mrCUDA: A middleware for migrating rCUDA virtual GPUs to native GPUs (Unrefereed Workshop Manuscript)

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

Abstract:
rCUDA is a remote CUDA execution middleware that creates virtual GPUs for a CUDA application. Code and data being executed on the virtual GPUs are transparently transferred to actual GPUs on remote nodes and the results of the execution are reported back to the application. With this capability, applications can use nodes that do not have enough idle GPUs by using rCUDA to borrow idle GPUs from some other nodes. However, those applications may suffer from the overhead of rCUDA; especially for applications that frequently call CUDA kernels or have to transfer a lot of data, the overhead can be detrimentally large. We propose mrCUDA, a middleware for transparently migrating rCUDA virtual GPUs to native GPUs at runtime and show that the overhead of mrCUDA is negligibly small compare to the overhead of rCUDA. Hence, mrCUDA allows applications to run on nodes that does not have enough idle GPUs (by using rCUDA) and later migrate the work back to native GPUs (thus, get rid of rCUDA overhead) when available.

1. Introduction

1.1 rCUDA: Remote CUDA Execution Middleware

rCUDA [1, 2, 3] is a CUDA-compatible GPU virtualization middleware developed by the Universidad Politecnica de Valencia, Spain. It composes of two main parts: rCUDA library and rCUDA server. rCUDA library intercepts all CUDA-related calls of an application and forwards the calls to rCUDA servers running on remote hosts. The rCUDA servers execute the calls on the GPUs on their nodes and return the results of the execution back to the rCUDA library, which in turn passes them back to the application. Any applications that use native CUDA can use rCUDA without any need to modify or recompile their source code since the rCUDA library has the same application programming interfaces (APIs) as *libcudart.so*, the CUDA API library. More information regarding how rCUDA works and its performance analysis can be found in [1, 2, 3, 4, 5, 6].

rCUDA has been proven useful in many situations. For example, Duato et al. [3] showed that CUDA applications could be run on a virtual machine by using rCUDA to redirect all of the CUDA-related calls to the GPUs on the host machine. Pena et al. [7] presented that one can reduce the number of GPUs in a cluster by consolidating GPUs into several GPGPU nodes and let CUDA applications running on other compute nodes, which do not have any GPU, to use some GPUs on those GPGPU nodes on demand by using rCUDA. They claimed that this method can lead to more efficient in space, energy, acquisition, and maintenance costs. Also in our previous work [8], we showed that we could get rid of the idle-GPU scattering problem, which in turn increase the resource utilization, in multi-GPU resource-sharing systems by using rCUDA to virtually consolidate idle GPUs scattered on various nodes into one node.

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

1.2 Problem Statement

In our previous work [8], we presented a mathematical model that expresses the overhead of the rCUDA. Since we are going to use that model to discuss about the rCUDA's overhead, we present it here again with updated parameters' values for the latest rCUDA version (v5.0):

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

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; 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; $rcuda\_lat = 50.62\ \mu s$ is the additional latency when using rCUDA; and $overhead_{\text{rcuda}} = 1.03$ is the additional bandwidth overhead when uses rCUDA.

According to the model, we can easily see that the rCUDA's overhead can be detrimentally large for some applications; more concretely, for applications that transfer a lot of GPU data or frequently call CUDA-related functions. For example as shown in **Fig. 1**, LAMMPS [9, 10], a molecular dynamics simulation application, experiences increasingly huge slow down as the number of simulation steps (minor x-axis) increase when using rCUDA. This is because the number of CUDA-related calls and the total amount of GPU data transfer increase as the number of simulation steps increase.

In our previous work [8], we showed that it is possible to

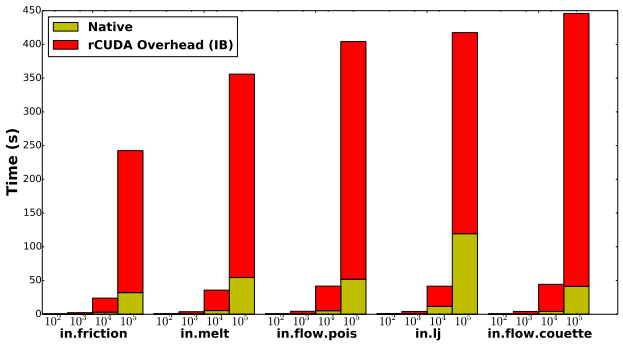


Fig. 1: Comparison of the execution time of LAMMPS on a remote GPU using rCUDA and a local GPU

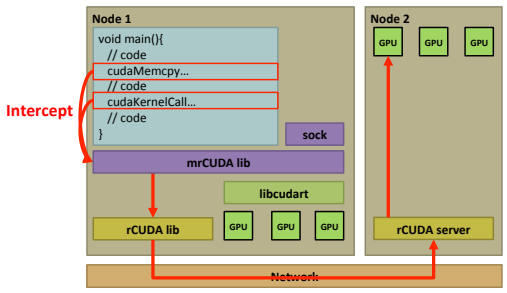
use rCUDA to reduce the wait time of GPU jobs in multi-GPU resource-sharing system by consolidating idle GPUs from different nodes. For example, a node that has three physical GPUs but has only one unoccupied GPU cannot be used to serve a job requesting more than one GPU per node. We showed that if we use rCUDA to "borrow" idle GPUs from other nodes, that node could be used to serve the job given that other unoccupied resources can satisfy the job. Doing so can magnificently reduce the average wait time of GPU jobs while increases their execution time a bit. However, for jobs such as LAMMPS, the large increasing in the execution time due to the rCUDA's overhead can easily overshadow the benefit for the wait-time reduction.

