs comes in while Job 2 is still running. The only way to let Job 3 concurrently uses the system with Job 2 is as shown in Fig. 4.B. However, the optimal assignment should be as shown in

Fig. 4.C. This tells us that optimal decisions may lead to a suboptimal state.

The cross-node remote GPU assignment problem caused by the nature of RQ could amplify the severity of the remote GPU execution overhead and the congested network problems. As discussed, too much use of remote GPUs could make all jobs running in the system suffer from additional execution time. Having suboptimal assignment means the severity of the overhead and the congested network problems may be higher than it should be. This implies that with the optimal assignment the system could finish more jobs earlier. In the next section, we will introduce remote GPU migration which could elegantly solve the cross-node remote GPU assignment problem.

3. MRQ: Improving RQ with Remote GPU Migration

3.1 Remote GPU Migration

Remote GPU migration is a technique for migrating execution on a remote GPU to another GPU. Its general concept is making the states and data on the destination GPU (the GPU to be migrated to) the same as those on the source remote GPU. There are many technologies that implement this technique, each has different applications, for example mrCUDA [5], remote GPU migration for VOCL [12], NVCR [13], etc. For improving RQ scheduling algorithm, we choose mrCUDA because it has low overhead, can completely cut off rCUDA's overhead after migration, and allows applications to seamlessly migrate execution on a remote GPU to a local GPU on demand.

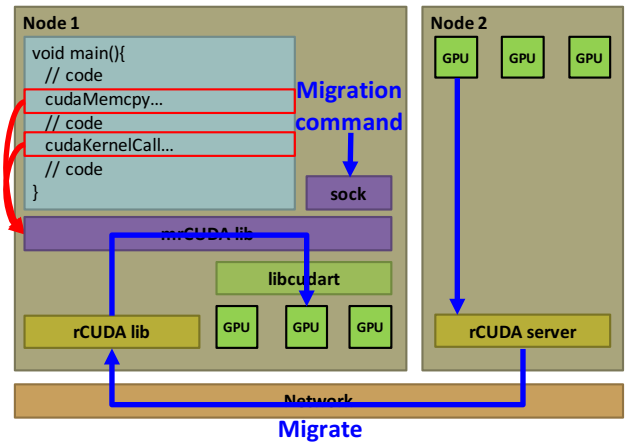


Fig. 5: The architecture of mrCUDA

mrCUDA [5] is an extension of rCUDA we developed to enable GPU execution migration from a remote GPU to a local GPU at runtime. While a process is using a remote GPU, mrCUDA records any CUDA commands that could affect the remote GPU's states and active memory regions. During the migration, mrCUDA replays the recorded commands on the selected local GPU to make that local GPU's states the same as those on the remote GPU, and copies data on the active memory regions to the local GPU. After the migration completes, successive CUDA commands are passed to the local GPU; thus, the remote GPU can be released from its duty. mrCUDA supports one-way one-time migration from a remote GPU to a local GPU, not vice versa. This feather comes from the fact that using a local GPU is always better than a remote GPU in term of performance. The one-way one-time migration feather is a strong point of mrCUDA because a process will suffer from migration at most once, and then rCUDA's overhead can be completely cut off. **Figure 5** shows the architecture of mrCUDA.

There are four types of overheads associated with mrCUDA: record, replay, memsync, and mhelper. We thoroughly discussed about these overheads in our previous paper [5], however we restate them again in this section because they are important for understanding the rest of this work and we are going to use them later in Section 4.1. Equation (3) to Equation (7) express those overheads and **Table 1** contains the descriptions and constant values of the variables in the equations.

$$time_{\text{record}} = (record_{\text{coef}})(num\_record) + record_{\text{const}} \tag{3}$$

$$time_{\text{mhelper}} = (mhelper_{\text{coefd}})(data\_size) + (mhelper_{\text{coefc}})(num\_calls) + mhelper_{\text{const}} \tag{4}$$

$$time_{\text{replay}} = (replay_{\text{coef}})(num\_record) + replay_{\text{const}} \tag{5}$$

$$time_{\text{memsync}} = \sum_{i}^{num\_region} \left(\frac{data\_size_i}{bw[data\_size_i]} + memsync_{\text{coef}} \times data\_size_i + memsync_{\text{const}} \right) + time_{\text{rcuda}} \tag{6}$$

$$bw[data\_size_i] = min(data\_size_i \times bw_{\text{coef}}, bw_{max}) \tag{7}$$

