OMB-UM: Design, Implementation, and Evaluation of CUDA Unified Memory Aware MPI Benchmarks

K. Vadambacheri Manian, C.-H. Chu, A. A. Awan, K. Shafie Khorassani, H. Subramoni, D. K. Panda
Department of Computer Science and Engineering
The Ohio State University
{vadambacherimanian.1, chu.368, awan.10, shafiekhorassani.1}@osu.edu
{subramon, panda}@cse.ohio-state.edu

***Abstract*—Unified Memory (UM) has significantly simplified the task of programming CUDA applications. With UM, the CUDA driver is responsible for managing the data movement between CPU and GPU and the programmer can focus on the actual designs. However, the performance of Unified Memory codes has not been on par with explicit device buffer based code. To this end, the latest NVIDIA Pascal and Volta GPUs with hardware support such as fine-grained page faults offer the best of both worlds, i.e., high-productivity and high-performance. However, these enhancements in the newer generation GPU architectures need to be evaluated in a different manner, especially in the context of MPI+CUDA applications.**

In this paper, we extend the widely used MPI benchmark OSU Micro-benchmarks (OMB) to support Unified Memory or Managed Memory based MPI benchmarks. The current version of OMB cannot effectively characterize UM-Aware MPI design because CUDA driver movements are not captured appropriately with standardized Host and Device buffer based benchmarks. To address this key challenge, we propose new designs for the OMB suite and extend point to point and collective benchmarks that exploit sender and receiver side CUDA kernels to emulate the effective location of the UM buffer on Host and Device. The new benchmarks allow the users to better understand the performance of codes with UM buffers through user-selectable knobs that enable or disable sender and receiver side CUDA kernels. In addition to the design and implementation, we provide a comprehensive performance evaluation of the new UM benchmarks in the OMB-UM suite on a wide variety of systems and MPI libraries. From these evaluations we also provide valuable insights on the performance of various MPI libraries on UM buffers which can lead to further improvement in the performance of UM in CUDA-Aware MPI libraries.

***Index Terms*—Benchmark, CUDA, GPU, Unified Memory, MPI, HPC**

I. Introduction

Micro-benchmarks play a vital role in characterizing the behavior of a system. A well-developed benchmark provides the user with various knobs to play around with to understand the system under study. Moreover, the results from the benchmark should not be ambiguous and should clearly expose the characteristics of the system as it is. Even though benchmarks have been written and maintained for almost all areas of computing, its usefulness is highly prominent in the area of parallel or high performance computing where a plethora of parameters interact in seemingly unintuitive ways. A benchmark which can rightly characterize such a system can provide great insights to the developer of parallel systems.

Message Passing Interface (MPI) is the de facto programming model for computing in parallel across multiple nodes of a High Performance Computing (HPC) cluster. Modern MPI libraries consist of numerous parameters which can act as knobs to tweak the MPI library to enable the best performance. Micro-benchmarks provide the opportunity to play around with these parameters to characterize an MPI library. As the prevalence of Graphics Processing Unit (GPU) architectures in HPC clusters continues to increase, the number of applications taking advantage of it to accelerate performance also have risen. GPUs are especially noted for their role in improving the performance of traditional HPC [1] and machine learning [2], [3] applications.

Compute Unified Device Architecture (CUDA) platform aids in the task of programming these high performance NVIDIA GPUs efficiently. Various CUDA versions are released over the years which provided new features that efficiently take advantage of the improvements in NVIDIA GPU architectures, thereby yielding higher performance. To harness computing power on GPU clusters, it is common to adopt CUDA-aware MPI libraries such as OpenMPI and MVAPICH2-GDR, which can be used to perform communication between GPUs within a node or between multiple GPU nodes. CUDA-aware MPI libraries provide efficient schemes to handle GPU-resident data directly in the MPI primitives; thus relieving the application developers from managing memory explicitly, increasing productivity and performance [4]. Benchmarks such as OMB [5] have been developed to characterize and study the performance of CUDA aware MPI libraries.