The need to use rCUDA to borrow an idle GPU from another node is, in many cases, temporary but there is no way for GPU applications that use rCUDA to stop using it at the runtime. The main reason why we need to ask a job to use the rCUDA is because the idle GPUs are scattered throughout multiple nodes such that there is not way to run that job, at that moment, using other means. However, it is possible that more local GPUs will be available afterward as some jobs may finish and release their occupying resources. Hence, if we have a way to migrate the execution on a remote GPU to a local GPU, we can enjoy the benefit of reduced wait time while keeping the extra execution time due to the rCUDA's overhead low.

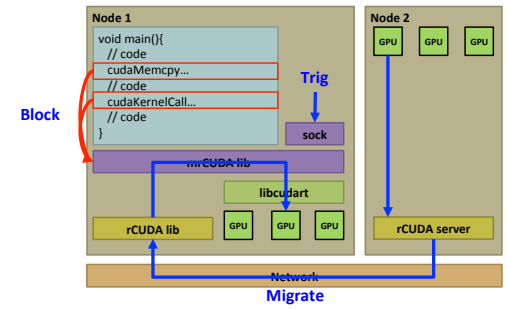
2. mrCUDA's Architecture

mrCUDA, stand for "migratable rCUDA", is a middleware between an application and the rCUDA library that allows the application to transparently migrate code and data running on a remote GPU to a local GPU at runtime. It works by intercepting every CUDA-related calls of the application and passing the calls to the rCUDA library or the native CUDA library as well as having a mechanism to migrate all data and execution from a remote GPU to a local GPU. It also provides a socket that an external application such as a scheduler can send a migration command in to start a migration. The GPU application does not aware of the migration and it can continue using CUDA without having to handle anything. The main concept of the migration is the synchronization between two GPUs' states and memory. We are going to introduce each part of the mrCUDA in this section.

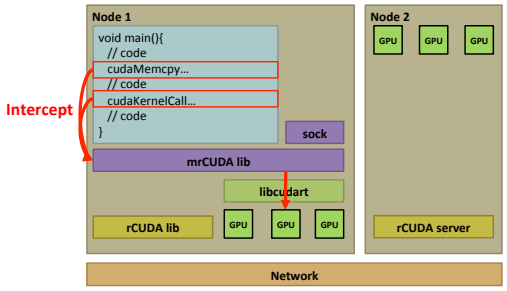
How the mrCUDA allows an application to transparently mi-



(a) rCUDA passthrough mode



(b) Migration mode



(c) CUDA passthrough mode

Fig. 2: Overview of how mrCUDA allows migration from a remote GPU to a local GPU

grate its GPU execution is shown in **Fig. 2**. The mrCUDA first starts operating in the "rCUDA passthrough mode". In this mode, the mrCUDA intercepts all of the CUDA-related calls of the application, "record" the order of some calls as well as active GPU memory regions, and passes the calls to the rCUDA library. **Fig. 2a** shows how it works. The next mode of operation is called "migration mode" as shown in **Fig. 2b**. This mode is activated at the time the mrCUDA receives a migration command via the provided socket. The command is in the form "migrate [virtual GPU ID] [local GPU ID]"; for example, the migration command "migrate 0 1" will ask the mrCUDA to migrate the virtual GPU ID 0 to the local GPU ID 1. At the start of this mode, all of the CUDA-related calls that have not been passed to the rCUDA library are put in a waiting queue. The mrCUDA then issues the *cudaDeviceSynchronize* call to the rCUDA library, which will not return until all the calls running on the remote GPU finish. At this point, the states and the memory of the remote GPU will not change any further. The mrCUDA then "replay" the calls recorded dur-

ing the rCUDA passthrough mode to the selected local GPU in order; doing this ensures that the states of the local GPU will end up the same as those of the remote GPU. After that, the mrCUDA does "mem-sync" which is copying all the active memory regions of the remote GPU to the same region on the local GPU. After it finished the mem-sync process, the memory of the two GPUs are ensured to be the same and the mrCUDA automatically enters the last mode "CUDA passthrough mode" as shown in **Fig. 2c**. In this mode, the mrCUDA first sends the calls in the waiting queue to the native CUDA library and then does the same for any further CUDA-related calls. All of these modes of operation allow the application to transparently migrate the GPU execution.

3. Migration Algorithm

The main concept of transparently migrating a GPU is making the states and the memory of the destination GPU the exact same copy of the origin GPU. The "states" of a GPU in this context include all of the GPU device's flags, executable code registered to the GPU, and active memory's virtual addresses; while the "memory" means the data residing in the active memory regions. Synchronizing the GPU device's flags and the registered executable code is quite simple since we can simply read the device's flags of the remote GPU and configure those of the local GPU with the same value and the registered executable code does not change during the GPU execution. However, synchronizing the active memory's virtual addresses is not a simple task.

Making the virtual address space of an active memory region on a local GPU the same as that on a remote GPU is not straightforward. For example, a virtual address returned from allocating the same size of memory using cudaMalloc for the first time and the second time even on the same GPU are almost always different. However, we need a way to recreate the exact same virtual address spaces of the same allocation on the local GPU since those virtual addresses could be stored anywhere in the program (both in the host memory and the device memory) and may be used later after the migration. A. Nukada. et al. [11] found that if two programs execute CUDA driver API calls in the same order, the virtual addresses returned from the memory allocation calls are always the same. We tested this out on CUDA runtime API – rCUDA supports CUDA runtime API not CUDA driver API – on a remote GPU served by an rCUDA server and on a local GPU. We found that with a little intervention we can make the virtual address spaces of those two GPUs the same. However, this assumption is true only if the remote GPU and the local GPU have the same GPU model and both use the same CUDA driver and runtime API versions.