mrCUDA's overheads can be categorized into two categories: migration overhead and operation overhead. Migration overhead is the additional time the process suffers during the migration of a remote GPU; hence, it happens at most once per remote GPU. Replay and memsync overheads are in this category since they are the overheads for copying the states and data from a remote GPU to a local GPU respectively. Operation overhead, on the other hand, prolongs the execution time while the process is using mrCUDA. Record overhead is the additional time for recording relevant CUDA commands before the migration; hence, it taxes the process when the process is using a remote GPU. MHelper overhead, however, taxes the process after the migration finishes, given that the migrated GPU is not the first GPU being migrated for this process. MHelper overhead comes purely from a technical problem regarding the overlapped address spaces when using multiple remote GPUs. In summary, the additional time mrCUDA taxes a process is as expressed in Equation (8).

$$time_{\text{mrcuda}} = time_{\text{record}} + time_{\text{replay}} + time_{\text{memsync}} + time_{\text{mhelper}} \tag{8}$$

According to the case study in our previous work [5], the

Table 1: Description and constant value of each variable in the mrCUDA overhead model equations

Variables	Values	Descriptions
$bw[data\_size_i]$	-	host-to-device memory copy bandwidth
bw_{coef}	47.21 ms^{-1}	GPU's memory copy bandwidth coefficient
bw_{max}	4.78 GB/s	maximum host-to-device GPU's memory copy bandwidth
$data\_size$	-	GPU data transfer size
$data\_size_i$	-	GPU data size of the active memory region i to be transferred
$memsync_{\text{coef}}$	0.057 s/GB	mem-sync overhead's coefficient
$memsync_{\text{const}}$	$\sim$ 0 s	mem-sync overhead's constant
$mhelper_{\text{coefc}}$	9.983 μs	mhelper overhead's GPU call coefficient
$mhelper_{\text{coefd}}$	0.687 s/GB	mhelper overhead's GPU data transfer coefficient
$mhelper_{\text{const}}$	2.934 ms	mhelper overhead's constant
num_region	-	number of active memory retions
num_calls	-	number of CUDA-related calls that are sent through mhelper processes
num_record	-	number of calls recorded
$record_{\text{coef}}$	0.2825 μs	record overhead's coefficient
$record_{\text{const}}$	0.3437 ms	record overhead's constant
$replay_{\text{coef}}$	1.031 μs	replay overhead's coefficient
$replay_{\text{const}}$	1.243 s	replay overhead's constant
$time_{\text{memsync}}$	-	mem-sync overhead
$time_{\text{mhelper}}$	-	mhelper overhead
$time_{\text{rcuda}}$	-	rCUDA's overhead
$time_{\text{record}}$	-	record overhead
$time_{\text{replay}}$	-	replay overhead

migration and the operation overheads accounted for less than 5% and 0.01%, respectively, of LAMMPS's total execution time, while rCUDA's overhead accounted for more than 80%. Readers are encouraged to consult our previous paper [5] for detailed explanation.

3.2 MRQ Scheduling Algorithm

As discussed in Section 2, there are three problems causing a job's execution time to increase originated by the nature of our previous solution; all of them can be solved with remote GPU migration. The remote GPU execution overhead problem occurs because of communication between the process and a remote GPU. Remote GPU migration technologies, especially mrCUDA, can migrate all works on the remote GPU to a local GPU; hence, there is no further communication after the migration. For the congested network problem, remote GPU daemon processes play a major role in decreasing the effective bandwidth of all processes. After migration to a local GPU, the remote GPU daemon process associated with the migrated remote GPU can be terminated. This frees up the network resource, and all processes in the system benefits from the migration. For cross-node remote GPU assignment problem, remote GPU migration can improve GPU assignment to the optimal state. For example, mrCUDA can change the GPU assignment as shown in Fig. 4.B to Fig. 4.C by migrating the remote GPU assignment of Job 2 to a local GPU Job 1 releases when leaves the system, before assigning all GPUs in Node 1 to Job 3.

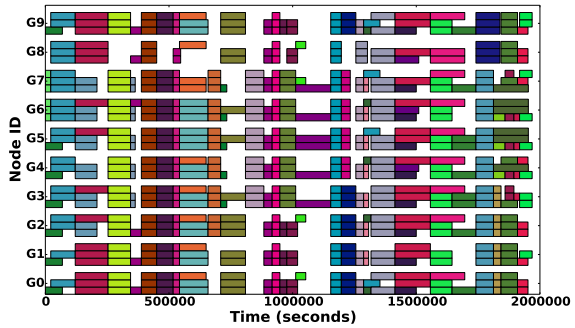
Our new solution improves the previous solution with remote GPU migration. The main concept of our new solution is the same as that of the previous solution (stated in Section 2) with an addition that a process should return a borrowed GPU as soon as a local GPU becomes available – even before considering assigning this GPU to a new job. There are two situations that lead to a local GPU becomes available: a job leaves the system, and the process that has borrowed it returns it back. In both cases, immediately migrating the execution on a remote GPU to the released local GPU is always the best decision since all the performance problems associated with using the remote GPU are solved as soon as possible.

