2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)

C-GDR: High-Performance Container-aware GPUDirect MPI Communication Schemes on RDMA Networks

Jie Zhang, Xiaoyi Lu, Ching-Hsiang Chu, and Dhabaleswar K. (DK) Panda
Department of Computer Science and Engineering, The Ohio State University
{zhang.2794, lu.932, chu.368, panda.2}@osu.edu

***Abstract*—In recent years, GPU-based platforms have received significant success for parallel applications. In addition to highly optimized computation kernels on GPUs, the cost of data movement on GPU clusters plays critical roles in delivering high performance for end applications. Many recent studies have been proposed to optimize the performance of GPU- or CUDA-aware communication runtimes and these designs have been widely adopted in the emerging GPU-based applications. These studies mainly focus on improving the communication performance on native environments, i.e., physical machines, however GPU-based communication schemes on cloud environments are not well studied yet. This paper first investigates the performance characteristics of state-of-the-art GPU-based communication schemes on both native and container-based environments, which show a significant demand to design high-performance container-aware communication schemes in GPU-enabled runtimes to deliver near-native performance for end applications on clouds. Next, we propose the C-GDR approach to design high-performance Container-aware GPUDirect communication schemes on RDMA networks. C-GDR allows communication runtimes to successfully detect process locality, GPU residency, NUMA, architecture information, and communication pattern to enable intelligent and dynamic selection of the best communication and data movement schemes on GPU-enabled clouds. We have integrated C-GDR with the MVAPICH2 library. Our evaluations show that MVAPICH2 with C-GDR has clear performance benefits on container-based cloud environments, compared to default MVAPICH2-GDR and Open MPI. For instance, our proposed C-GDR can outperform default MVAPICH2-GDR schemes by up to 66% on micro-benchmarks and up to 26% on HPC applications over a container-based environment.**

I. Introduction

Graphics Processing Unit (GPU)-based platforms have been widely used in many modern HPC and Cloud Computing environments. The computational power of the GPU has changed the way for researchers and developers to highly parallelize their applications on such high-performance heterogeneous computing platforms. For instance, GPU has been becoming one of the most important driving factors of fast and scalable applications such as artificial intelligence, computation chemistry, and weather forecasting. To efficiently utilizing GPUs for parallel applications, in addition to design highly optimized computing kernels on GPUs, the performance of data movement operations on GPU clusters also makes significant impact.

However, the existence of GPUs significantly complicates the communication runtime designs on heterogeneous clusters. As shown in Figure 1, we classify the state-of-the-art data movement approaches that can be applied for moving GPU-resident data within a node as the following four schemes. **cudaMemcpy**: The default mechanism can be used to copy data from GPU memory to system (CPU) memory region, which could be shared between processes. The process can also copy the data from the shared system memory to the GPU memory. **GDRCOPY**: A data movement library based on GPUDirect Remote Direct Memory Access (RDMA) technology to provide low-latency data copy between GPU memory and system memory [28]. **cudaIPC**: NVIDIA has introduced Inter-Process Communication (IPC) in the Compute Unified Device Architecture (CUDA) for directly moving data between GPUs [23]. With this feature, the data movement can be occurred through Peripheral Component Interconnect express (PCIe) bus or NVLink without involving CPU and system memory. **GDR**: With GPUDirect RDMA (GDR) technology, third-party hardware such as InfiniBand Host Channel Adapter (HCA) can directly access GPU memory through PCIe bus. In this way, one can leverage GDR read and write operations to allow HCA performing the data movement between GPUs and bypass the CPU. The problem gets even more complicated on the cloud environment. For instance, we can deploy multiple containers on the same pysical or virtualized host with different placement schemes (intra-socket or inter-socket). Each data movement approach introduced above might have different applicable scenarios or bring different performance characteristics on such diverse container configurations compared to the cases on native environments.

A. Motivation

Many recent studies [7], [19], [27] have been proposed in the community to optimize the performance of GPU-aware (or CUDA-aware) communication runtimes to take advantage of the novel features of GPU and RDMA capable networks and these designs have been well received in the HPC community. However, most of these studies focus on improving the communication performance with GPUs on native environments (i.e., physical machines). With more and more adoptions of GPUs in cloud computing environments, especially container-based cloud platforms, it is desired to investigate whether these existing communication schemes can still perform well on cloud-based platforms (e.g., Docker). Unfortunately, we find GPU-based communication schemes are not well studied yet in container-based cloud computing environments. The complexity of designing efficient GPU-based communication schemes on clouds is significantly increased under the

*This research is supported in part by NSF grants #IIS-1636846, #CNS-1513120, #CCF-1565414, #ACI-1664137, and #CCF-1822987.

1530-2075/19/$31.00 ©2019 IEEE
DOI 10.1109/IPDPS.2019.00034

container environments on such heterogeneous systems. The challenges of supporting efficient GPU-based communication under the container environments can be broadly summarized as follows.

- What are the performance characteristics of applying existing native GPU-based communication schemes in the container environments?
- How to intelligently and transparently select an optimal communication scheme for end applications to adapt with different communication patterns, (e.g., latency- or bandwidth-sensitive), different container deployment cases (e.g., intra-socket or inter-socket), different message sizes, and hardware architectures?
- How to design an adaptive and modular approach to dynamically use the optimal GPU-based communication schemes in the communication runtime such as Message Passing Interface (MPI) library for container environments?

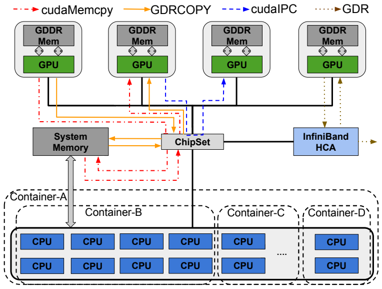


Fig. 1: Data Movement Strategies between GPUs in Container Environments within a node

B. Contributions

The selection of the optimal GPU-based communication schemes in clouds is not a trivial problem. Unfortunately, there is a dearth of intelligent container-aware communication scheduling mechanisms to provide adaptive and optimal GPU-based communication schemes in clouds. As presented in Table I, Intel MPI and MVAPICH2 support CPU-based (Host memory) communication in native environment, while MVPIACH2-Virt supports CPU-based (Host memory) communication in cloud environments (VM/Container/Nested Virtualization). Open MPI, IBM Spectrum MPI, and MVAPICH2-GDR support both CPU-based and GPU-based (Device memory with GPUDirect RDMA) communication in native environment. Container-aware GPU-to-GPU communication schemes do not exist even though it is highly demanded with the emerging trend of HPC clouds. To the best of our knowlegde, C-GDR proposed in this paper is the first work to fill this gap and proven to work efficiently in the container-based cloud environments.

