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

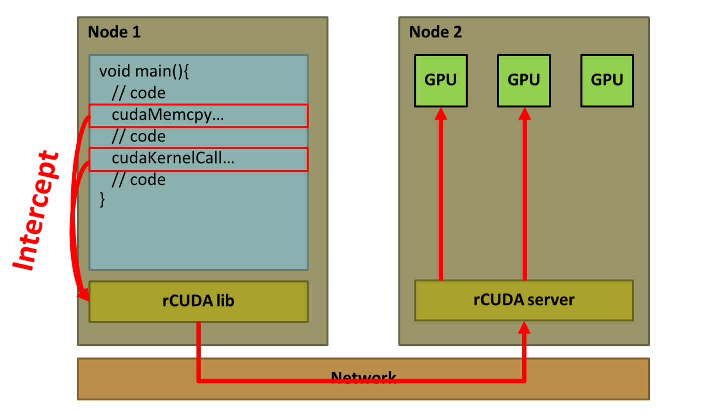
Pak Markthub, Akihiro Nomura and Satoshi Matsuoka

Tokyo Institute of Technology

Background

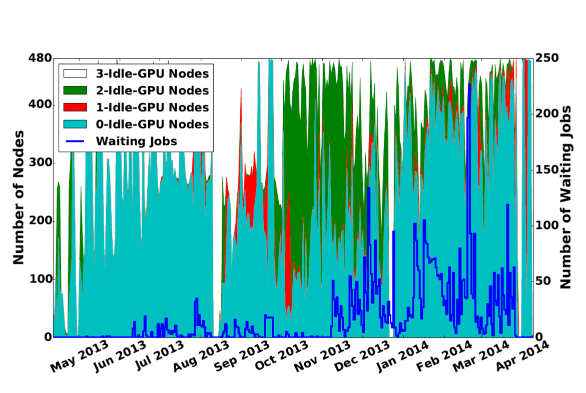
Multi-GPU node-sharing systems usually experience lower resource utilization than expected due to the idle-GPU scattering problem. In our previous work [1], we showed that by smartly letting some processes of some GPU jobs to use rCUDA (remote CUDA execution), this problem can be efficiently solved.

rCUDA [2] Remote CUDA Execution Technology

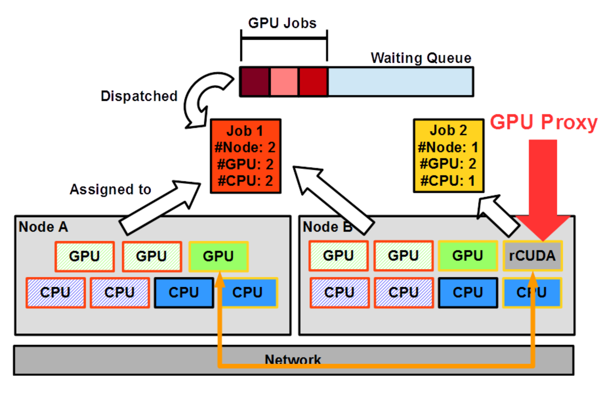


State-of-the-art remote CUDA execution technology that transparently enables CUDA applications to use remote GPUs.

Idle-GPU Scattering Problem solved by using rCUDA



The graph shows the number of nodes that had idle GPUs on TSUBAME2.5's G queue batch-queue system. There were idle GPUs even though there were GPU jobs waiting!!!



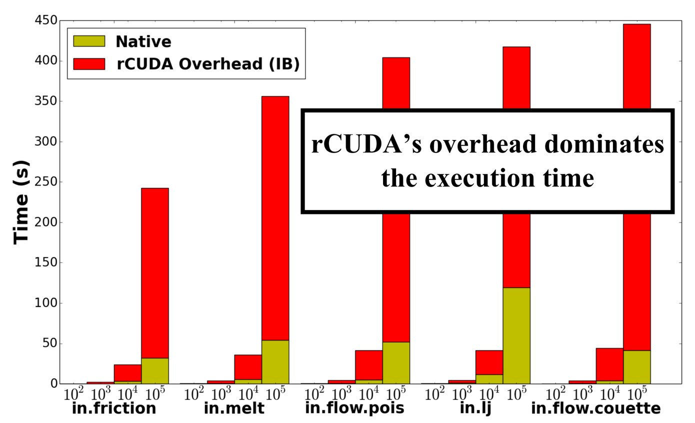
Without rCUDA, the system cannot concurrently serve Job 1 and Job 2 because Job 2 need two GPUs in one node, but the unoccupied GPUs are on different nodes.