MRQ scheduling algorithm is a realization of our new solution. It combines RQ scheduling algorithm with mrCUDA to solve the performance problems caused by the nature of RQ. How MRQ works is very similar to RQ except it uses mrCUDA instead of rCUDA. Also, as soon as a GPU becomes available, MRQ takes a look at that node to see whether there is a borrowing GPU from another node or not. If there is a borrowing GPU, MRQ will send a migration command to the associated mrCUDA socket to migrate the execution on the remote GPU to the local GPU that has just become available. This mechanism has higher priority than assigning resources to a waiting job; in other words, MRQ considers reassigning a GPU that has become available to a local process before assigning it to a new job.

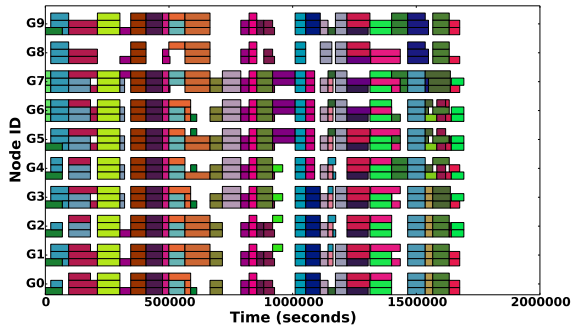
To get an impression on how MRQ may improve RQ, we show, in **Fig. 6**, the GPU occupancy pattern of an example job set on a system composing of ten three-GPU nodes when using RQ and MRQ to schedule. This figure was created from our scheduler we are going to explain in Section 4.1. However, since we use this figure only for explanation not evaluation, we omit the details regarding how we obtained the result. As shown in the figure, most of the jobs have shorter execution time when using MRQ compared with when using RQ. Moreover, the job placement is more compact on MRQ. This is because when a job has shorter execution time, more waiting jobs can use the system earlier. As the result, MRQ can finish this job set earlier than RQ could.

4. Performance Comparison of FCFS, RQ, and MRQ

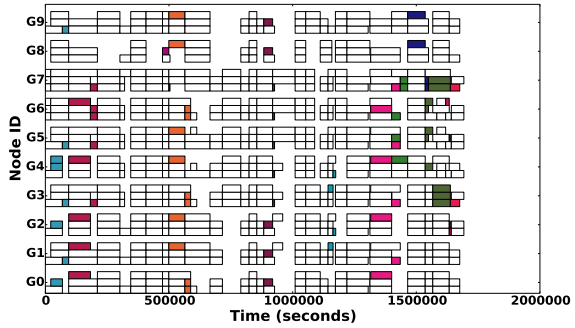
To evaluate our new solution, we compared the performance of RQ, and MRQ against FCFS on various job characteristics on the same system. We used the simulation method we are going to explain in Section 4.1 to obtain the performance of each scheduling algorithm. Since RQ improves FCFS and MRQ improves RQ, we compared the performance of RQ and MRQ against FCFS in the **lifetime decrease** term. Lifetime of a job is the job's waiting time plus execution time, in other words, the time the user has to wait since the submission until the job finishes the execution. Because we expect a job running with RQ to has shorter lifetime compared with running with FCFS, we define lifetime decrease on RQ as the lifetime of a job running with FCFS minus the lifetime of that job running with RQ. Similarly, lifetime decrease on MRQ is defined as the lifetime of a job running with FCFS minus the lifetime of that job running with MRQ. We chose this term as the comparison metric because it captures both the gain and the loss of a job when using RQ/MRQ compared with FCFS.