TABLE I: **Existing MPI stacks Comparison on Usage**

MPI	CPU	GPU(Direct)	Native	Cloud
Intel MPI [1]	√		√	
MVAPICH2 [2]	√		√	
Open MPI [5]	√	√	√	
IBM Spectrum MPI [15]	√	√	√	
MVAPICH2-GDR [22]	√	√	√	
MVAPCH2-Virt [17], [18], [30]	√			√
C-GDR	√	√		√

This paper first investigates the performance characteristics of state-of-the-art GPU-based communication schemes on container-based environments. Our studies expose a significant demand on designing high-performance container-aware communication schemes in GPU-enabled runtimes, e.g., GPU-aware MPI, to deliver the optimal communication performance for applications on clouds. Next, we propose the C-GDR approach to design high-performance container-aware GPUDirect communication schemes on RDMA networks. C-GDR allows communication runtimes to detect process locality, GPU residency, non-uniform memory access (NUMA), architecture information, and communication pattern. Hence, C-GDR enables communication libraries to intelligently select the best communication paths during runtime to speed up data communication between GPUs on container-based cloud environments. We have integrated C-GDR designs into the MVAPICH2 MPI library. We conduct extensive performance evaluations of MVAPICH2 with C-GDR schemes on native and cloud environments. The results show that our proposed C-GDR designs can outperform default MVAPICH2-GDR schemes by up to 66% with MPI micro-benchmarks on a containter-based environment. Moreover, the evaluation with multiple important HPC applications such as HOOMD-blue [12] and Jacobi computation shows that our proposed design can achieve up to 26% performance benefit compared to default schemes running on 16 Docker instances. MVAPICH2 with C-GDR also presents significant performance benefit compared to Open MPI. Overall, C-GDR brings clear benefit to HPC workloads on container-based GPU systems without changing users' usage behavior. To summary, this paper makes the following key contributions:

- Provide a detailed performance study of the existing GPU-based communication schemes on container-based cloud environments
- Design container-aware GPUDirect communication schemes on RDMA networks to enable intelligent communication between GPUs
- Implement the proposed schemes into the widely used MVAPICH2 MPI library to support container-aware features on GPU-enabled systems
- Conduct comprehensive performance evaluations of the proposed container-aware designs with benchmarks and applications on a real Docker-based cloud environment

The rest of this paper is organized as follows. Section II provides background knowledge related to this work. Section III presents the study and analysis of the current GPU-based communication schemes on the cloud environment, followed by the proposed designs in Section IV. Performance evaluation is demonstrated in Section V. Finally, we present related work in the literature in Section VI and conclude the work in Section VII.

II. Background

In this section, we provide the necessary background knowledge related to this work.

A. Container-based Virtualization

Virtualization provides abstractions of multiple virtual resources by utilizing an intermediate software layer on top of the underlying system. *Hypervisor-based virtualization* is one of the most popular virtualization techniques, such as Xen, VMware, KVM. Virtualization, in its most common form, consists of a hypervisor on top of a host operating system that provides a full abstraction of virtual machine (VM). *Container-based virtualization* is a lightweight alternative to the hypervisor-based virtualization. The host kernel allows the execution of several isolated user-space instances run a different software stack (e.g., system libraries, services, applications). Container-based virtualization provides self-contained execution environments, effectively isolating applications that rely on the same kernel in the Linux operating system, but it does not introduce a layer of virtual hardware. There are two core mature Linux technologies to build containers. First, namespace isolation isolates a group of processes at various levels: networking, filesystem, users, process identifiers, etc. Second, cgroups (control groups) groups processes and limits their resources usage. Several container-based solutions have been developed, such as Docker, LXC, and Googles lmctfy. In this paper, we deploy Docker, which is a popular open-source platform for building and running containers and offers several important features, including portable deployment across machines, versioning, reuse of container image and a searchable public registry for images.

B. GPUDirect Technology and CUDA-Aware MPI

NVIDIA GPUDirect technology [3] is a set of features to enable efficient communication among NVIDIA GPUs and other devices. It significantly enhances communications performance on GPU clusters. GPUDirect, through RDMA feature, allows third-party PCIe devices with direct access to GPU memory. This feature is called GPUDirect RDMA (GDR) and is currently supported by Mellanox InfiniBand network host channel adapters (HCAs). This provides a path for moving data to/from GPU device memory over an InfiniBand network that completely bypasses the host CPU and its memory. Through GDR technology, several MPI implementations such as Open MPI, MVAPICH2, Spectrum MPI and Cray MPI provide "CUDA awareness". These CUDA-Aware MPI runtimes can deliver both high performance and productivity.

III. Performance Characteristics of GPU Communication Schemes on Container-based Environments

Choosing the optimal data movement scheme for a given message is a challenging task in the native environment. It becomes even more complicated in the container-based cloud environment because various configurations of container deployment can be used in the cloud. In this section, we conduct experiments to understand the performance characteristics of the GPU communication schemes on native and cloud environments. Based on the performance study, we provide design guidance to optimize GPU communication on clouds.

A. GPU Communication Schemes on Cloud

Communication schemes on HPC systems have been substantially studied and optimized in the last few decades. However, it has been significantly changed since GPU joins the HPC community. Specifically, GPU-to-GPU communication can be roughly categorized into Intra-node and Inter-node cases. Intra-node refers to the case that two or more GPU devices are equipped onto the same physical node. The communication happens from one GPU buffer to another GPU buffer within the node, while Inter-node case means the GPU-to-GPU communication goes across different physical nodes via the network. Table II describes existing GPU-to-GPU data movement mechanisms in details.

B. Performance Study of GPU Communication on Cloud

In this section, we conduct the experiments to understand the performance characteristics of GPU-to-GPU communication with different data movement schemes in the cloud environment.

As presented in Figure 2, we observe the clear performance differences across different data movement schemes. This observation implies the necessity and significance of studying the GPU-to-GPU communication performance in the cloud environment. Moreover, this paper focuses on optimizing intra-node MPI communication across GPUs in containers. Thus, the performance of inter-node communication will not be affected, which is also verified in Section V-B. Please also note that some data movement strategies, like GDR-loopback, are typically not enabled for the MPI runtime in the native environment. This is because the MPI processes with the same hostname are automatically identified as co-located, intra-node GPU-to-GPU communication schemes will be applied. Thus, in order to make the lines in Figure 2 be more focused and more precise for container environments, we do not include native performance here. The native performance will be shown and discussed in Section V to provide a comprehensive evaluation.