Recently, Unified Memory (UM), a new feature being introduced and enhanced in recent CUDA releases, significantly improves the productivity for developing CUDA-enabled applications. Prior to the introduction of UM, the memory was allocated either on the host (through *malloc*) or the device (through *cudaMalloc*), and the programmer had to explicitly perform the data movement between device and host using CUDA application programming interfaces (APIs) such as *cudaMemcpy*. UM provides an abstraction of unified memory address space for the programmer. As a result, the programmer no longer has to explicitly manage the data movement, which will be handled by the underlying CUDA driver, enhancing productivity. While UM improves productivity for the pure CUDA applications, it falls short in performance when UM is used for MPI+CUDA applications. To address the performance issue, UM-aware MPI designs to alleviate the performance impact [6] were proposed.

More recently, the deployment of advanced GPU page fault engines in the NVIDIA Pascal and Volta GPU architectures significantly improve the performance of page migration between CPU and GPU for UM. This development along with the inherent nature of UM's effective location being either in the CPU or GPU adds a unique feature to the UM aware MPI library which cannot be fully characterized using previous micro-benchmarks such OMB [5]. Thus, there is a need in the HPC community to develop a benchmark such as OMB-UM that can thoroughly study and characterize UM aware MPI libraries and take full advantage of the UM.

In this context, we take up the challenges and propose OMB-UM that aims to fill these gaps to enable full characterization of UM-based MPI libraries on current and future GPU architectures. To the best of our knowledge, this is the first work to design, implement and evaluate UM-Aware MPI benchmark suite on modern GPU clusters. This paper makes the following key contributions.

- Design and implementation of comprehensive UM-aware point-to-point benchmarks with extended options to specify the effective location of the UM buffer.
- Design and implementation of UM-aware collective communication benchmarks with extended options to determine the effective location of the UM buffer.
- Comprehensive performance evaluation of the new UM-aware benchmarks on a wide variety of systems and multiple CUDA-aware MPI libraries.

II. Motivation and Challenges

Prior to the introduction of UM, the data buffers used in the CUDA-aware MPI library can be identified either as a host buffer or device buffer based on their location. These buffers reside in their respective location throughout the entire duration of the MPI+CUDA program. With the introduction of UM, another buffer type called unified memory is added to the buffer types handled by CUDA-aware/UM-aware MPI library. The unique nature of UM buffers, unlike the other types is that they can now move between host and device. They are moved by the underlying CUDA driver based on whether CPU or GPU is accessing the buffer. The *effective location* of an UM buffer is its current location of residence which can be either 'host' or 'device' and will change dynamically during program runtime.

A. Limitations of current UM-aware MPI benchmarks

The variability of UM buffer's effective location in-turn affects the characteristics of MPI operations using this buffer. The current UM-aware MPI benchmark suite such as OMB lacks options to capture accurately the effect of this variability on performance of MPI operations. This is explained by taking 2DStencil code as an example.

Figure 1 shows the computation kernel of the 2D stencil code. The computation pattern starts with a CUDA kernel then followed by MPI calls and then followed by couple of CUDA

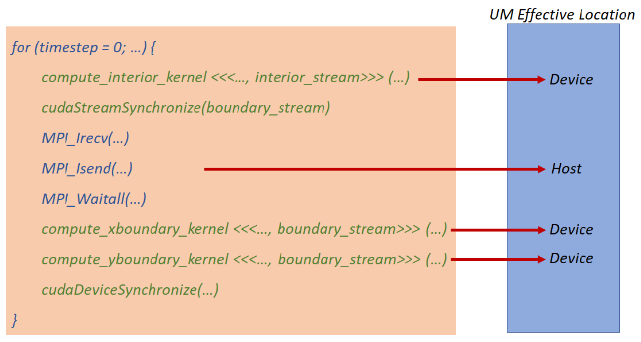

```
for (timestep = 0; ...) {
    compute_interior_kernel <<<..., interior_stream>>> (...)
    cudaStreamSynchronize(boundary_stream)
    MPI_Irecv(...)
    MPI_Isend(...)
    MPI_Waitall(...)
    compute_xboundary_kernel <<<..., boundary_stream>>> (...)
    compute_yboundary_kernel <<<..., boundary_stream>>> (...)
    cudaDeviceSynchronize(...)
}
```