(a) GPU occupancy pattern when using RQ scheduling algorithm



(b) GPU occupancy pattern when using MRQ scheduling algorithm



(c) GPU occupancy pattern when using MRQ scheduling algorithm colored only the GPUs involved in migration

Fig. 6: Comparison of GPU occupancy pattern of the same job set when using RQ and MRQ

4.1 Simulation Method

We used our simulator written in Python2.7 for simulation. The simulator used time-step-wise numerical method to simulate scheduling patterns. This method concerned only the points in time where there is a change in the system such as when a job arrives to the system, or when a job finishes processing. There was no actual job running in the simulator; instead, between each time step, the simulator assumed consistent parameters (e.g. effective network bandwidth for each job, etc.). By using these parameters, the simulator computed the execution time of a job as follow:

$$time_{\text{exec}} = time_{\text{net}} + \max_{0 \le i \le \#gpu} (time_{\text{gpu\_comm\_i}}) + time_{\text{other}} \quad (9)$$

$$time_{\text{net}} = (net\_lat)(net\_conn\_count) + \frac{net\_data\_size}{bw_{\text{eff}}} \quad (10)$$

$$time_{\text{gpu\_comm\_i}} = time_{\text{cuda}} + time_{\text{rcuda}} + time_{\text{mrcuda}} \quad (11)$$

$$time_{\text{cuda}} = (gpu\_call\_count)(gpu\_lat) + \frac{gpusize}{gpu\_bw} \quad (12)$$

The variables in the above equations are as defined in **Table 2**.

Table 2: Simulation Parameter Definition

Parameter	Definition
$time_{\text{exec}}$	a job's execution time
$time_{\text{net}}$	the time the job spends on network
$time_{\text{gpu\_comm\_i}}$	the time the job spends on the i^{th}-GPU communication
$time_{\text{other}}$	the time the job spends doing other work except what stated above, such as GPU/CPU computation, local disk access, etc.
$\#gpu$	the number of GPUs the job uses
net_lat	the latency of the network
net_conn_count	the number of network connection the job makes
net_data_size	the amount of data the job transfers through the network
$time_{\text{cuda}}$	the time the job spends on local GPU communication; 0 if this is not a remote GPU
$time_{\text{rcuda}}$	defined in Equation (1); 0 if rCUDA is not used for this GPU
$time_{\text{mrcuda}}$	defined in Equation (8); 0 if mrCUDA is not used for this GPU
gpu_call_count	the number of GPU invocations
gpu_lat	the latency of GPU communication
gpu_data_size	the amount of GPU data transfer
gpu_bw	the bandwidth for transferring GPU data

Table 3: Characteristics of each simulated node

Field Name	Value		Description
cpus	8		Number of CPUs
gpus	3		Number of GPUs
memory	22	GB	Amount of memory
net_bw	7	GB/s	Network bandwidth
net_lat	1.2	μs	Network latency
gpu_bw	7	GB/s	GPU communication bandwidth
gpu_lat	10	μs	CUDA call's latency

Our simulated nodes' characteristics always followed the characteristics of the nodes in TSUBAME2.5's G queue system. **Table 3** shows the characteristics of the simulated nodes. We used 100 simulated nodes for all experiments. For the interconnection network, we assumed that any packets originated from a node could reach any other nodes with the same network bandwidth and the same network latency. Hence, we can view the network congestion as it virtually happens at the network interface of each node, as we discussed in Section 2.3.

The job sets we used in each experiment had common characteristics as described in **Table 4**, each job set contained exactly 10,000 jobs. We used probability models to generate the job characteristics to see the effect of each job characteristic on the performance of RQ and MRQ compared with FCFS. For number

Table 4: Characteristics of each simulated job

Variable	Default Value	Description
nodect	$Gauss(30, 30)$ nodes	Number of requested nodes
interarrival_time	$Expo(1)$ seconds	Interarrival time
$time_{\text{other}}$	$Gauss(10^3, 10^2)$ seconds	Walltime when running alone on the system
gpu_call_count	$Gauss(10^6, 10^5)$	Number of GPU invocations
gpusize	$Gauss(\mu, \mu - 1)$ B	Amount of GPU data; always varied from $log(\mu) = 9$ to 12
ngpus	$P[1] = P[3] = 0.1; P[2] = 0.8$	Number of requested GPUs
ncpus	Equal to ngpus	Number of requested CPUs
net_conn_count	$Gauss(10^4, 10^3)$	Number of network invocations
netsize	$Gauss(4, 1)$ GB	Number of requested CPUs
mem	$Gauss(8, 1)$ GB	Amount of requested memory

* $Gauss(\mu, \sigma)$ refers to a Gaussian random variable whose expected value and variance are μ and σ^2 respectively. $Expo(\beta)$ refers to an Exponential random variable whose expected value is β.

of requested GPUs per node (*ngpus*), we let jobs requesting two GPUs per node be the majority by default because this situation is likely to create the scattered idle-GPU problem than letting jobs requesting one or three GPUs per node dominate the system; hence, we can easily see how MRQ performs compared with RQ when facing with a severe scattered idle-GPU problem.