The need to execute CUDA-related calls in the same order leads to the implementation of the "replay method". The replay method was proposed by Nukada et al. [11] as a way to implement checkpoint/restart for CUDA applications. Their library records the CUDA driver API calls of a program from the start until the latest checkpoint and replays them to get the same GPU virtual address spaces when the program has to restart from a failure. We use the same idea but apply to the CUDA runtime API calls instead. From our experiment, we found that the mrCUDA needs to record only a small subset of CUDA runtime APIs shown in **Table 1** to reproduce the same virtual address spaces on a local GPU. Most of the CUDA runtime APIs can be directly recorded and replayed as they are – the mrCUDA does not need to modify the order or any arguments passed to those calls – however recording and replaying some APIs are not straightforward.

Table 1: CUDA runtime APIs that needed to be recorded and replayed

CUDA API	Special Intervention
__cudaRegisterFatBinary	Need to propagate the new fatCubinHandle's address to some other recorded APIs.
__cudaRegisterFunction __cudaRegisterTexture __cudaUnregisterFatBinary	Need the new fatCubinHandle's address returned from __cudaRegisterFatBinary.
__cudaRegisterVar	Need the new fatCubinHandle's address returned from __cudaRegisterFatBinary and the variable address has to be included into the active memory region table.
cudaMalloc*	The allocated address has to be added to the active memory region table.
cudaFree	The freed address is removed from the active memory region table.
cudaStreamCreate	Two additional calls are needed after finish replaying all of the __cuda*.
cudaBindTexture cudaHostAlloc cudaSetDeviceFlags	-

cudaMalloc*: the API whose name starts with cudaMalloc such as cudaMallocArray.
__cuda*: the API whose name starts with __cuda such as __cudaRegisterFunction.

As shown in Table 1, most of the *__cuda** relies on the *fatCubinHandle* variable, which is returned from calling *__cudaRegisterFatBinary* – the function for putting the GPU code of an application to the GPU devices and thus it is the first CUDA call for every application. However, unlike other CUDA runtime APIs, the *__cudaRegisterFatBinary* usually does not return the same *fatCubinHandle* address. Since the *fatCubinHandle* variable is needed in other functions, the mrCUDA replaces the recorded *fatCubinHandle* parameters of every API calls with the new value when the *__cudaRegisterBinary* function is replayed on the local GPU.

__cudaRegisterVar, *cudaMalloc**, and *cudaFree* affect the allocated memory regions of the GPU. Since the mrCUDA does not have to synchronize the data of non-active memory regions, the mrCUDA keeps a list of active memory regions in a table; the addresses returned from *__cudaRegisterVar* and *cudaMalloc** are put in that table while the addresses passed to *cudaFree* are removed from the table. By using this technique, the mrCUDA can reduce the amount of data it has to copy from a remote GPU to a local GPU.

cudaStreamCreate affects the memory addresses returned from calling *cudaMalloc**. From our experiment, subsequent calls to *cudaMalloc* functions after calling a *cudaStreamCreate* return different memory addresses compare with calling the same sequence of *cudaMalloc* without *cudaStreamCreate*. Moreover, the rCUDA server seems to call *cudaStreamCreate* two times right after finishing all of the initializing *__cuda** function calls even though there is no *cudaStreamCreate* call in the application code. Hence, the mrCUDA has to add two additional *cudaStreamCreate*

```
__cudaRegisterFatBinary
__cudaRegisterFunction
__cudaRegisterFunction
...
__cudaRegisterVar
// Two additional cudaStreamCreate calls are added here
cudaStreamCreate
cudaStreamCreate
cudaMalloc
...
cudaLaunch
__cudaUnregisterFatBinary
...
__cudaUnregisterFatBinary
```

Fig. 3: An example of a recorded CUDA-call list

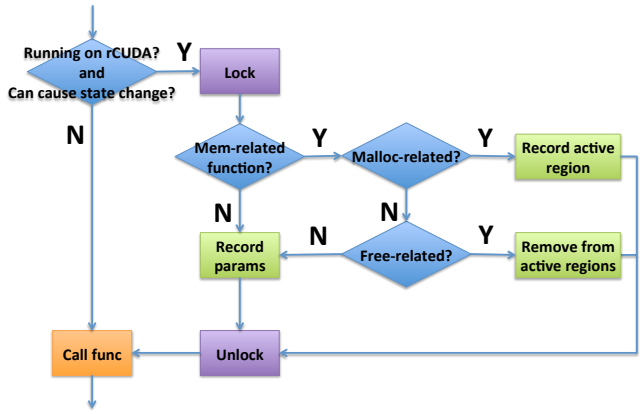


Fig. 4: Record algorithm of mrCUDA

calls to the recorded CUDA-call list right after the first sequence of *__cuda** calls, as shown in **Fig. 3** for example.

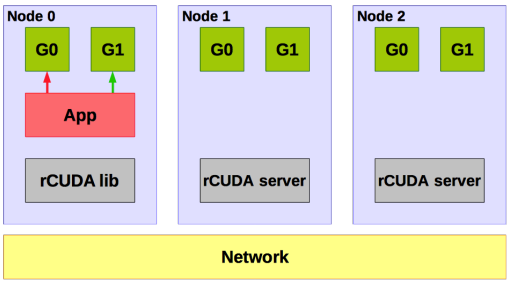
The mrCUDA uses the discussed knowledge in the "record", "replay", and "mem-sync" processes. The overview of the record process is shown in **Fig. 4**. Basically, the mrCUDA takes a look at each intercepted CUDA-related call and decides whether it has to record the call. If it has to record the call, it puts that call in the recorded list as well as handles the special intervention mentioned in Table 1. For the replay process, the mrCUDA first adds two *cudaStreamCreate* to the appropriate position in the recorded list and executes each recorded call in order as well as handles the special intervention. After finish the replay process, the mrCUDA starts the mem-sync process; the mrCUDA takes a look at the active memory region table and copies the data on those regions from the remote GPU to the local GPU. Finishing the mem-sync process marks the end of the migration.