1) Latency-sensitive Benchmark: In the latency-sensitive benchmark, i.e., *osu_latency* in OSU Micro-Benchmarks (OMB), it uses blocking communication interfaces like MPI_Send and MPI_Recv to ensure the completion of each communication operation.

From Figure 2(a)-2(b), we can see that GDRCOPY in the container environment brings the lowest latency for the small messages (1-16 bytes), while GDR-loopback achieves the optimal performance for the medium messages (16-16K bytes), then cudaIPC outperforms other schemes for the large messages. Because of the high latency of GDRCOPY for large messages, we ignore and remove it from Figure 2(b) in order to show clear performance comparison among other schemes. Our observation indicates that there is no one particular data

TABLE II: **Existing GPU-to-GPU Data Movement Mechanisms**

	Data Movement	Description
Intra-node	cudaIPC	CUDA Inter-Process Communication (IPC) facilitates direct copy of data between GPU device buffers allocated by different processes on the same node, which bypasses the host memory and thus eliminates the data staging overhead (from GPU device memory to the host memory). As shown in Figure 1, this is only applicable in the intra-node case.
	cudaMemcpy	Whenever cudaIPC is not available or does not provide good performance, an explicit data staging scheme through the shared memory region on the host is unavoidable. cudaMemcpy is one of the data staging schemes, which copies data between GPU device memory and host memory by specifying the direction of the copy.
	GDRCOPY	GDRCOPY is another data staging scheme, which provides a low-latency GPU memory copy operation based on NVIDIA GPUDirect RDMA technology. Basically, it offers the capability to create user-space mappings of GPU memory via one PCIe BAR (Base Address Register) of the GPU. The user-space mappings can then be manipulated as if it is the plain host memory [28], as shown in Figure 1.
Inter-node	GDR	As introduced in Section II-B, GDR technology enables a path for moving data to/from GPU device memory over an InfiniBand Host Channel Adaptor (HCA) that completely bypasses the host CPU and its memory. If the GDR feature is available, HCA can directly read the source data on one GPUs memory and write to another GPUs memory. However, due to the performance concern, many communication runtimes have designs to stage the GPU-resident data through the host memory, where an advanced host-based pipeline design is common [8], [22], [27]. The same staging schemes, as described in the Intra-node case can be applied here as well.
	GDR-loopback	In the container-based cloud environment, container deployment is flexible. Multiple containers could be deployed on the same node. However, they do not recognize each other, even though the communicating peers are within the same node physically. The communication in this case actually operates in the loopback manner with GDR.

movement strategy that can benefit for all message sizes. It is critical to carefully organize the different data movement strategies according to the varying message size.

Moreover, we notice that the shared memory-based intra-node GPU-to-GPU data movement schemes, such as GDRCOPY and cudaIPC, cannot be applied in the co-located containers scenario due to lack of locality-aware support. The only one scheme they can utilize is GDR-loopback, even though the communicating peers are physically co-located.

2) Bandwidth-sensitive Benchmark: Here, the bandwidth test, i.e., *osu_bw* in OMB, is performed between two processes within a node. The test is issuing multiple non-blocking communications like MPI_Isend and MPI_Irecv calls to saturate the available bandwidth of IB HCA. As shown in Figure 2(c)-2(d), we can observe that the performance is very different for different data movement schemes in the container. As we have seen in the latency tests. This again implies the different communication paths and data movement strategies need to be carefully selected for different message sizes in the container environment.

In order to achieve optimal performance, we also notice that the switch points to the optimal schemes are different between latency-sensitive tests and bandwidth-sensitive tests on the same physical configuration. For instance, in Figure 2(b), in order to deliver the lowest latency, GDR-loopback is switched to cudaIPC at around 16K bytes message size, while it is approximately 512K bytes for the bandwidth test in Figure 2(d).

C. Analysis and Design Principles for Optimal GPU Communication on Cloud

The major difference between the container and native environments is the capability to detect the physical location of CPUs and GPUs. In the container environment, the communication between the co-located containers is always treated as the inter-node case due to the lack of locality-aware capability in the current GPU-aware communication runtimes. Therefore, GDR-loopback communication path will always be used, and the GPU communication cannot leverage other communication schemes such as GDRCOPY, cudaMemcpy, and cudaIPC. The experimental results in Figure 2 provide following insights:

1) No one particular data movement scheme can deliver optimal communication performance over all the different message sizes.

2) In order to deliver optimal communication performance, it is necessary to appropriately coordinate the different data movement strategies.

3) For the co-located container case, the shared memory based intra-node data movement schemes cannot be applied currently, even though they perform the best on some message sizes. Therefore, it is required to have locality-aware support to enable the selection of the optimal communication channel.

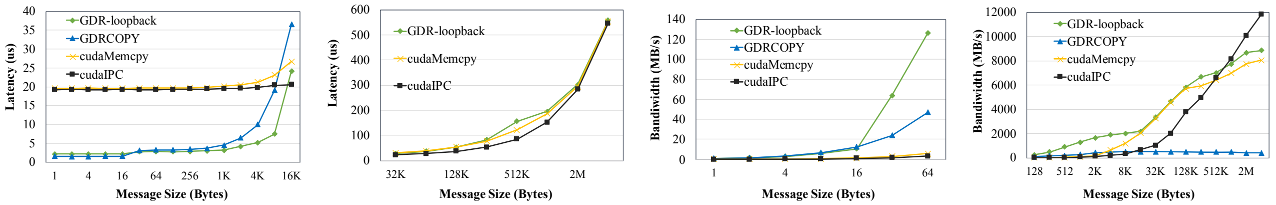
4) As comparing Figure 2(b) with Figure 2(d), we can find that the switch points among different optimal schemes are different for latency-intensive and bandwidth-intensive tests.