4.2 The Effect of GPU Communication Intensity

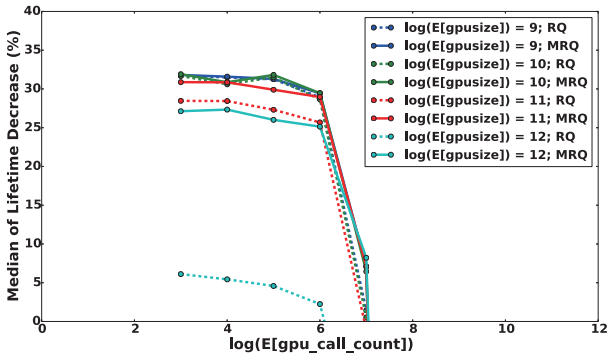


Fig. 7: Lifetime decrease of RQ and MRQ when varying the GPU communication intensity characteristics

There are many job characteristics that may affect the performance of RQ and MRQ. In this subsection, we show how GPU communication intensity, i.e. number of GPU invocations (*gpu_call_count*) and amount of transferred GPU data (*gpusize*), affect the jobs' lifetime when running with FCFS, RQ, and MRQ. We modeled both *gpu_call_count* and *gpusize* with Gaussian random function $Gauss(\mu, \mu - 1)$, while the others followed what stated in Table 4. **Figure 7** shows the result of the experiment. The x-axis represents the logarithm of the expected value of *gpu_call_count*, while the y-axis shows the median of lifetime decrease on RQ/MRQ; we use median instead of mean because it has higher tolerance to outlier values. The result of the experiment tells us that the jobs' lifetime tended to decrease as GPU communication intensity increased. At some points, FCFS performed better than both RQ and MRQ; this was because the loss from the increasing of execution time outweighed the gain from the decreasing of waiting time, as we intensively discussed in Section 2. However, the graph tells us that MRQ was able to handle GPU communication intensity and performed better than or equal to RQ, thank to the migration ability of mrCUDA and how MRQ prioritizes GPU migration over assigning a GPU to a new job.

4.3 The Effect of Job Length

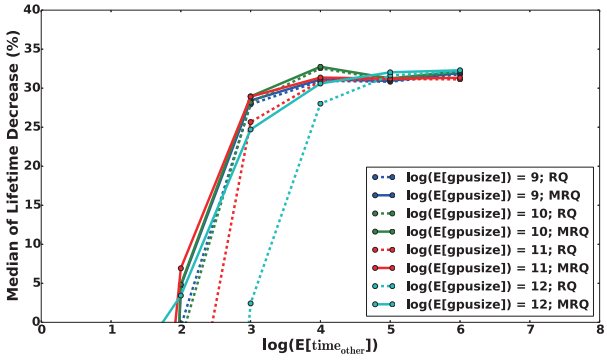


Fig. 8: Lifetime decrease of RQ and MRQ when varying job length for various GPU communication intensity

In this experiment, we varied the length of the jobs in each job set to see the effect of job length on the performance of FCFS, RQ, and MRQ. According to Equation (9), there are three parameters that contribute to a job's execution time: $time_{\text{net}}$, $time_{\text{gpu\_comm\_i}}$, and $time_{\text{other}}$. Since RQ and MRQ affect the first two parameters, we varied $time_{\text{other}}$ to make the job ran shorter or longer; we modeled $time_{\text{other}}$ with Gaussian random function $Gauss(\mu, \mu - 1)$. The result of this experiment is shown in **Fig. 8**. The figure tells us that RQ and MRQ are not good at handling short jobs. This is because $time_{\text{net}}$ and $time_{\text{gpu\_comm\_i}}$ can easily outweigh $time_{\text{other}}$ for such jobs. However, MRQ showed much higher degree of tolerance than RQ; this also thank to GPU migration.