4. Multi-GPU Migration

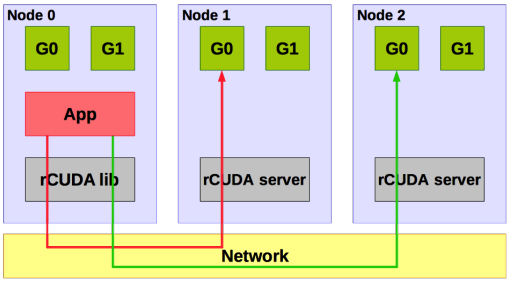
Table 2: Output of executing the simple cudaMalloc program on each configuration shown in Fig. 5

Configuration	a (GPU ID 0)	b (GPU ID 1)
First configuration	0x702e20000	0x705d40000
Second configuration	0x702e20000	0x702e20000
Third configuration	0x702e20000	0x705c40000

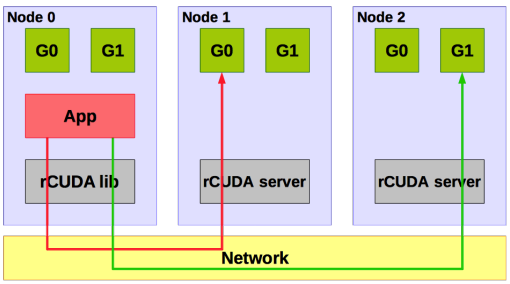
Handling multi-GPU migration is quite complex compare to the single-GPU migration. The main problem is that most of the



(a) First configuration: Local GPUs



(b) Second configuration: Remote GPU 0 of Node 1 and Remote GPU 0 of Node 2



(c) Third configuration: Remote GPU 0 of Node 1 and Remote GPU 1 of Node 2

Fig. 5: Three different configurations of how we can execute a multi-GPU application

time the replay method cannot recreate the exact same GPU virtual address spaces on other local GPUs after the first migration. Consider executing a very simple multi-GPU *cudaMalloc* program using each configuration shown in **Fig. 5** on three nodes that have two NVIDIA Tesla C2050 GPUs. The program calls *cudaMalloc* two times – the first time on GPU ID 0 and the second time on GPU ID 1 – and saves the returned addresses on the variables a and b respectively; the GPU ID 0 and the GPU ID 1 are the GPU IDs seen by the program, which eventually mapped to the red line and the green line, respectively, on each configuration. The result of the execution is as shown in **Table 2**. Looking at the result, one can clearly see that the second GPU' virtual address spaces depends on the configurations. The returned addresses on the first configuration, shows that the values of the variable a and b are different. This happens because the CUDA library implicitly switches between two default contexts when the program uses two GPUs. In contrast, on the second configuration, the values of both variables are the same. The reason is the allocation is done on the same GPU on the different rCUDA servers; hence, both

rCUDA servers virtually use the "same" context and thus return the same virtual addresses. The strange thing happens when an rCUDA server does not use the GPU ID 0 as in the third configuration. In this case, the rCUDA server that uses a GPU other than GPU 0 creates a new context before executing any CUDA-related calls sent from the rCUDA library. This in turn causes the address space to shift; thus, the address space does not correspond to the previous two configurations. These problems prohibit us to directly use the replay method to migrate the GPU execution.

We propose "one-process-one-GPU" execution method and "mhelper" process to help the mrCUDA handling multi-GPU migration. The core concept of the one-process-one-GPU execution method is one process handles only one GPU. With this method, the virtual address spaces of different GPUs can be overlapped (the problem of the second configuration). Moreover, this method allows us to manage the virtual address spaces of different GPUs separately (the problem of the third configuration). The mrCUDA realizes the one-process-one-GPU execution method by creating a mhelper process, which acts like a local rCUDA server, for each GPU migration excepts for the first migration; the mrCUDA uses the application process to handle the first GPU migration.

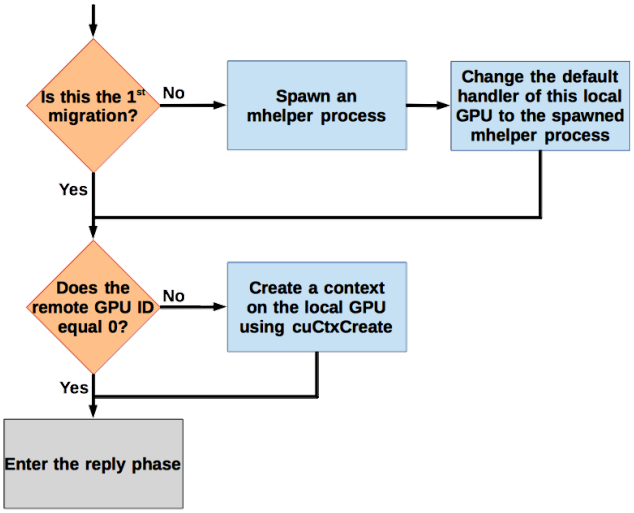


Fig. 6: Pre-replay algorithm for handling multi-GPU migration

The method the mrCUDA uses to migrate multiple GPUs is the same as what we discussed in Section 3 with additional phase before beginning the replay process; we call it "pre-replay" phase and the algorithm for this additional phase is as shown in **Fig. 6**. The pre-replay algorithm is quite easy to understand. Before entering the replay process of any GPU migration, the mrCUDA checks whether this migration is the first migration. If it is not, the mrCUDA spawns an mhelper process and change the default handler of this GPU from the rCUDA library to the spawned process; this means that all further calls to this GPU will be forwarded to the mhelper process. After that regardless the first migration, the mrCUDA checks the remote GPU ID – the remote GPU ID is known to the mrCUDA since the application has been started because it is passed to the rCUDA library as an environment variable. If the remote GPU ID is not zero, this means that a context is created on the remote GPU. The mrCUDA issues a *cuCtxCreate* to the default handler (native CUDA library or an mhelper process) of the local GPU to mimic the context creation of the remote GPU. This concludes the pre-replay phase. One remark regarding why we cannot use an rCUDA server in place of an mhelper process is that the rCUDA server does not support CUDA driver API (*cuCtxCreate* is a CUDA driver API).