Based on these insights, the design principles of optimal GPU-based communication schemes in the container-based environment can be summarized as follows:

- A **locality-aware support** is required to allow runtimes to enable the intra-node communication paths such as GDRCOPY, cudaMemcpy, and cudaIPC if applicable
- An **intelligent communication path scheduling mechanism** is needed to allow runtimes to dynamically select the optimal communication path and data movement scheme for the given message size
- A real-time **workload characterization tracing mechanism**, to allow runtimes to be aware of the latency-sensitive or bandwidth-sensitive communication workloads, is needed to dynamically switch the communication path during application runtime

IV. Proposed Design of C-GDR in MVAPICH2

In this section, we take MVAPICH2, a popular MPI library as a case study to provide the high performance container-aware GPUDirect communication schemes on RDMA networks, based on the insights and guidance what we have explored in Section III for the container-based HPC cloud environment. Figure 3 presents an overview of our proposed C-GDR designs in MVAPICH2. As we can see, a node is equipped with one multi-core processor, one HCA, and multiple GPU devices. Accordingly, multiple containers are



(a) 1 - 16K Bytes

(b) 32K - 4M Bytes

(c) 1 - 64 Bytes

(d) 128 - 4M Bytes

Fig. 2: Latency and Bandwidth comparison of data movement strategies on Docker container environment within a node. The testbed cloud is described in Section V. MVAPICH2-GDR 2.3a, which is a GPU-aware MPI library, and OSU Micro-Benchmark [6] suite are used. To evaluate the performance of cudaMemcpy, GDRCOPY, and cudaIPC, one Docker container is deployed, which equips with a 4-core CPU and two GPUs. To evaluate the performance of GDR-loopback, two Docker containers are deployed on the same host, and each container is allocated with a 4-core CPU and one dedicated GPU device. The HCA is shared by the two containers. In this deployment, each container launches one MPI process and exchanges the data on GPU with each other. The experiments are conducted over ten runs with 1,000 iterations of each run.

deployed to fully take advantage of these powerful computing resources.

In order to support high-performance GPUDirect communication schemes with RDMA network on container-based HPC cloud environments, three new modules are introduced into the MPI library, which includes a GPU Locality-aware Detection module, a Workload Characterization Tracing module, and a Communication Coordinator (Scheduling) module. As introduced in Section III, there exist multiple different communication paths on a GPU-based platform. In the bare-metal environment, MVAPICH2 library uses cudaMemcpy, cudaIPC, and GDRCOPY communication channels for intra-node GPU-to-GPU (i.e., device to device) message transfer while utilizing GDR and Host-based Pipeline channels for inter-node GPU to GPU communication, as presented in the bottom layer of Figure 3.

In the container-based HPC cloud environment, the communication channels and the communication channel coordination can work in the same way as the ones in the bare-metal environment. However, the GPU-to-GPU communication between two co-resident containers will be considered as the inter-node communication (GDR-loopback), due to the lack of GPU locality-aware support. Therefore, the GPU Locality-aware Detection module can help MPI runtime and the applications running on top of it to dynamically and transparently detect the MPI processes in the co-resident containers. With this module, we have the opportunities to reschedule the MPI-based communications between co-resident containters with GPUs going over more efficient communication channels. Moreover, there can be multiple different container deployment schemes on NUMA architecture. Accordingly, the communication between co-resident GPUs can be significantly affected by the varying container deployments from both functionality and performance perspectives. The NUMA-aware Support module is responsible for providing NUMA information to MPI processes. With the aid of the NUMA-aware Support module, the source process is able to figure out whether the destination process is running on the same socket or the different ones before the real communication takes place. The Communication Scheduling module will leverage the GPU locality information and NUMA information generated by GPU Locality-aware Detection module and NUMA-aware Support module, respectively to reschedule the communication going through the appropriate and optimal underlying channel, based on the communication characteristics, which we have explored on the container-based HPC cloud environment in Section III.

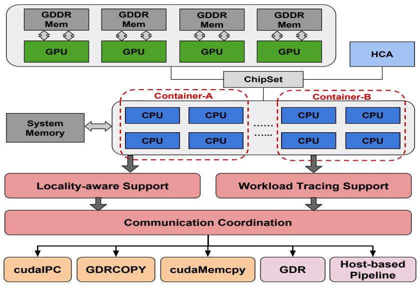


Fig. 3: Overview of C-GDR in MVAPICH2

A. *GPU Locality-aware Detection*

The GPU Locality-aware Detection module is responsible for dynamically and transparently detecting the location information of communication processes between the co-resident GPUs. Since the shared memory segments, semaphores and message queues can be shared across multiple Docker containers by sharing the IPC namespace when launching containers. We allocate such shared memory segments on each physical node and create a GPU Locality-aware List on it. Each MPI process associating with one GPU in co-resident containers writes its locality information into this shared list structure according to its global rank. After a synchronization, it can be guaranteed that the locality information of all local MPI processes has been collected up and stored in the GPU Locality-aware List. If the user launches two MPI processes to carry out GPU-to-GPU communication, the GPU Locality-aware Detection module is able to quickly identify whether it is the co-resident GPUs communication by checking the locality information on the list according to their global MPI ranks.

Figure 4 illustrates an example of launching a 6-process MPI job. Two containers (Container-A and Container-B) are deployed on the same host, and each container is equipped with one GPU device. There is one MPI process in each container, and the other four MPI processes are running on another host. In the GPU Locality-aware Detection module, the two MPI processes (rank 0 and rank 1) write their identifications on positions 0 and 1 on the GPU Locality-aware List, respectively. There will be ‘0’ in the other four positions on the list as those four MPI processes are not running on the same host. If MPI processes with rank 0 and 1 are going to execute GPU-to-GPU message transfers, the fact of co-residence of the GPU devices can be efficiently identified by checking the corresponding positions in the GPU Locality-aware List. The number of local processes on the host can be acquired by traversing and counting the positions with the written identifications. Their local ordering will still be maintained by their positions in the list. It is costly to frequently access the GPU Locality-aware Detection module for each message transfer. Each MPI process, therefore, scans the locality results generated by GPU Locality-aware Detection module and maintains its local copy for all the peer processes. When considering process migration or other scenarios which might cause the locality to change, the proposed Locality-aware Detection module need to be re-triggered to update the locality information. Take the migration for instance, the communication channel will be suspended before migration to guarantee that there is no on-the-fly messages during migration [31]. Once the migration procedure finishes, the locality information of all processes needs to be updated in order to resume the communication onward.

The fixed number of bytes is used to tag each MPI process in GPU Locality-aware List. This guarantees that multiple processes belonging to co-resident containers are able to write their locality information on their corresponding positions concurrently without introducing lock/unlock operations. This approach reduces the overhead of the locality detection procedure. For instance, an MPI job with one million processes only occupies $T \times 1M$ bytes memory space for the list, assuming T as the fixed number of bytes for tagging each MPI process. The space complexity is $\mathcal{O}(N)$, where N is the number of MPI processes. It thus brings good scalability on the container-based HPC cloud environment.