4.4 The Effect of Job Size

The amount of requested resources usually differs for each job. We can see this characteristic as the size of each job. In this experiment, we varied the number of requested nodes (*nodect*) of each job, again with $Gauss(\mu, \mu - 1)$, to see how good RQ and MRQ can manage jobs on each size. **Figure 9** shows the result of this experiment. According to the graph, both RQ and MRQ are better at handling large jobs. This is because as *nodect* increases, FCFS finds it harder to let multiple jobs use the system. In this respect, RQ and MRQ are better since remote GPU execution enables more compact resource assignment. However, the more

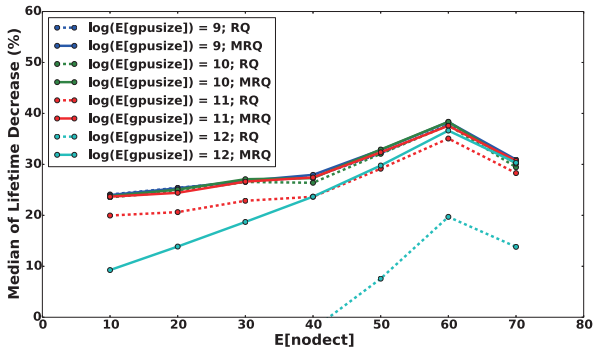


Fig. 9: Lifetime decrease of RQ and MRQ when varying job size for various GPU communication intensity

compact the resource assignment is, the more severe the performance problems, as we discussed in Section 2. As the result, MRQ performed better than RQ because it can solve those performance problems.

4.5 The Effect of Number of Requested GPUs

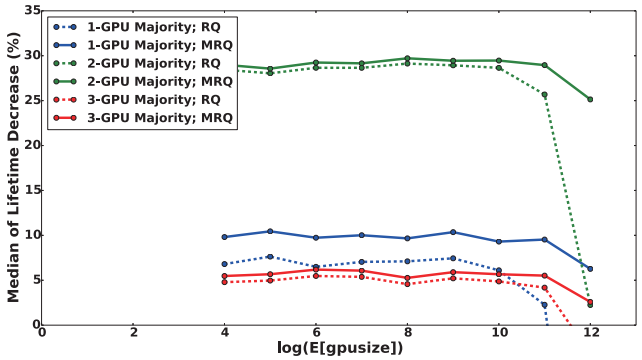


Fig. 10: Lifetime decrease of RQ and MRQ when varying number of requested GPUs for various GPU communication intensity

The number of requested GPUs per node is a major factor that decides how severe the scattered idle-GPU problem is. For a system which has three GPUs per node, such as our simulated system, the severity of the scattered idle-GPU problem is likely to be higher if jobs requesting two GPUs per node dominate the system; since a node having at least one unoccupied GPU can be directly assigned to a job requesting one GPU per node and a job requesting three GPUs per node usually occupied the entire nodes, the severity of the scattered idle-GPU problem is likely to be low if these two types of jobs dominate the system. In this experiment, we varied the probability of the number of requested GPUs per node for each job set to see how well RQ and MRQ can handle those job sets compared with FCFS. **Figure 10** shows the result of this experiment. On the legend, x-GPU majority refers to the probability $P[ngpus = x] = 0.8, \forall x \neq y, P[ngpus = y] = 0.1, x, y \in \{1, 2, 3\}$. According to the figure, the lifetime decrease was much higher (more than three times) when jobs requesting two GPUs per node dominated the system. This result aligns with our hypothesis regarding how the number of requested GPUs affect the severity of the scattered idle-GPU problem. Similar to the other experiments, MRQ performed better than RQ in all cases.

4.6 The Effect of Waiting Queue Length

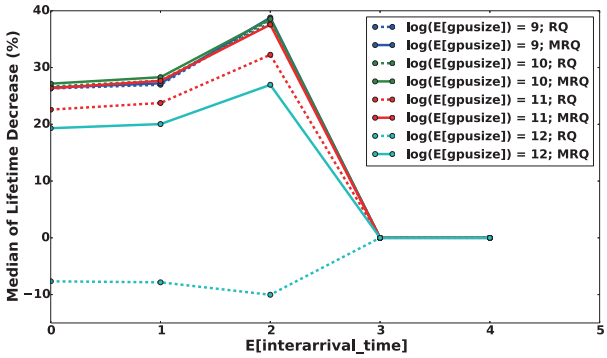


Fig. 11: Lifetime decrease of RQ and MRQ when varying wating queue length for various GPU communication intensity

Unlike the previous experiments which we assumed the system were busy by modeling $interarrival\_time = Expo(1)$ s, in this experiment we show how well RQ and MRQ perform compared with FCFS on various waiting queue length. To simulate various waiting queue length, we varied the interarrival time with $interarrival\_time = Expo(\mu)$ while fixing other parameters. We also varied *gpusize* to show the effect of waiting queue length and GPU communication intensity together. From the result shown in **Fig. 11**, we can see that MRQ performed better than RQ and FCFS in all cases, while RQ performed better than FCFS when the GPU communication intensity was not high. The graph also tells us that when the system is not busy ($E[interarrival\_time] \geq 3$), the performance of FCFS, RQ, and MRQ are the same. This is because RQ and MRQ avoid assigning jobs with remote GPUs; hence, they are reduced to FCFS when the scattered idle-GPU problem does not exist.