Problems

Problem 1: By allowing jobs to use rCUDA, the systems may experience more network congestion because rCUDA changes GPU communication from intra-node to inter-node communication.

Problem 2: For some applications, especially HPC apps, the rCUDA's overhead may outweigh the apps' execution time.

rCUDA's Overhead on LAMMPS



rCUDA's overhead dominates the execution time

* **Major x-axis:** input file, **Minor x-axis:** total iterations

* **Network:** InfiniBand 4xFDR; **GPU:** NVIDIA Tesla K20c

rCUDA Overhead Model

$$time_{rcuda} = (latency_{rcuda} + latency_{net})num\_cuda\_call + \frac{datasize}{bw_{eff}} coef_{rcuda}$$

$latency_{rcuda} = 50.62\ \mu s$, $coef_{rcuda} = 1.03$

More than 5 times larger than the native CUDA's latency

Network's Characteristics   App's Characteristics

According to the equation, applications that frequently call CUDA Runtime APIs or transfer large amount of GPU data (GPU-communication-intensive applications) are likely to experience detrimental slow down when using rCUDA.

Proposal

Objective: Minimize rCUDA jobs' execution time and network congestion while allowing the system to serve more jobs by eliminating the idle-GPU scattering problem.

Motivation: Local GPUs may become available while rCUDA jobs are running since some jobs sharing the same nodes finish and release occupied GPUs; however, rCUDA does not allow jobs to change remote GPU assignment at runtime — local GPUs cannot be used by the rCUDA jobs.

Propose:

1. **mrCUDA**, a very low overhead middleware that enables transparent live-migration of CUDA execution from remote to local GPUs — the target users are CUDA applications and schedulers, which decide when each remote GPU of each job should be migrated.
2. **One-way one-time** remote-to-local GPU migration technique with very low overhead for CUDA applications — in term of performance, executing on local GPUs is always better than executing on remote GPUs.
3. **Multi-GPU migration** technique for rCUDA — this solves the overlapped address space problem happening when using multiple remote GPUs.

Conclusion

- mrCUDA is a middleware that allows applications to migrate CUDA execution from remote GPUs to local GPUs at runtime without having to modify the applications' source code.
- The main concept is making the states and active memory data on the local GPUs the same as those on the remote GPUs.
- The migration overhead is very small, almost negligible, compared to rCUDA's overhead.
- mrCUDA allows applications to enjoy early execution by using rCUDA to borrow idle GPUs from remote nodes while keeping rCUDA's overhead minimum by migrating the remote GPU execution to local GPUs when available, with very low cost.

mrCUDA

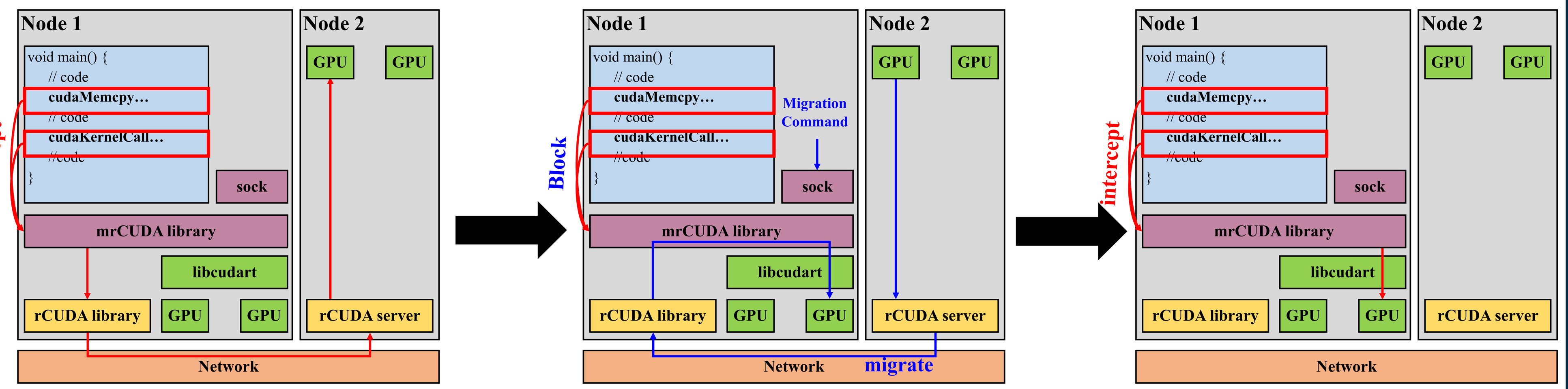
mrCUDA: A very low-overhead middleware that enables live-migration of CUDA execution from remote to local GPUs without any need for applications to modify or recompile their source code.