Fig. 1. Pseudocode of a common 2DStencil kernel in CUDA+MPI applications

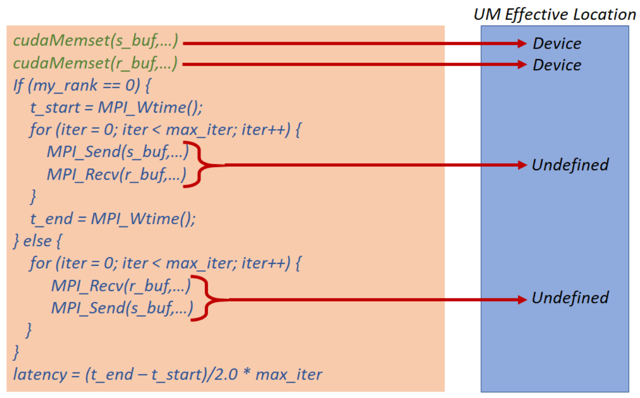

```
cudaMemset(s_buf,...)
cudaMemset(r_buf,...)
If (my_rank == 0) {
  t_start = MPI_Wtime();
  for (iter = 0; iter < max_iter; iter++) {
    MPI_Send(s_buf,...)
    MPI_Recv(r_buf,...)
  }
  t_end = MPI_Wtime();
} else {
  for (iter = 0; iter < max_iter; iter++) {
    MPI_Recv(r_buf,...)
    MPI_Send(s_buf,...)
  }
}
latency = (t_end – t_start)/2.0 * max_iter
```

Fig. 2. Pseudocode of existing latency benchmark in OMB suite

kernels. When this pattern is present within a for loop, the computation alternates between device and host. When UM buffers are used in this stencil code, the effective location of the UM buffer also alternates between host and device.

On the other hand Figure 2 shows the computation kernel inside the osu_latency benchmark from the OMB benchmark suite. Here we could see that the device buffers are set in the beginning of the benchmark using cudaMemset() and then MPI_Send() and MPI_Recv() MPI calls are used inside for loops to calculate the latency. Thus when UM buffers are used in this benchmark, the effective location of the buffer will initially be on the device during the initialization phase. During the iteration of the loop the effective location of the UM buffer can either be host or device depending upon the MPI implementation and hence 'Undefined'. Thus the current version of OMB benchmark is unable to capture the variability in effective location of UM buffers as found in the stencil code described above. Thus there is a need to extend the OMB suite to capture the effective location variability of UM buffers.

B. Benchmark Extensions

The proposed benchmark extensions provide the necessary options to explicitly dictate the effective location of the UM buffer. For the point to point MPI operations the proposed OMB-UM adds four possible effective locations for the UM buffer namely 1)MH-MH 2)MD-MH 3)MH-MD 4)MD-MD. Here H indicates the effective location of a UM buffer as host while D indicates the effective location of a UM buffer as a device. Furthermore, the first 'Mx' and second 'My' in Mx-My refers to the UM buffer type of process 0 and process 1 respectively. For collective operations the -d managed option already available in OMB is extended to set the effective location of UM buffer as 'device' similar to MD-MD in point to point operations.

C. Broad Challenge

The following broad challenge is investigated in this paper: ***How can a full-fledged UM aware OMB (OMB-UM) be designed to provide the facility to set the four possible effective locations for UM buffers leading to the full characterization of UM aware MPI on modern GPU clusters?***. This broad challenge leads to the following key questions that need to be addressed:

- How to achieve the different data placements on UM provided that the underlying UM driver migrates pages between host and device based on the recent access?
- How to properly measure and adjust the kernel launch time overhead during the timing calculations inside the benchmark?

III. Background

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

A. NVIDIA GPU, CUDA and Unified Memory Technology

NVIDIA General Purpose Graphics Processing Unit (GPGPU) has been widely deployed in HPC systems to accelerate applications due to its massive parallelism and high memory bandwidth. CUDA is the parallel computing platform and programming model to harness the computing power of NVIDIA GPUs. Unified Virtual Addressing (UVA) is a feature introduced in CUDA 4.0 that provides a single virtual memory address space for all CPU and GPU memory. Through UVA, the physical memory location can be determined from a pointer value that is accessible by the GPU regardless of where it resides in the system. Unified memory was introduced by NVIDIA in CUDA 6.0 to simplify the complexity of sharing data between the CPU and GPU. Prior to this feature, in order to share data between the CPU and GPU, memory had to be allocated on both the host and the device and explicit copies made between them, incurring additional overhead. Unified memory provides a single, unified virtual address space to applications for accessing CPU and GPU memory. The managed memory is shared between CPU and GPU, eliminating the need for explicit data transfers in applications. Unlike UVA, with unified memory, the CUDA driver automatically migrates data between CPU and GPU. Since the release of CUDA 8.0 and NVIDIA Pascal GPU architecture, the hardware support for unified memory has significantly improved [7]. Memory page faulting support is a significant enhancement enabling on-demand migration and concurrent access. On-demand migration yields improved performance through a small amount of page movement and better data locality.

