mrCUDA: Low-Overhead Middleware for Transparently Migrating CUDA Execution from Remote to Local GPUs

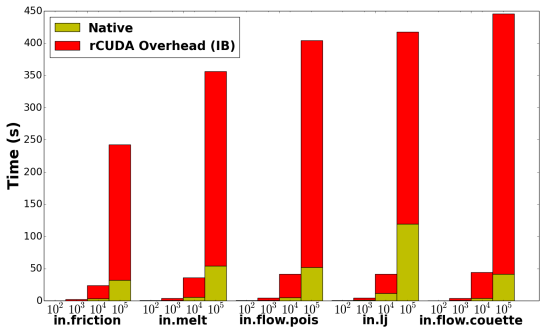
[Extended Abstract]

Pak Markthub markthub.p.aa@m.titech.ac.jp
Akihiro Nomura nomura.a.ac@m.titech.ac.jp
Satoshi Matsuoka matsu@is.titech.ac.jp

Tokyo Institute of Technology
2-12-1 Ookayama, Meguro-ku
Tokyo 152-8550 JAPAN

1. INTRODUCTION

rCUDA [1] is a state-of-the-art technology that enables CUDA applications running on one node to transparently use GPUs on other nodes. It works by intercepting all CUDA Runtime API calls, and send them to rCUDA servers running on remote nodes to execute those calls on those remote nodes' GPUs. With rCUDA, CUDA applications cannot tell the difference between remote and local GPUs in terms of functionality.



* **Major x-axis:** Input files
* **Minor x-axis:** Total iterations ("run" number)
* Running on two nodes connected via InfiniBand 4xFDR; each node has one NVIDIA Tesla K20c

Figure 1: rCUDA's overhead on LAMMPS

Despite having great functionality, some applications experience prohibitively longer execution time when using rCUDA. This is because rCUDA changes applications' GPU communication from intra-node to inter-node communication, and adds extra processing layers to every CUDA call. For example, LAMMPS exhibits increasing slow down as we increase its total iterations ("run" number), as shown in Figure 1.

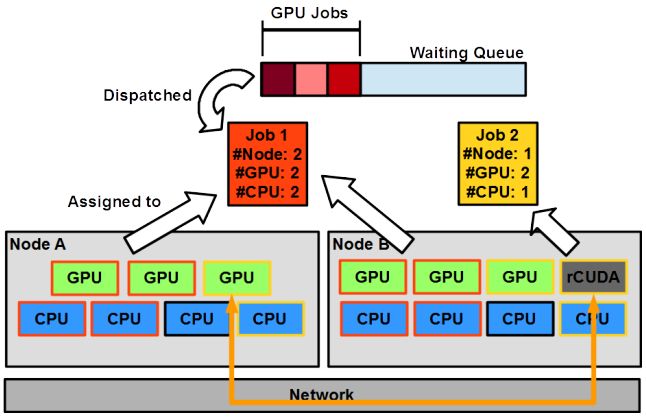


Figure 2: Example of using rCUDA to solve the idle-GPU scattering problem in a multi-GPU node-sharing system

In our previous work [2], we proposed a way to use rCUDA to solve the idle-GPU scattering problem on multi-GPU node-sharing systems. For example in the system shown in Figure 2, without rCUDA, Job 1 and Job 2 cannot concurrently use the system even though there are enough resources to serve both jobs. This kind of situation leads to low resource utilization in many heterogeneous systems. Our scheduling algorithm asks a job to use rCUDA when the assigned nodes do not have enough unoccupied GPUs. However, more local GPUs might become available while the job is running as some jobs sharing the same nodes finish processing. Unfortunately, rCUDA does not allow changing remote-GPU assignment at runtime.

2. MRCUDA

We propose mrCUDA, a middleware that enables transparent live-migrating GPU execution from remote to local GPUs. Since local GPU communication is always better than remote GPU communication, mrCUDA employs one-way one-time migration. The core concept of the migration is making the states and data on selected local GPUs the same as those on the corresponding remote GPUs. Basically, mrCUDA intercepts all CUDA Runtime API calls, **records** a small subset of those calls that are needed for