Main Concept: Making the states and data on the active memory regions of the selected local GPUs the same as those on the remote GPUs.

Highlighted Features:

1. **Transparent and Live Migration:** No need for applications to modify their source code or stop computation during the migration.
2. **One-Way One-Time Migration:** Since local is always better than remote in terms of performance, there is no need to support the reverse migration; doing so obviously increases overhead.
3. **Concurrent Use of Local and Remote GPUs:** mrCUDA handles each GPU migration separately; jobs can migrate as soon as a local GPU becomes available (no need to wait for enough number of local GPUs to become available).
4. **Low Overhead:** The migration overhead is very low, almost negligible, and mrCUDA can completely cut-off rCUDA's overhead after migration.

How mrCUDA Works?



First
Allow apps to use remote GPUs by sending every intercepted CUDA Runtime API calls to rCUDA.

Second
Upon receiving a migration command, block all calls and migrate remote GPUs' states and data to local GPUs

Last
Allow further intercepted calls to come in but send them to the migrating-to local GPUs instead

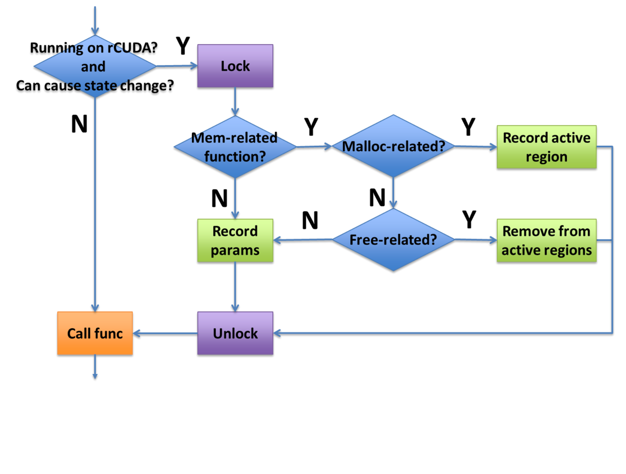
Migration Algorithm

Nukada et al [3]: calling the same order of CUDA Driver APIs on the same GPU always results in the same GPU's states and active memory regions.

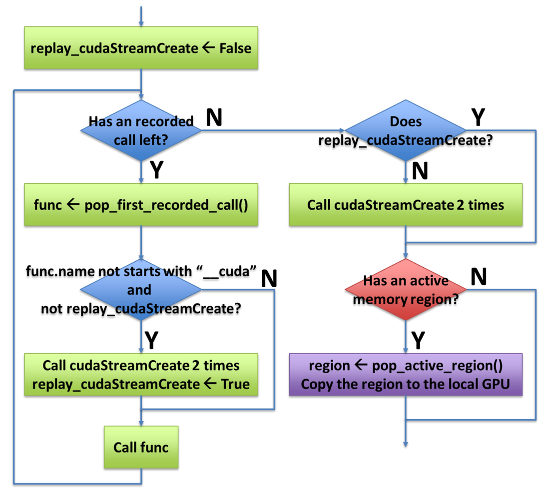
mrCUDA's migration algorithm:

1. Before migration, mrCUDA **records** intercepted CUDA Runtime API calls that can change the remote GPUs' states and active memory regions.
2. During the migration, mrCUDA **replays** the recorded calls in order to make the selected local GPUs' states and memory regions the same as the remote GPUs'.
3. mrCUDA copies (**mem-sync**) data on the remote GPUs' active memory regions to the corresponding local GPUs'.