5. Case Study: Performance Comparison using TSUBAME2.5's G Queue's Recorded Job Set

Real job characteristics usually do not follow any statistical models; hence, there might be some problems that are not covered in the previous experiments. In this case study, we simulated the job set using the job characteristics recorded in TSUBAME2.5's G queue's scheduler log from Aug, 01 - 15, 2015 (a busy period) in order to see how RQ and MRQ perform compared with FCFS on a real-world situation. Even though the log recorded many important job characteristics such as arrival time, execution time, amount of requested resources, etc., some were not available. To complete all necessary job characteristics for simulation, we set *net_conn_count* and *netsize* to the values shown in Table 4, and set $gpusize = Gauss(10^9, 10^8)$. We varied $gpu\_call\_count = Gauss(\mu, \mu - 1)$ to see the performance of RQ and MRQ on various GPU communication intensity, which was not recorded in the log.

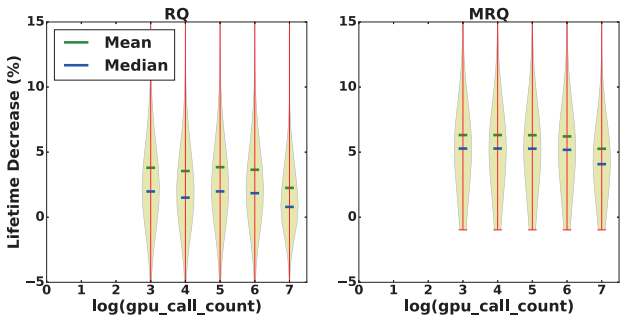


Fig. 12: Lifetime decrease of RQ and MRQ on the job set whose characteristics came from TSUBAME2.5's G queue's scheduler log on various GPU communication intensity

Figure 12 shows the result of this case study. The result shows that MRQ was able to reduce the lifetime about 2.5 times more than RQ on average. Moreover, MRQ were able to handle GPU communication intensity about 10 times better. This finding aligns with the results from the previous experiments. However, this result also shows that there were a lot fewer sacrificed jobs (the jobs that have longer lifetime when using RQ/MRQ compared with FCFS) when using MRQ. This confirms our hypothesis in Section 3 that the new solution can efficiently solve the performance problems, which caused by the nature of our previous solution.

One remark regarding the result is the percent lifetime decrease was not as high as what we have shown in Section 4. This mismatch might come from many factors, however we believe that the number of requested GPUs per node was mainly responsible for this phenomenon. Unlike our assumption in Section 4, the recorded job set had $P[1] = 0.025$, $P[2] = 0.124$, and $P[3] = 0.851$ for *ngpus*, which means that the severity of the scattered idle-GPU problem was much lower in the recorded job set than in our experiments in Section 4. Despite that, using MRQ is still preferred over FCFS as many jobs are still benefit from having shorter lifetime.

6. Related Work

VT. Ravi et al. [14] proposed a scheduling algorithm that can be used to increase overall resource utilization in node-sharing heterogeneous systems. They assumed that jobs using the systems could be switched between CPU-only implementation and GPU implementation. For example, LAMMPS [10] allows users to run on CPU only or use GPUs to accelerate the code. With this assumption, it is possible to let jobs run on only CPUs when there are not enough GPUs available. Despite their method has been proven to produce high resource utilization, in our opinion, there are still a lot of jobs that cannot be switched from GPU implementation to CPU-only implementation. This limits the situation where one can use this technique.

S. Soner et al. [15] proposed a scheduling algorithm that uses a mathematical optimization method to schedule heterogeneous jobs in node-sharing systems. They viewed heterogeneous job scheduling problem as an online scheduling problem and implemented a scheduling algorithm that uses the Integer Programming optimization method to schedule jobs. Since this method can guarantee optimal scheduling from the viewpoint of online scheduling, it can make the best use of all available resources at each time period. In our opinion, one might be able to increase more resource utilization by combining our work, which addresses the scattered idle-GPU problem, to their work as doing so enables the use of Integer Programming optimization to optimally select which jobs should use which remote GPUs as well as when to migrate to local GPUs.