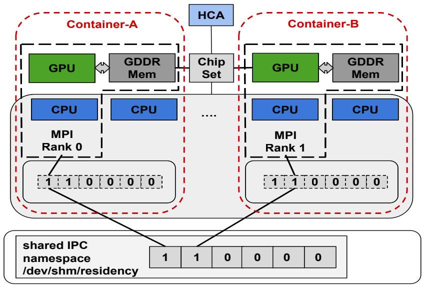


Fig. 4: GPU Locality-aware Detection Module in C-GDR

In addition, there can be different placement schemes to deploy the containers on a NUMA architecture. The communication performance will also be affected by the placements accordingly. The GPU Locality-aware Detection module can also be used to provide NUMA information of peer MPI processes for the following Communication Scheduling module, so that some performance bottlenecks and functional limitations can be avoided during the communication rescheduling phase. We assume that the administrators or cloud deployment stack can specify the CPU cores to launch the containers and different containers will not be launched with the same sets of the cores to eliminate the unnecessary performance interference. When the Docker engine is invoked to launch a container with the specified core IDs, it forms a tuple with the container name, the corresponding core IDs, and the associated NUMA node ID, e.g., [Container, Cores, and Sockets]. Then such tuple is visible to each MPI process in the co-resident containers through shared IPC namespace. If the destination process is identified as co-resident through GPU Locality-aware Detection module, NUMA-aware Support is triggered to further compare the NUMA node IDs of the destination MPI processes with its own ID to identify the relative NUMA information. More specifically, it can be identified that whether the message transfer will be across a socket or not.

B. Workload Characterization Tracing

In Section III, we observe that latency and bandwidth benchmarks have different switch points of their best performance numbers for different communication channels in the container environment. This implies that it is required to dynamically control the channel switch point in the runtime in order to deliver the optimal communication performance for different types of workloads. C-GDR provides a Workload Characterization Tracing module, which is responsible for keeping track of the communication patterns. The Workload Characterization Trace module can persistently update the respective counters of MPI_Send/Recv and MPI_Isend/Irecv to decide the workload is latency-intensive or bandwidth-intensive. For instance, Figure 5 shows Rank 0’s structure of Workload Characterization Tracing Module. When the process with rank 0 needs to send/receive a message to/from the process with rank 3, it first checks the locality information of the destination process (Rank 3) by its locality detection module. If the destination process is detected as a co-located process, it updates the Send/Recv counter if the communication is in the blocking mode; otherwise, the Isend/Irecv counter is updated for the non-blocking communication. Upon one of these two counters exceeds the predefined threshold, like Send/Recv counter, we treat the workload as the latency-intensive workload, and the communication channel switch point can be adaptively updated to achieve the optimal latency performance. The threshold could be different with different underlying configurations. The user can tune it through the runtime parameter. As the workload characterization tracing results can be quickly updated and easily maintained in the performance critical path, it does not incur severe performance overhead.

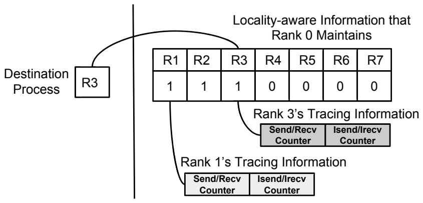


Fig. 5: Workload Characterization Tracing Module in C-GDR

C. Communication Scheduling

The Communication Scheduling module reschedules the message to go through the appropriate communication channel in order to deliver the optimal GPU-to-GPU communication performance in container-based cloud environments. In this module, there are four function units including GPU Locality Loader, Workload Characterization Parser, Message Attribute Parser, and Communication Scheduler. GPU Locality Loader reads the locality information including the NUMA placement of the destination process from the GPU Locality-aware Detection module. Workload Characterization Parser parses the tracing results from the Workload Characterization Tracing module. Message Attribute Parser obtains the attributes of the message, such as message type and message size. For a communication request to a specific destination process, Communication Scheduler selects the appropriate communication channel based on all the information in the above three aspects. By utilizing the Locality-aware Detection module, the communications between the co-located processes are able to use the high-performance intra-node communication channels, such as GDRCOPY, cudaMemcpy, and cudaIPC for different message sizes. From our experiments, we observe that GDR-loopback scheme can deliver better performance than those shared memory based intra-node data movement schemes for some message sizes. In this scenario, the Communication Scheduling module can also select the GDR-loopback communication channel for the specific range of message size, even though the communication processes are detected as the co-located case. If the workload characterization tracing result indicate that one of the counters exceeds the predefined threshold, the Communication Scheduling module will select the communication channel based on the comparison result between the message size and the channel switch point. For instance, once the Isend/Irecv counter exceeds the threshold, the workload is identified as the bandwidth-intensive workload and the switch point from GDR-loopback to cudaIPC will be updated from 16KB to 512KB. After that, messages less than 512KB will still go through the GDR-loopback channel.

Through our experiments, we summarize the optimal scheduling policy in container-based cloud environments in Table III. For the latency-sensitive workloads, GDRCOPY is selected for GPU-to-GPU communication with 1-16 bytes message size. For the message size which is larger than 16 bytes and less than 16K bytes, GDR-loopback is selected instead of the intra-node data movement schemes. Then cudaIPC is utilized for the large messages transfer. As there is a different performance characterization for the bandwidth-sensitive workloads, GDR-loopback is chosen as the optimal GPU-to-GPU communication scheme with the message size range from 16 bytes to 512K bytes.

TABLE III: Best schemes discovered for given message ranges for latency-sensitive and bandwidth-sensitive benchmarks

	Latency-sensitive	**Bandwidth-sensitive**
msg $\leq$ 16B	GDRCOPY	
16B $<$ msg $<$ 16KB	GDR-loopback	GDR-loopback
16KB $\leq$ msg $\leq$ 512KB	cudaIPC	
msg $>$ 512KB		cudaIPC