Record Algorithm



Replay & Mem-Sync Algorithms



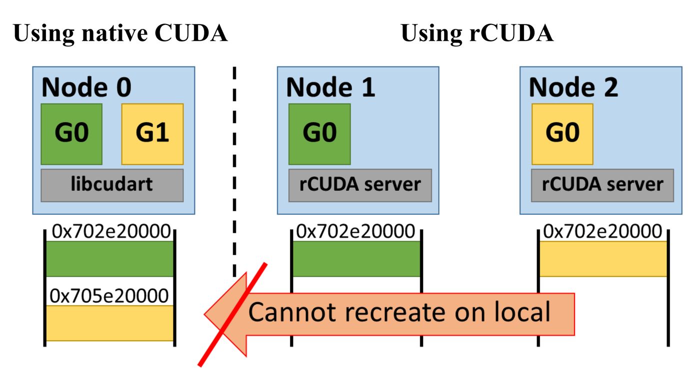
Multi-GPU Migration

Problem: Remote GPUs may have overlapped virtual address spaces; however, native CUDA prevents the overlapping of GPUs' virtual address spaces on the same process.

Proposed Solution:

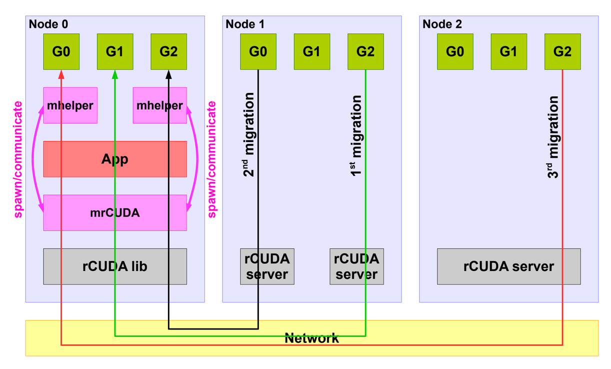
1. Use **one-process-one-GPU** execution model — it is possible to create overlapped address spaces of two local GPUs if each of them is assigned to different process.
2. **mhelper** — very low overhead local rCUDA server

Overlapped Address Spaces Problem



Using one process cannot simulate overlapped GPUs' address spaces but multiple processes can

Multi-GPU Migration with mhelper



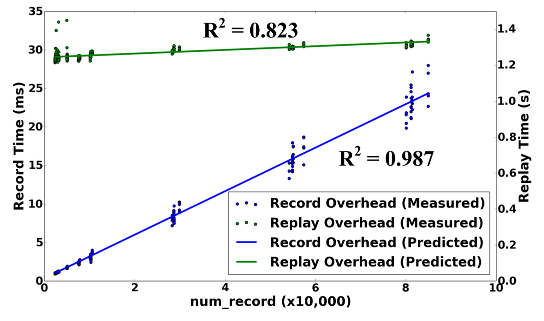
mrCUDA uses applications' main process to handle the first migration in order to minimize overhead

mrCUDA's Overhead Models

The models are mainly for schedulers since schedulers have to decide when each remote GPUs should be migrated to a local GPU.

Record & Replay Overhead Models

$$time_{record} = (record_{coef})(num\_record) + record_{const}$$
$$time_{replay} = (replay_{coef})(num\_record) + replay_{const}$$

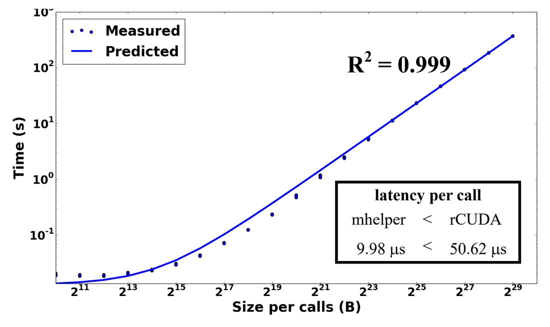


Record and replay overhead grow a lot slower than rCUDA's overhead.