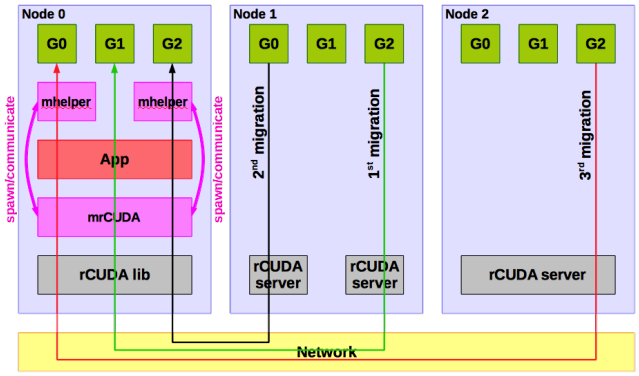


Fig. 7: Example of multi-GPU migration

Figure 7 shows an example of how the mrCUDA handles multi-GPU migration. In the figure, the application uses three GPUs; two remote GPUs from the Node 1 and one remote GPU from the Node 2 – as an rCUDA server uses one process to handle one virtual GPU, it can be viewed as if there are two rCUDA servers on the Node 1. The first migration is the migration from the GPU 2 of the Node 1 to the local GPU 1 of the Node 0. For this migration, the mrCUDA does not spawn an mhelper process since this is the first migration. However, the mrCUDA issues a *cuCtxCreate* to the native CUDA library to create a context on the local GPU 1 because the remote GPU ID is two (not zero). The second migration happens from the remote GPU 0 of the Node 1 to the local GPU 2 of the Node 0. For this migration, the mrCUDA has to spawn an mhelper process but does not have to create a context on the local GPU. The last migration is from the remote GPU 2 of the Node 2 to the local GPU 0. For this migration, the mrCUDA has to spawn an mhelper process and issues a *cuCtxCreate* call. One remark regarding the multi-GPU migration is that there is no need for every migration to happen at the same time; this means that an application can used both local GPUs and remote GPUs concurrently.

5. mrCUDA's Overhead

There are four types of overhead associated with the mrCUDA: record overhead, replay overhead, mem-sync overhead, and mhelper overhead. The record overhead is the additional time for recording CUDA-related calls. It takes effect as long as the mrCUDA operates in the rCUDA passthrough mode. The replay overhead and mem-sync overhead, on the other hand, only take effect during the migration mode as they associate with the replay process and the mem-sync process respectively. The mhelper overhead is only applied for multi-GPU migration. It takes effect in both the migration mode and the CUDA passthrough mode as the mrCUDA needs to communicate with mhelper processes for any GPU-related calls. We begin this section by explaining and

giving a mathematical model for each type of overhead.

The record overhead is always greater than zero regardless the GPU migration occurs. This is because the mrCUDA begins in the rCUDA passthrough mode. In this mode, a small subset of the CUDA-related calls has to be recorded in order to properly replay in the migration mode. Recording these calls take time; even though the time spending on recording a call is very little, a large number of recorded calls might take considerable amount of time. We first give the record overhead model for calculating the time spending on the record process. The record overhead model is as shown in Equation 2.

$$time_{\text{record}} = (record_{\text{coef}})(num\_record) + record_{\text{const}} \quad (2)$$

where $time_{\text{record}}$ is the additional time spending on the record process; $record_{\text{coef}}$ is the coefficient of the record overhead; num_record is the total number of recorded calls; and $record_{\text{const}}$ is the constant of the record overhead.

The mhelper overhead is the additional time spending on the communication between the mrCUDA and an mhelper process. In the current implementation of the mhelper, we use two UNIX pipes for passing messages between the main process and the mhelper process and we also use share memory to copy dynamic-length data between the two processes. These make the communication time of the first migrated GPU and the rest difference. The Equation 3 shows the mhelper overhead model that explains the additional time spending on the communication.

$$\begin{aligned} time_{\text{mhelper}} = {} & (mhelper_{\text{coefd}})(data\_size) + \\ & (mhelper_{\text{coefc}})(num\_calls) + mhelper_{\text{const}} \end{aligned} \quad (3)$$

where $time_{\text{mhelper}}$ is the additional time spending on the mhelper process communication; $mhelper_{\text{coefd}}$ is the coefficient of the overhead for data transfer; $data\_size$ is the total data transfer size between the main memory and the GPU memory such as when the application calls cudaMemcpy; $mhelper_{\text{coefc}}$ is the coefficient of the overhead for CUDA-related calls; num_calls is the total number of CUDA-related calls that go through the mhelper processes; and $mhelper_{\text{const}}$ is the constant of the mhelper overhead.

The replay overhead is the additional time spending on replaying the recorded calls. This overhead only applies to the process that migrates a GPU. Since we include the entire mhelper communication overhead in the mhelper overhead, we ignore the mhelper communication in this model. The replay overhead is as shown in Equation 4.

$$time_{\text{replay}} = (replay_{\text{coef}})(num\_record) + replay_{\text{const}} \quad (4)$$

where $time_{\text{replay}}$ is the additional time spending on the replay process; $replay_{\text{coef}}$ is the coefficient of the replay overhead; num_record is the total number of recorded calls; and $replay_{\text{const}}$ is the constant of the replay overhead.