V. Performance Evaluation

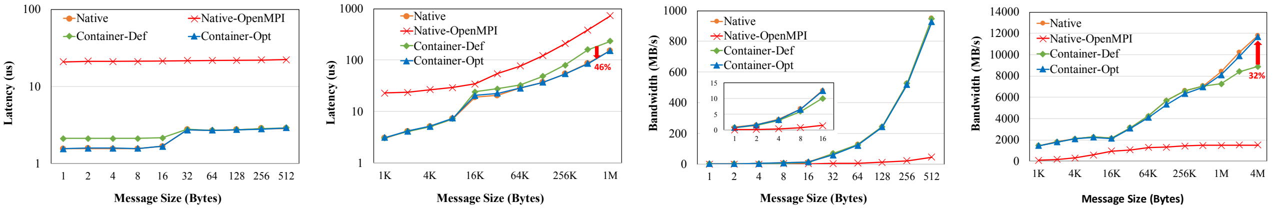
A. Experimental Testbed

Our testbed consists of eight physical nodes. Each node has a dual-socket 28-core 2.4 GHz Intel Xeon E5-2680 (Broadwell) processor with 128 GB main memory and is equipped with one Mellanox ConnectX-4 EDR (100 Gbps) HCA and one NVIDIA K-80 GK210GL GPU. Please note that each K-80 is a dual-GPU card. The two GPU cards and the HCA are connected to the same socket. We deploy 16 Docker containers using NVIDIA Docker 1.01 [21] on these eight physical nodes to make the images agnostic of the NVIDIA driver. Each node has two containers, which are pinned to the same socket. Each container is equipped with one GPU card. On both physical nodes and containers, we use CentOS Linux 7 as the operating system. In addition, we use MLNX_OFED_LINUX-3.4-2.0.0, NVIDIA Driver 384.81 and CUDA Toolkit 8.0. 'Native' denotes the performance of the running process in the bare-metal environment with MVAPICH2-GDR. 'Container-Def' denotes the performance of the running process in container environment binding with the same physical core and GPU device like the ones in the 'Native' scheme. 'Container-Opt' denotes the corresponding performance in the container case with our proposed optimizations.

B. MPI Level Point-to-Point Micro-benchmarks

In this subsection, we evaluate the influence of our proposed designs on point to point communication performance with OSU Micro-Benchmark suite. We focus on the performance evaluation of inter-node and single node inter-container case in this section.

Figures 6(a) and 6(c) show the evaluation results of latency and bandwidth performance results for small messages. We can observe that both Native and proposed container (Container-Opt) schemes perform better than the default container (Container-Def) scheme for 1-16 bytes messages. The performance benefit comes from choosing GDRCOPY as the optimal communication channel. For example, the latencies of Native, Container-Def, and Container-Opt with 4 bytes message size are $1.55us$, $2.1us$, and $1.57us$, respectively. Compared to the Container-Def case, the Container-Opt scheme could reduce the latency by up to 27%. The bandwidths of Native, Container-Def, and Container-Opt with 16 bytes message size are 12.61 MB/s, 10.14 MB/s, and 12.55 MB/s,



(a) Intra-Node Latency: Small Messages (b) Intra-Node Latency: Large Messages (c) Intra-Node Bandwidth: Small Messages (d) Intra-node Bandwidth: Large Messages

Fig. 6: MPI Point-to-Point Performance for GPU-to-GPU Communication

respectively. Compared to the Container-Def scheme, the Container-Opt scheme could improve the bandwidth by up to 24%. In addition, Container-Opt only incurs minor overhead, compared to the Native scheme.

From Figure 6(a)-6(b) and Figure 6(c)-6(d), we can clearly observe that both Container-Def and Container-Opt schemes start having a similar performance after 16 bytes message size. The is because GDR-loopback scheme is selected by the communication coordinator. For example, the latencies of Container-Def and Container-Opt with 8K bytes message size are 7.2us, and 7.34us, respectively, as shown in Figure 6(b). The bandwidths of Container-Def and Container-Opt with 256K bytes message size are 6.59GB/s and 6.43GB/s, respectively, as shown in Figure 6(d). Compared to the Native scheme, both Container-Opt and Container-Def can deliver the near-native performance. For large messages, cudaIPC performs better than other schemes, that is why we can see Container-Opt brings up to 46% improvement for latency and 32% improvement for bandwidth, compared with Container-Def. Note that the optimal communication channel switch points (from GDR-loopback to cudaIPC) are different for latency (around 16KB) and bandwidth (around 512KB), which further verifies the performance characterization results in Table III. As we can see, the proposed Container-Opt keeps delivering near-native performance. To compare with other MPI stacks, we also present the intra-node point-to-point performance with Open MPI in the native environment, which is denoted as 'Native-OpenMPI'. As can be seen in Figure 6, the evaluation results show clear performance gaps on both latency and bandwidth tests as comparing Native-OpenMPI with our proposed C-GDR design. This implies that there is no container-aware design in Open MPI for GPU communication.

In addition, all GPU-to-GPU data movement schemes that we have investigated are applied across multiple containers within a single node. The data movement scheme for inter-node case remains the same. Thus, the inter-node latency and bandwidth performance of both Container-Def and Container-Opt schemes achieve similar performance as the Native scheme. We omit these results due to the space limit. The MPI micro-benchmark level point-to-point evaluation results indicate that the Container-Opt scheme could always select the optimal communication channel for different message sizes and communication patterns to achieve the optimal performance, compared with the Container-Def scheme.

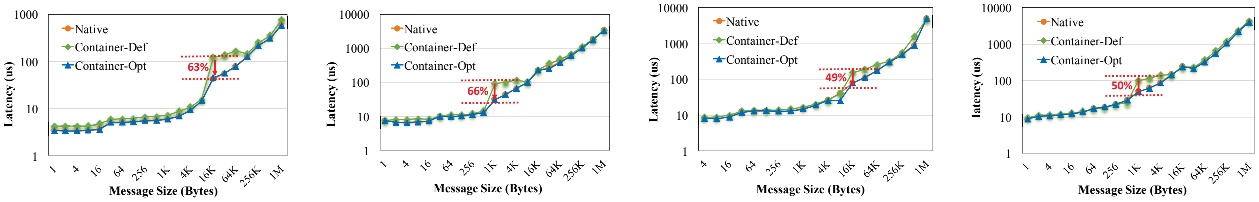
C. MPI Level Collective Micro-benchmarks

In this section, we evaluate the proposed C-GDR communication schemes with four MPI level collective operations including MPI_Bcast, MPI_Allgather, MPI_Allreduce, and MPI_Alltoall. We choose these four collective operations since they are widely used by GPU-based applications. The performance results are shown in Figure 7. The evaluation results indicate that our optimized communication scheme Container-Opt can achieve near-native performance for collectives also. Compared with the performance of Container-Def scheme in the container environment, our optimized scheme brings up to 63%, 66%, 49%, and 50% performance improvement for MPI_Bcast, MPI_Allgather, MPI_Allreduce, and MPI_Alltoall, respectively. We also evaluate these four collective operations with Open MPI. The evaluation results with Open MPI again show significant performance gaps as compared with our proposed C-GDR design. To clearly illustrate the benefit between Container-Opt and Container-Def, we do not include Open MPI results in Figure 7.

D. Application Performance Evaluation

In this section, we evaluate our proposed C-GDR scheme Container-Opt with several end applications, which include Jacobi solver, HOOMD-blue Lennard-Jones liquid (Hoomd-LJ), and Anelastic Wave Propagation (AWP-ODC), as shown in Figure 8. Jacobi solves the Poisson equation on a rectangle with Dirichlet boundary conditions. It leverages the CUDA-aware MPI to directly send/receive (MPI_Sendrecv) through the device buffer without explicitly staging the data to the host buffer. We can adjust the message size for sending and receiving. Jacobi-16B and Jacobi-512KB mean that we use 16 bytes and 512K bytes as message sizes, respectively.