- Record and replay overhead are proportional to the number of recorded calls while rCUDA's overhead depends on the number of intercepted calls (e.g. for LAMMPS, 1% of intercepted calls have to be recorded).

mhelper Overhead Model

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



mhelper has higher bandwidth and lower latency than rCUDA.

- The bandwidth does not depends on network quality
- The latency is about one-fifth.

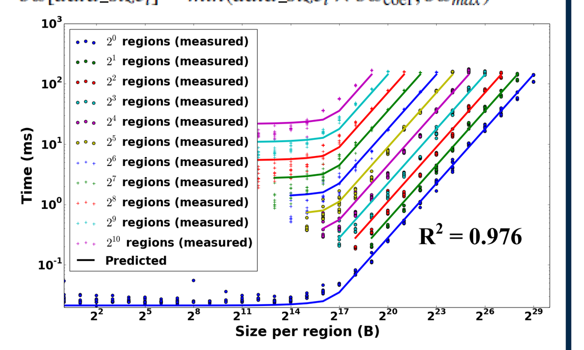
Mem-Sync Overhead Model

$$time_{memsync} = \underbrace{time_{rcuda}}_{\text{rCUDA's over-}} + \sum_{i}^{num\_active\_region} \left(\overbrace{\frac{data\_size_i}{bw[data\_size_i]}}^{\text{cudaMemcpy}} + \underbrace{(memsync_{coef} \times data\_size_i) + memsync_{const}}_{\text{internal management}} \right)$$

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

Transferring small number of large regions is better than transferring large number of small regions.

- For small region, the transfer time is quite constant.
- For large region, the transfer time is proportional to the region's size.

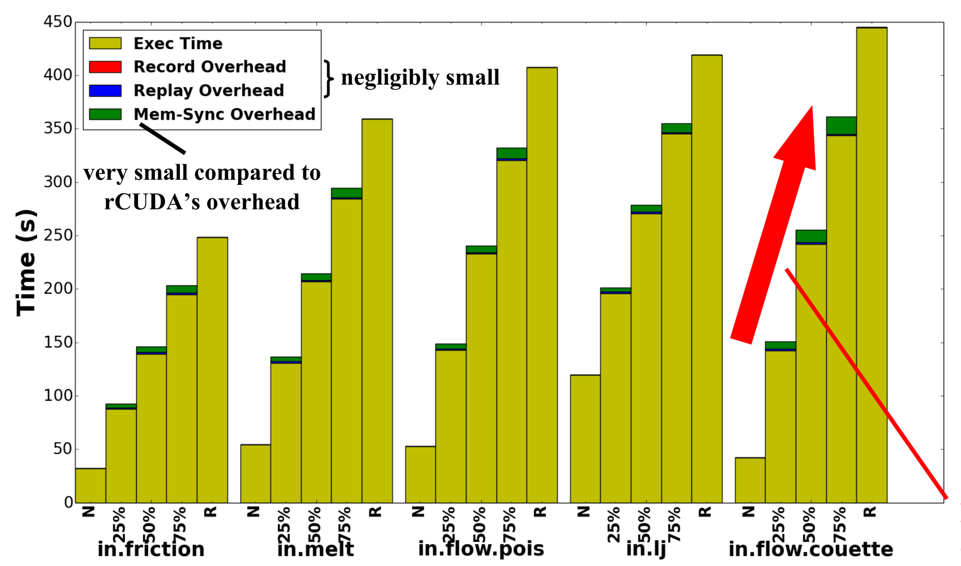


Values of the Models' Variables

Variables	Values
$record_{coef}$	2.825×10^{-7} s
$record_{const}$	3.437×10^{-4} s
$replay_{coef}$	1.031×10^{-6} s
$replay_{const}$	1.243 s
$mhelper_{coef\ d}$	6.871×10^{-10} s/B
$mhelper_{coef\ c}$	9.983×10^{-6} s
$mhelper_{const}$	2.934×10^{-3} s
$memsync_{coef}$	5.686×10^{-11} s/B
$memsync_{const}$	Almost 0
bw_{coef}	4.721×10^{4} s^{-1}
bw_{max}	4.779 GB/s