The last type of the mrCUDA overhead is the mem-sync overhead. It is the additional time spending on copying data from the active memory regions on remote GPUs to local GPUs. This overhead only applies for the processes that do migration. Also, since the mhelper communication overhead has been already included in the mhelper overhead model, we ignore it in this model.

The mem-sync overhead model is as shown in Equation 5.

$$\begin{aligned} time_{\text{memsync}} = time_{\text{rcuda}} + \sum_{i}^{num\_active\_region} \Bigg(& \frac{data\_size_i}{bw[data\_size_i]} + \\ & (memsync_{\text{coef}} \times data\_size_i) + memsync_{\text{const}} \Bigg) \end{aligned} \quad (5)$$

$$bw[data\_size_i] = min(data\_size_i \times bw_{\text{coef}}, bw_{max}) \quad (6)$$

where $time_{\text{memsync}}$ is the additional time spending on the mem-sync process; $time_{\text{rcuda}}$ is the time spending on copying data from the rCUDA, which is defined in Equation 1; num_active_region is the total number of active memory regions; $data\_size_i$ is the size of the data on the active region i. $memsync_{\text{coef}}$ is the coefficient of the mem-sync overhead; $memsync_{\text{const}}$ is the constant value of the mem-sync overhead; $bw[data\_size_i]$, defined in Equation 6 is the host-to-device memory copy bandwidth, which grows linearly to the data size; bw_{coef} is the coefficient of the memory copy bandwidth; and bw_{max} is the maximum host-to-device memory copy bandwidth.

The mem-sync overhead model is not as complicated as its look. According to the mem-sync process explained in Section 3, one can expect that the mem-sync Overhead will depend on the number and the size of active memory regions as well as the overhead of the rCUDA. The part that people who are not familiar with CUDA might not notice is that the size of the data transfer affects the cudaMemcpy's bandwidth; this bandwidth increases linearly as the data size grows and reaches a maximum limit at one point. This knowledge is required for understanding why LAMMPS has low but visible mem-sync overhead, which we are going to discuss in Section 7.

6. Evaluation and Discussion

We conducted an experiment to see the record overhead and the replay overhead of the mrCUDA on LAMMPS. We used two compute nodes of our small testbed; each node 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 via FDR InfiniBand. For the LAMMPS configurations, we used the *in.friction*, *in.melt*, *in.flow.pois*, *in.lj*, and *in.flow.couette* inputs provided by LAMMPS in its example folder. For each input, we varied the number of simulation steps (*run* variable) and tested the migration in four steps: at 25%, 50%, and 75% of the total number of steps, and when there is no migration at all – the mrCUDA is always in the rCUDA passthrough mode. We then timed the record and the replay processes, using the linear regression to find the parameters' values of the corresponding models, and compared each model with its respective results.

Figure 8 shows the result of the experiment. For the record process, the overhead increases linearly as the number of recorded calls increases. Using the linear regression, we find that $record_{\text{coef}} = 2.825 \times 10^{-7}$ s and $record_{\text{const}} = 0.3437$ ms for the record overhead model . The fitness of the model (R^2) is 0.987. One remark regarding the number of recorded calls of the LAMMPS is that it is quite small, around 2%, com-

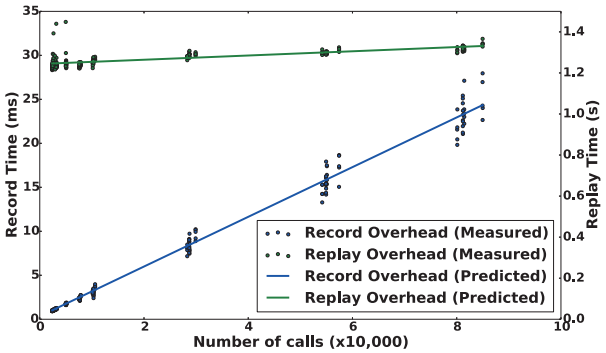


Fig. 8: Measured and predicted record and replay overhead of the mrCUDA when running with LAMMPS

pares to the total number of CUDA-related calls. For the replay process, the result abides the linear increment as the replay overhead model predicts. Using the linear regression, we get $replay_{\text{coef}} = 1.031 \times 10^{-6}$ s and $replay_{\text{const}} = 1.243$ s. The fitness of the model (R^2) is 0.823. The reason why the $replay_{\text{const}}$ is a few magnitudes greater than $record_{\text{const}}$ is that the mrCUDA needs to wait for any calls that have been issued to the remote GPU to finish. Also, $replay_{\text{coef}}$ is about one order of magnitude greater than $record_{\text{coef}}$; this is because the mrCUDA needs to call real functions on a local GPU in the replay process while it just needs to record what the calls are in the record process.

We conducted another experiment to see the mem-sync overhead of the mrCUDA. We performed the experiment on the same compute nodes as the previous experiment but used a very simple program we created instead in order to eliminate the effect of the other overhead. The program we created received the number and the size of *cudaMalloc*. It then used those parameters to create the number of equal-size active memory regions by calling *cudaMalloc* and waited for a migration to happen. We created a hook in our mrCUDA and recorded the time spending on the mem-sync process. By using the linear regression, we got the parameter for Equation 5 and Equation 6.