7. Conclusion and Future Work

Our previous solution, which makes the best out of remote GPU execution to solve the scattered idle-GPU problem, enables more GPU jobs to start executing earlier (have shorter waiting time) while having their execution time increases as the trade off. However, in some situations the increasing in their execution time outweighs the benefit of having shorter waiting time. Our new solution combines a remote GPU migration technique to the previous solution to solve the extra execution time problem. By prioritizing migrating execution on a remote GPU to a local GPU that becomes available over assigning that local GPU to a new job, the new solution is proven to outperform the previous solution in every situation. According to our experiments, the new solution can decrease a job's lifetime (waiting time + execution time) as much as five times than the previous solution could. This confirms that remote GPU migration could efficiently solve the performance problem created by remote GPU execution, and suggests that the new solution is preferred over the previous one.

Despite all improvement, the new solution is sill not suitable for short jobs. For future work, we plan to study the scattered idle-GPU problem on a multi-GPU batch-queue node-sharing system usually being occupied with short jobs. We also plan to extend our solution to cover various network topologies and consider using remote-to-remote GPU migration to dynamically optimize remote GPU assignment.

Acknowledgments

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] Pak Markthub, Akihiro Nomura, and Satoshi Matsuoka. Using rCUDA to Reduce GPU Resource-Assignment Fragmentation Caused by Job Scheduler. In *Parallel and Distributed Computing, Applications and Technologies (PDCAT 2014), 2014 15th International Conference on*, pages 105–112. IEEE, 2014.

[2] Pak Markthub, Akihiro Nomura, and Satoshi Matsuoka. Increasing gpu batch queue's utilization using rcuda (unrefereed workshop manuscript). *IPSJ SIG Notes 2014-HPC-145*, 2014(24), 2014.

[3] J Duato, FD Igual, and R Mayo. An efficient implemen-

tation of GPU virtualization in high performance clusters. *Euro-Par 2009 Parallel Processing Workshops*, pages 385–394, 2010.

[4] J Duato, AJ Pena, and Federico 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.

[5] Pak Markthub, Akihiro Nomura, and Satoshi Matsuoka. mrCUDA: A middleware for migrating rCUDA virtual GPUs to native GPUs (Unrefereed Workshop Manuscript). *IPSJ SIG Notes 2015-HPC-150*, 2015(6), Jul 2015.

[6] Shucai Xiao, Pavan Balaji, Qian Zhu, Rajeev Thakur, Susan Coghlan, Heshan Lin, Gaojin Wen, Jue Hong, and Wu-chun Feng. Vocl: An optimized environment for transparent virtualization of graphics processing units. In *Innovative Parallel Computing (InPar), 2012*, pages 1–12. IEEE, 2012.

[7] Tyng-Yeu Liang and Yu-Wei Chang. Gridcuda: a grid-enabled cuda programming toolkit. In *Advanced Information Networking and Applications (WAINA), 2011 IEEE Workshops of International Conference on*, pages 141–146. IEEE, 2011.

[8] Federico Silla. Is remote gpu virtualization useful? `http://rcuda.net/pub/rCUDA_barna_15.pdf`, September 2015. [Accessed: 2015 Oct 12].

[9] Jose Duato, Antonio J. Pena, Federico Silla, Rafael Mayo, and Enrique S. Quintana-Orti. Performance of CUDA Virtualized Remote GPUs in High Performance Clusters. *2011 International Conference on Parallel Processing*, pages 365–374, September 2011.

[10] Lammps molecular dynamics simulator. `http://lammps.sandia.gov` [Accessed: 2015 Jun 03].

[11] Steve Plimpton, Paul Crozier, and Aidan Thompson. Lammps-large-scale atomic/molecular massively parallel simulator. *Sandia National Laboratories*, 18, 2007.

[12] Shucai Xiao, Pavan Balaji, James Dinan, Qian Zhu, Rajeev Thakur, Susan Coghlan, Heshan Lin, Gaojin Wen, Jue Hong, and Wu-chun Feng. Transparent accelerator migration in a virtualized gpu environment. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 124–131. IEEE, 2012.

[13] Akira Nukada, Hiroyuki Takizawa, and Satoshi Matsuoka. NVCR: A transparent checkpoint-restart library for NVIDIA CUDA. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 104–113. IEEE, 2011.

[14] Vignesh T Ravi, Michela Becchi, Wei Jiang, Gagan Agrawal, and Srimat Chakradhar. Scheduling concurrent applications on a cluster of cpu–gpu nodes. *Future Generation Computer Systems*, 29(8):2262–2271, 2013.

[15] Seren Soner and Can Özturan. Integer programming based heterogeneous cpu–gpu cluster schedulers for slurm resource manager. *Journal of Computer and System Sciences*, 81(1):38–56, 2015.