Case Studies

Migrating CUDA Execution of LAMMPS at Runtime with mrCUDA

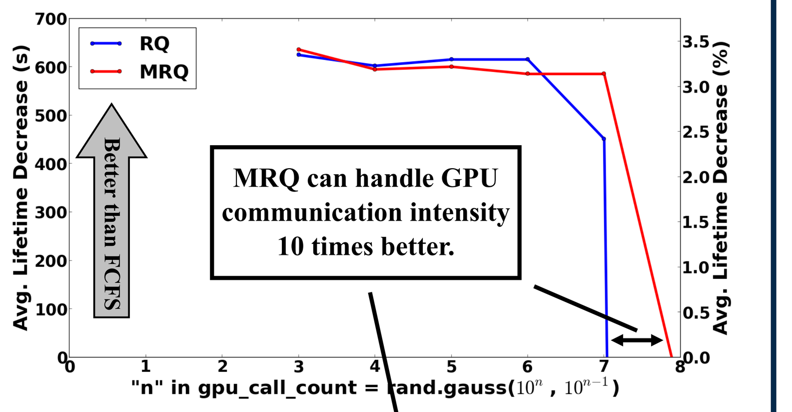


- The graph shows execution time of LAMMPS when:
  - **N:** used native CUDA; **R:** used rCUDA
  - **x%** migrated after LAMMPS finished x% of its total iterations
- Mem-Sync Overhead is less than 5% while Record and Replay Overhead is less than 0.01%.
- The performance of LAMMPS improves about 60%, 40%, and 20% compared with remote-only execution given that migration occurs after LAMMPS finished 25%, 50%, and 75% of its total iterations.
- Because mrCUDA employs one-way one-time migration with very low overhead, it has great potential in improving the performance of applications that temporarily need to use remote GPUs.

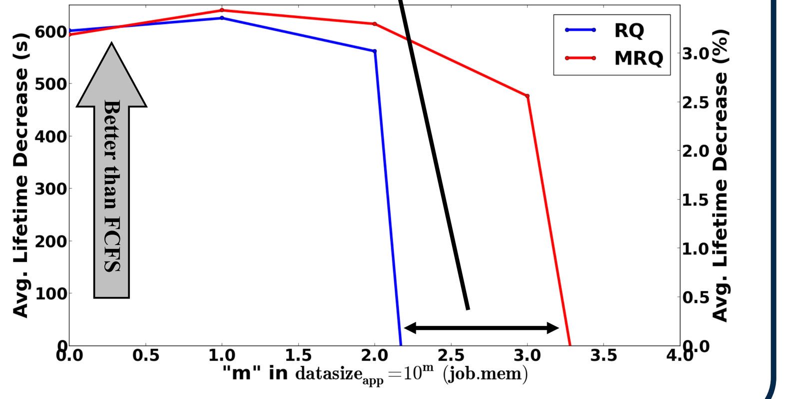
linearly increase due to rCUDA's overhead before the migration.

Improving Solution to Idle-GPU Scattering Problem with mrCUDA

- We show an advantage of using mrCUDA in improving our proposed solution [1] to the idle-GPU scattering problem.
- RQ is the scheduling algorithm that came from applying our solution to the first-come-first-serve (FCFS) scheduling algorithm. Basically, it uses FCFS as the base algorithm to find as many nodes that have sufficient GPUs to serve a waiting jobs as possible. If there are not enough nodes, RQ uses rCUDA to virtually consolidate GPUs into some nodes to make the number of runnable nodes sufficient.
- MRQ is the improved version of RQ that utilizes mrCUDA. Fundamentally, MRQ works the same as RQ but uses mrCUDA instead of rCUDA. Furthermore, upon a running job leaves the system, MRQ asks other running jobs sharing the same nodes as the leaving jobs to migrate remote GPU execution to the available local GPUs.
- The graphs show the average lifetime (wait time + execution time) decrease of jobs when using RQ and MRQ compared with using FCFS while varying GPU communication intensity (x-axis; refer to rCUDA Overhead Model for the variables' explanation).



MRQ can handle GPU communication intensity 10 times better.



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 (PDCAT2014), 2014 15th International Conference on, pages 105–112. IEEE, 2014.

[2] 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.

[3] 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.