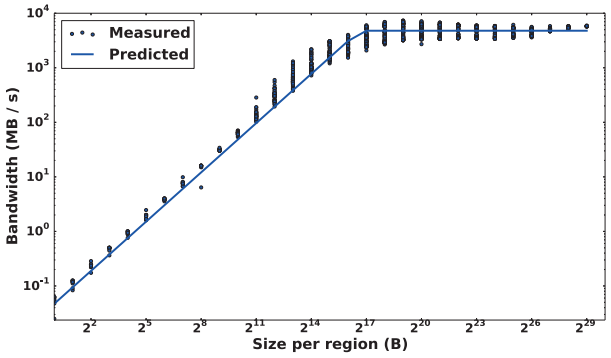


Fig. 9: Measured and predicted host-to-device cudaMemcpy's bandwidth from local cache to a local GPU in the mem-sync process

Figure 9 shows the measured and predicted (Equation 6) values of the host-to-device cudaMemcpy's bandwidth from local cache to the local GPU in the mem-sync process. The graph shows that the bandwidth increases proportionally to the size per region and hits the maximum limit around 2^{17} bytes per region. By using the linear regression on the bandwidth data of the small regions and the average on the rest, we got $bw_{\text{coef}} = 4.721 \times 10^4$ s^{-1} and $bw_{\text{max}} = 4.779$ GB/s. The R^2 is 0.974.

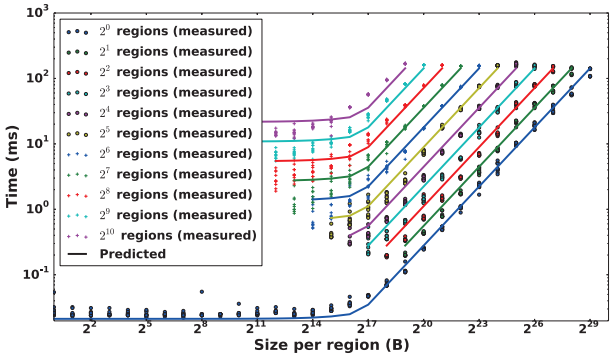


Fig. 10: Measured and predicted mem-sync overhead excluding the rCUDA's overhead of the mrCUDA for various number and size of active regions

Figure 10 shows the measured and predicted (Equation 5) values of the mem-sync overhead without the rCUDA's overhead for various number of and size of active memory regions. The reasons why we got rid of the rCUDA's overhead are because we wanted to see the overhead of our implemented portion (not the rCUDA) and get the values of the unknown parameters in Equation 5. In the graph, the dots represent the measured values while the lines represent the predicted values; each color represents each number of active regions starting from 2^0 to 2^{10}. From the graph, we can see that the number of active regions can significantly increase the overhead while only the large-size regions influence the additional time. This tells us that transferring large but small number of active regions is better than transferring small but large number of active regions. Also, by using the linear regression, we got $memsync_{\text{coef}} = 5.686 \times 10^{-11}$ s/B while $memsync_{\text{const}}$ is almost zero. The R^2 is 0.976.

The last type of the mrCUDA's overheads we measured is the mhelper overhead. For benchmarking this overhead, we created a program that calls a null kernel (an empty CUDA user-defined kernel) several times and copies various-size data from a host's memory to a GPU's memory several times. This program used two GPUs and before we began timing, we forced the mrCUDA to migrate all of the remote GPU execution. Also, those calls were executed on the second-migrated GPU; hence, they went though an mhelper process. Since this program needed two GPUs, we used another three nodes in our testbed; those nodes had the same components and configurations as the nodes we used in the previous experiments expect that each had two NVIDIA Tesla C2050 GPUs. We chose one node to run our program and the other nodes to host rCUDA servers. We also used Equation 3 to predict the outcome of the experiment.

Figures 11 and 12 show the result of the benchmark and predicted values on the null-kernel calling section and the *cudaMem-*

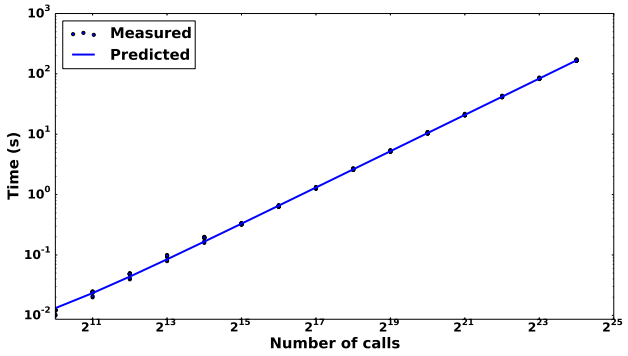


Fig. 11: Measured and predicted mhelper overhead when calling a null kernel for various times

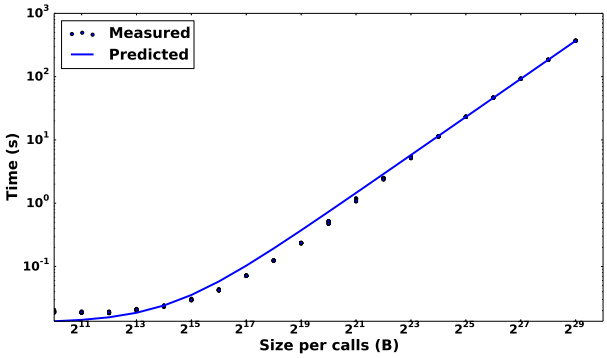


Fig. 12: Measured and predicted mhelper overhead when copying various sizes of data from the host's memory to the second-migrated GPU's memory for 1,000 times

cpy section respectively. In the null-kernel calling section, the *data_size* parameter in Equation 3 is 0. Hence, by using the linear regression, we got $mhelper_{\text{coefc}} = 9.983 \times 10^{-6}$ s and $mhelper_{\text{const}} = 2.934 \times 10^{-3}$ s. The graph in **Fig. 11** shows the linear increment of the overhead as predicted by the model. In the *cudaMemcpy* section, we varied the data size per call but fixed the number of *cudaMemcpy* calls to 1,000 calls. Again, by using the linear regression, we got $mhelper_{\text{coefd}} = 6.871 \times 10^{-10}$ s/B. **Fig. 12** shows that the increasing pattern of the overhead is as predicted. The fitness of the model (R^2) of both figures is 0.999.