In the Jacobi-16B case, the evaluation results indicate that our proposed Container-Opt can bring 25% communication performance improvement, compared with the default case, while having a similar performance with the one on the native environment. This is because the GDRCOPY is used in the native and Container-Opt case, which can bring optimal communication performance for message transfer with 16 bytes message size, as what we summarized in Table III. In the Jacobi-512K case, the cudaIPC scheme is used to deliver the optimal communication performance. This is the reason we see similar communication times between Native and Container-



(a) MPI_Bcast

(b) MPI_Allgather

(c) MPI_Allreduce

(d) MPI_AlltoAll

Fig. 7: MPI Collective Communication Performance across 16 GPU Devices

Opt schemes, and 26% performance improvement compared with the Container-Def scheme.

Both Hoomd-LJ and AWP-ODC dominantly use around 1M bytes message for communication, and the intra-node based communication scheme (cudaIPC) performs better than the GDR-loopback scheme, as we summarized in Table III. Accordingly, we can see from Figure 8, our proposed Container-Opt is able to achieve the optimal performance for applications Hoomd-LJ and AWP-ODC. Our C-GDR design can bring 10% and 14% performance improvement for Hoomd-LJ and AWP-ODC, respectively, compared with the Container-Def case.

VI. Related Work

There are four ways to use GPU in a Virtual Machine (VM): I/O pass-through, device emulation, API remoting, and mediated pass-through. In a virtualized environment, the GPU device could be directly passed through to a specific VM [9]. Using this technique, Amazon has provided GPU instances to HPC customers. Intel has introduced VT-d allows GPU to be passed to a virtual machine exclusively [10]. With GPU device passthrough, the device is dedicated to a specific virtual machine, so it sacrifices the sharing capability of the virtualized environment. CPU virtualization could be done through device emulation; however, such emulation technique could be done with GPUs easily.

GPU virtualization could also be achieved through API remoting which is commonly used in commercial software. API remoting forwards GPU commands from guest OS to host. VMGL [20] replaces the standard OpenGL library in Linux Guests with its own implementation to pass the OpenGL commands to VMM. Shi et al. present a CUDA-oriented GPU virtualization solution in [26]. It uses API interception to capture CUDA calls on the guest OS with a wrapper library, and then redirects them to the host OS where a stub service was running. Duato et al. [11] have proposed a library to allow each node in a cluster access any of the CUDA-compatible accelerators installed in the cluster nodes. Remote GPUs are virtualized devices made available by a wrapper library replacing the CUDA Runtime. This library forwards the API calls to a remote server and retrieves the results from those remote executions to offer them to the calling application. Gupta, et al. [14] use the same technique to forward CUDA command and OpenCL commands, solving the problem of virtualizing GPGPU devices. VMware products consist of a virtual PCI device and its corresponding driver for different operating systems. The host handles all accesses to the virtual PCI device inside a VM by a user-level process, where the actual GPUvm presents a GPU virtualization solution on an NVIDIA card [29] and it implements both para- and full-virtualization. However, full-virtualization exhibits a considerable overhead for MMIO handling. Compared to native, the performance of optimized para-virtualization is two to three times slower. Since NVIDIA has individual graphics memory on the PCI card, GPUvm cannot handle page faults caused by NVIDIA GPUs [13]. NVIDIA GRID [4] is a proprietary virtualization solution from NVIDIA for Kepler architecture. However, there are no technical details about their products available to the public. Reano et al. propose optimizations at InfiniBand network verbs-level to accelerate GPU virtualization framework [25]. Ravi et al. implement a scheduling policy, based on affinity score between GPU kernels when consolidating kernels among multiple VMs [24]. Iserte et al. propose to decouple real GPUs from the compute nodes by using the virtualization technology rCUDA [16]. Compared to these work, our work focuses on analyzing and characterizing different GPU-to-GPU communication schemes on container-based cloud environments, identifying performance bottlenecks. Based on our findings, we further propose C-GDR, a high performance container-aware GPUDirect communication schemes on RDMA networks, which can dynamically schedule the optimal communication channels.

VII. Conclusion and Future Work

The increase of cloud-based applications leveraging GPUs has made it vital for researchers and developers to understand and design efficient GPU-based communication schemes in cloud environments. This paper first investigates the performance characteristics of state-of-the-art GPU-based communication schemes on both native and container-based environments and identifies performance bottlenecks for communication in GPU-enabled cloud environments. To alleviate the bottlenecks identified, this paper presents the C-GDR approach to design high-performance container-aware GPUDirect communication schemes on RDMA networks and integrates C-GDR with the MVAPICH2 MPI library. The proposed designs provide locality-aware, NUMA-aware, and communication-pattern-aware capabilities to enable intelligent and adaptive communication coordination for the optimal communication performance on GPU-enabled clouds. The techniques we used are based on tradeoffs for performance and easy-to-adopt

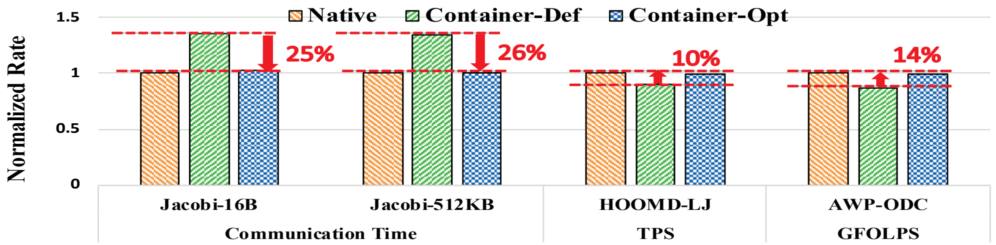


Fig. 8: Application Performance on 16 GPUs (Communication Time, lower is better; TPS and GFLOS, higher is better)

for a large user base. Performance evaluations show that MVAPICH2 with C-GDR has notable performance benefit compared with Open MPI. Further, MVAPICH2 with C-GDR can outperform default MVAPICH2-GDR schemes by up to 66% on micro-benchmarks and 26% performance benefit for various applications on container-based GPU-enabled clouds.

In the future, we plan to explore VM live migration support with GPU-enabled VM environments.

References

[1] Intel MPI Library. http://software.intel.com/en-us/intel-mpi-library/. [Last Accessed: February 25, 2019].