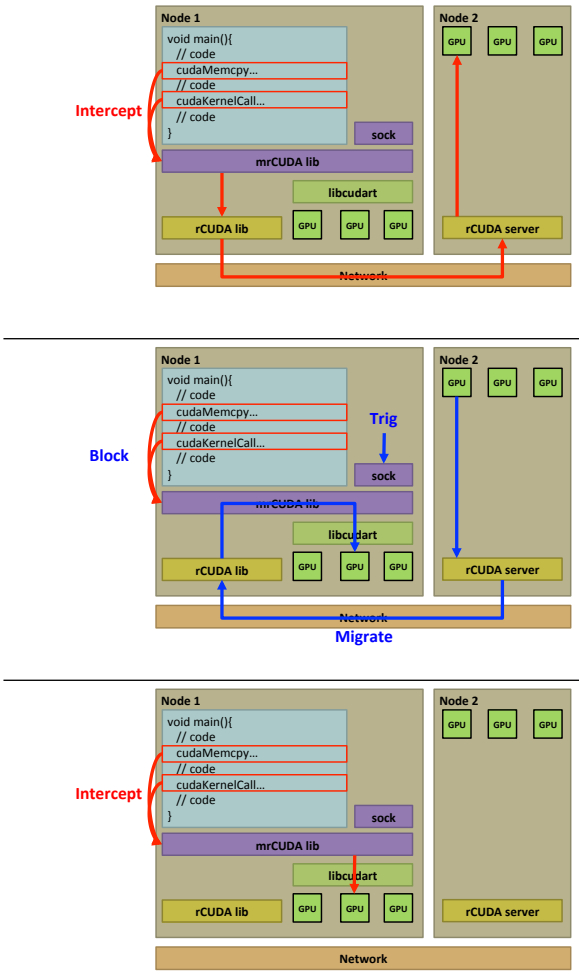
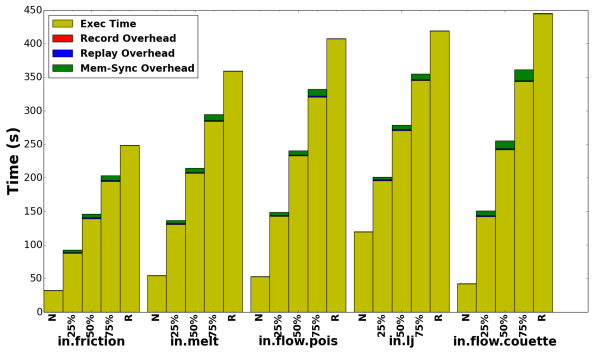


Figure 3: How mrCUDA works

synchronizing the states and data, and passes the calls to rCUDA library. Upon receiving a migration signal, mrCUDA blocks further calls, uses the recorded information to recreate the current states on the selected local GPU by **replaying** the recorded calls in order, and copies (**memsync**) data from the remote to the local GPU. After finishing the migration, mrCUDA forwards further calls to native CUDA library without further processing. Figure 3 shows how mrCUDA works.

For handling multi-GPU migration, mrCUDA employs one-process-one-GPU execution model. As an rCUDA server spawns one process for handling each remote GPU, virtual address spaces of remote GPUs can be overlapped. In order to simulate this and keep mrCUDA's overhead minimum, mrCUDA uses application's main process for the first migration and spawns an **mhelper** process, which acts as a local rCUDA server with very low overhead, for other migration.

3. EVALUATION



* **Major x-axis:** Input files
* **Minor x-axis:** 'N' for native CUDA; 'R' for rCUDA; 'x%' for migrating after finish 'x%' of total iterations
* **Total iterations:** 10^5
* Running on two nodes connected via InfiniBand 4xFDR; each node has one NVIDIA Tesla K20c

Figure 4: mrCUDA's overhead on LAMMPS

We evaluated our work by building a mathematical model for each type of mrCUDA's overhead (record, replay, memsync, and mhelper) and used micro-benchmark to validate the models. We also measured actual mrCUDA's overhead on LAMMPS. The result (Figure 4) shows that mrCUDA's overhead is very low, almost negligible, compared with rCUDA's overhead.

4. CONCLUSION

mrCUDA is a low-overhead middleware that enables CUDA processes to transparently live-migrate their GPU execution from remote to local GPUs. The main concept is making the states and data on selected local GPUs the same as those on the corresponding remote GPUs. According to mrCUDA's overhead models and the real measurement on LAMMPS, mrCUDA's overhead is very low, almost negligible, compared with rCUDA's overhead. Hence, mrCUDA allows multi-GPU node-sharing systems to serve more jobs while keeping the jobs' execution time minimum.

5. REFERENCES

[1] J. Duato, F. 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] P. Markthub, A. Nomura, and S. 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.