7. Case Study: Using mrCUDA to Reduce rCUDA's Overhead at Runtime

As discussed in Section 1.2, some applications may suffer a great deal from the rCUDA's overhead. However, with the help from the mrCUDA, those applications do not necessary have to use rCUDA throughout their execution time if local GPUs are available at some points afterward. We show the benefit of using the mrCUDA with concrete result in this section by showing that the execution time of LAMMPS can be reduced using migration.

We conducted an experiment to see the total execution time of LAMMPS with mrCUDA. We used the same two compute nodes, each had one GPU, and the same LAMMPS configurations as discussed in Section 6. We fixed the run variable of each input to 10^5; the reason why we chose this run number is because the rCUDA's overhead is large and thus we can easily see the benefit of the mrCUDA.

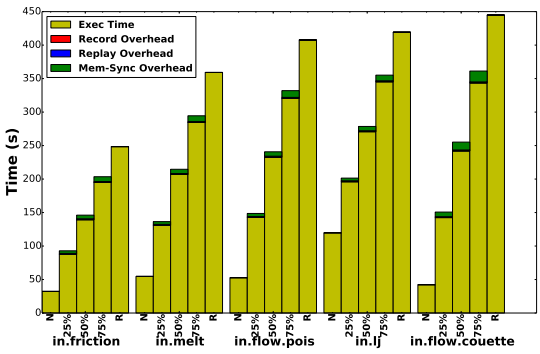


Fig. 13: Execution time with mrCUDA's overhead breakdown of LAMMPS migrated at various simulation steps

Figure 13 shows the execution time of LAMMPS for each input when using the native CUDA, using the mrCUDA with migration, and using the mrCUDA without migration respectively. The major x-axis shows the input names; the minor x-axis shows the tests for native CUDA case (N), migrating at 25%, 50%, and 75% of the total steps (number), and no migration at all (R), respectively. Each of the mrCUDA's overheads is also shown with different color on the same bar of each test. According to the figure, we can clearly see that the total execution time for migration cases grows linearly to the total number of iterations executed before migration. For the mrCUDA's overhead, the record overhead and the replay overhead are negligibly small. However, the mem-sync overhead is noticeable and increases proportional to the migration point; this is because the number of active memory regions slowly increases, which then translated to more overhead.

8. Related Work

S. Xiao. et al. [12] proposed a remote GPU migration technique on the VOCL [13], a remote GPU execution middleware for OpenCL. The main concept is to first make the memory image of the remote GPU that is the source of the migration stable by blocking further commands to be executed on that remote GPU and waiting for the commands being executed to finish. Then, the middleware copies the entire memory of that remote GPU to another remote GPU to synchronize the data. Lastly, the middleware changes the destination of commands to the new remote GPU. This enables OpenCL applications that use the VOCL to live migrate a remote GPU without having to modify the source code. Even though this is a very good work, it only works with OpenCL applications.

A. Nukada. et al. [11] proposed NVCR, a transparent checkpoint/restart library for NVIDIA CUDA. The NVCR library intercepts all of CUDA driver API called by an application and records the essential sequences of execution on a file. Also at a checkpoint, it transfers all of the GPU memory to the file. This, combine with a normal checkpoint/restart library such as Berkeley Lab Checkpoint/Restart [14], allows CUDA applications to restart their computation at a checkpoint on the same or different nodes. Even though the NVCR library is designed for check-

point/restart, we apply the same technique for GPU migration.

CheCUDA [15] is a checkpoint/restart library for CUDA applications. Its main concept is almost the same as the NVCR but applications need to call the provided functions to checkpoint/restart. Actually, the NVCR's authors were inspired by the CheCUDA and made the CheCUDA's checkpoint/restart automatic and transparent.

9. Conclusion and Future Work

mrCUDA is a middleware we developed that allows applications to migrate CUDA execution from remote GPUs to local GPUs at runtime without having to modify the applications' source code. Its main concept is making the states and active memory data of the local GPUs the same as those of the remote GPUs. The migration overhead is quite small compare to the rCUDA's overhead. This allows applications to enjoy using rCUDA to borrow idle GPUs from remote nodes when there are not enough unoccupied local GPUs, and to reduce the rCUDA's overhead by migrating the remote GPU execution to local GPUs when available with very low cost. Our case study on LAMMPS shows the proportional decreasing of the execution time in regards with the migration point.

Even though the mrCUDA has very low overhead compare to the rCUDA's overhead, it is still noticeable. In order to lower the overhead, the overhead models we provided suggest that we should avoid copying a large number of small active memory regions in the mem-sync process as well as finding a way to reduce the mhelper communication overhead. We plan to solve these by aggregating small active memory regions to larger one before copying and using asynchronous and proactive communication with the mhelper in our future work.

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

[2] 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.

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

[4] 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.

[5] C Reaño, AJ 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.

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

[7] Antonio J Peña, Carlos Reaño, Federico Silla, Rafael Mayo, Enrique S Quintana-Ortí, and José Duato. A complete and efficient cuda-sharing solution for hpc clusters. *Parallel Computing*, 40(10):574–588, 2014.

[8] 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.

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

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

[11] 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.

[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] 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.

[14] Paul H Hargrove and Jason C Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. In *Journal of Physics: Conference Series*, volume 46, page 494. IOP Publishing, 2006.

[15] Hiroyuki Takizawa, Katsuto Sato, Kazuhiko Komatsu, and Hiroaki Kobayashi. Checuda: A checkpoint/restart tool for cuda applications. In *Parallel and Distributed Computing, Applications and Technologies, 2009 International Conference on*, pages 408–413. IEEE, 2009.