[2] MVAPICH: MPI over InfiniBand, Omni-Path, Ethernet/iWARP, and RoCE. http://mvapich.cse.ohio-state.edu/. [Last Accessed: February 25, 2019].

[3] NVIDIA GPUDirect RDMA. http://docs.nvidia.com/cuda/gpudirect-rdma/. [Last Accessed: February 25, 2019].

[4] NVIDIA GRID. http://www.nvidia.com/object/grid-technology.html. [Last Accessed: February 25, 2019].

[5] OpenMPI: Open Source High Performance Computing.

[6] OSU Micro-benchmarks. http://mvapich.cse.ohio-state.edu/benchmarks/. [Last Accessed: February 25, 2019].

[7] C.-H. Chu, K. Hamidouche, A. Venkatesh, D. S. Banerjee, H. Subramoni, and D. K. Panda. Exploiting Maximal Overlap for Non-Contiguous Data Movement Processing on Modern GPU-Enabled Systems. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 983–992, May 2016.

[8] C.-H. Chu, X. Lu, A. A. Awan, H. Subramoni, B. Elton, and D. K. Panda. Exploiting Hardware Multicast and GPUDirect RDMA for Efficient Broadcast. *IEEE Transactions on Parallel and Distributed Systems*, 30(3):575–588, March 2019.

[9] Y. Dong, J. Dai, Z. Huang, H. Guan, K. Tian, and Y. Jiang. Towards High-quality I/O Virtualization. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, SYSTOR '09, 2009.

[10] M. Dowty and J. Sugerman. GPU Virtualization on VMware's Hosted I/O Architecture. *SIGOPS Oper. Syst. Rev.*, 2009.

[11] J. Duato, A. J. Pea, F. Silla, R. Mayo, and E. S. Quintana-Ort. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *2010 International Conference on High Performance Computing Simulation*, June 2010.

[12] J. Glaser, T. D. Nguyen, J. A. Anderson, P. Lui, F. Spiga, J. A. Millan, D. C. Morse, and S. C. Glotzer. Strong Scaling of General-purpose Molecular Dynamics Simulations on GPUs. *Computer Physics Communications*, 192:97 – 107, 2015.

[13] M. Gottschlag, M. Hillenbrand, J. Kehne, J. Stoess, and F. Bellosa. LoGV: Low-Overhead GPGPU Virtualization. In *2013 IEEE 10th International Conference on High Performance Computing and Communications 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, pages 1721–1726, Nov 2013.

[14] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan. GViM: GPU-accelerated Virtual Machines. In *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*, HPCVirt '09, 2009.

[15] IBM Spectrum MPI. https://www.ibm.com/us-en/marketplace/spectrum-mpi. [Last Accessed: February 25, 2019].

[16] S. Iserte, F. J. Clemente-Castelló, A. Castelló, R. Mayo, and E. S. Quintana-Ortí. Enabling GPU Virtualization in Cloud Environments. In *Proceedings of the 6th International Conference on Cloud Computing and Services Science - Volume 1 and 2*, CLOSER 2016, 2016.

[17] J. Zhang and X. Lu and D. K. Panda. Designing Locality and NUMA Aware MPI Runtime for Nested Virtualization based HPC Cloud with SR-IOV Enabled InfiniBand . In *13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '17)*, Xi'an, China, April 2017.

[18] J. Zhang, X. Lu, J. Jose, M. Li, R. Shi, D. K. Panda. High Performance MPI Library over SR-IOV Enabled InfiniBand Clusters. In *Proceedings of International Conference on High Performance Computing (HiPC)*, Goa, India, December 17-20 2014.

[19] G. Jo, J. Nah, J. Lee, J. Kim, and J. Lee. Accelerating LINPACK with MPI-OpenCL on Clusters of Multi-GPU Nodes. *IEEE Transactions on Parallel and Distributed Systems*, 26(7):1814–1825, July 2015.

[20] H. A. Lagar-Cavilla, N. Tolia, M. Satyanarayanan, and E. de Lara. VMM-independent Graphics Acceleration. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, 2007.

[21] NVIDIA Docker. https://github.com/NVIDIA/nvidia-docker/wiki. [Last Accessed: February 25, 2019].

[22] S. Potluri, K. Hamidouche, A. Venkatesh, D. Bureddy, and D. Panda. Efficient Inter-node MPI Communication Using GPUDirect RDMA for InfiniBand Clusters with NVIDIA GPUs. In *Parallel Processing (ICPP), 2013 42nd International Conference on*, pages 80–89, Oct 2013.

[23] S. Potluri, H. Wang, D. Bureddy, A. K. Singh, C. Rosales, and D. K. Panda. Optimizing MPI Communication on Multi-GPU Systems Using CUDA Inter-Process Communication. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 1848–1857, May 2012.

[24] V. T. Ravi, M. Becchi, G. Agrawal, and S. Chakradhar. Supporting GPU Sharing in Cloud Environments with a Transparent Runtime Consolidation Framework. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, HPDC '11, 2011.

[25] C. Reano and F. Silla. InfiniBand Verbs Optimizations for Remote GPU Virtualization. In *2015 IEEE International Conference on Cluster Computing*, 2015.

[26] L. Shi, H. Chen, and J. Sun. vCUDA: GPU accelerated high performance computing in virtual machines. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–11, May 2009.

[27] R. Shi, S. Potluri, K. Hamidouche, X. Lu, K. Tomko, and D. Panda. A Scalable and Portable Approach to Accelerate Hybrid HPL on Heterogeneous CPU-GPU Clusters. In *Proceeding of CLUSTER (CLUSTER'13)*, Indianapolis, Indiana, USA, 2013.

[28] R. Shi, S. Potluri, K. Hamidouche, J. Perkins, M. Li, D. Rossetti, and D. K. Panda. Designing Efficient Small Message Transfer Mechanism for Inter-node MPI Communication on InfiniBand GPU Clusters. In *2014 21st International Conference on High Performance Computing (HiPC)*, pages 1–10, Dec 2014.

[29] Y. Suzuki, S. Kato, H. Yamada, and K. Kono. GPUvm: Why Not Virtualizing GPUs at the Hypervisor? In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, 2014.

[30] J. Zhang, X. Lu, and D. K. Panda. High Performance MPI Library for Container-Based HPC Cloud on InfiniBand Clusters. In *2016 45th International Conference on Parallel Processing (ICPP)*, Aug 2016.

[31] J. Zhang, X. Lu, and D. K. Panda. High-Performance Virtual Machine Migration Framework for MPI Applications on SR-IOV enabled InfiniBand Clusters. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Orlando, USA, May 2017.