The latest NVIDIA Volta GPU architecture, further optimizes the performance of unified memory through the access counter hardware feature. It ensures memory pages are moved to the physical memory of the processor, i.e., to the physical memory of CPU or GPU, based on which accesses the pages most frequently. To further improve productivity, Volta over NVLink2 interconnect addresses translation services (ATS) which allows the GPU to directly access page tables of CPU [8].

B. CUDA-Aware MPI

MPI is a de-facto standard programming model used for parallel applications on HPC systems. With the popularity of GPU-enabled HPC systems in recent years, CUDA-Aware MPI [9] has been widely adopted in communication libraries such as OpenMPI [10], MVAPICH2 [11], and more [12]. CUDA-Aware MPI eliminates explicit handling of data movement between the CPU and GPU by directly passing pointers to GPU memory instead. Advanced CUDA-Aware MPI libraries take advantage of NVIDIA GPUDirect technology to provide low-latency and high-bandwidth data transfer among GPUs within and across nodes through Peer-to-Peer (P2P) Communication and GPUDirect Remote Direct Memory Access (RDMA) [13]. CUDA managed memory-aware MPI designs [14], [6] are proposed and evaluated on NVIDIA Kepler GPU architecture. Unified memory technology has significantly evolved over the years leading to a dearth of understanding CUDA-Aware MPI with the use of CUDA Unified Memory on the modern GPU architectures like NVIDIA Pascal and Volta. This paper aims to fill this gap in knowledge.

C. Advanced CUDA-IPC design in MVAPICH2 MPI

Advanced CUDA-IPC designs for managed memory are proposed [6] to improve the performance of large message communication within a single node. These designs take advantage of NVIDIA GPUDirect Peer to Peer technology to load and store data directly between two GPU memories. Unfortunately this technology is available only for device buffers and not for managed buffers. Advanced CUDA-IPC design for managed memory [6] brings this technology for managed buffers by copying the contents of managed buffers to a device buffer and performing CUDA-IPC on them.

D. MPI-Level Performance Benchmark

There are many MPI benchmarks existed for different purpose, however, very few of them supports CUDA-Aware MPI and UM-Aware buffers. The OSU Micro-benchmark (OMB) suite is used with various MPI libraries to evaluate performance of MPI primitives on CPU and GPU clusters. In order to evaluate various communication configurations on GPU clusters, OMB provides support for evaluating the performance of point-to-point, multi-pair, and collective communication [15], [5]. The point-to-point MPI benchmarks include tests for latency, uni-directional bandwidth, and bi-directional bandwidth. The latency test is carried out in a ping-pong manner using MPI_Send and MPI_Recv. The uni-directional bandwidth test starts off with sending a number of back-to-back messages from the sender using MPI_ISend. The sender waits (MPI_Waitall) for a message from the receiver process confirming it received all the data using MPI_IRecv. Similar logic is followed in the bi-directional bandwidth test but with both the sender and the receiver processes sending back-to-back messages to each other. Both processes then wait for confirmation from each other that all the messages have been received. These benchmarks take two parameters to indicate whether the buffers reside on the host or device. The various communication patterns include inter-node and intra-node communication on the host, on the device, or between the host and device.

IV. Proposed Designs

The proposed OMB-UM is aiming to provide flexibility to benchmark communication performance with the various effective location of UM that mimic real applications using CUDA-Aware MPI libraries as described in Figure 1. This section delves into how proper data placements are achieved in OMB-UM and describes the subsequent algorithms for MPI operations that use them internally.

A. Data Placements of Unified Memory

The on-demand migration of UM between CPU and GPU occurs transparently via the underlying CUDA driver whenever CPU or GPU *touches (i.e., accesses)* the managed buffer. To explicitly move the memory pages from CPU to GPU, one could launch a kernel that reads the managed memory buffer in the GPU thereby touching it, and we refer this to *data migration kernel* in the rest of the paper. This forces the underlying CUDA driver to migrate the memory pages to GPU if needed. Similarly, reading the managed memory buffer from CPU will force the buffer to be moved to the system memory. When performing data transfer between two processes, the

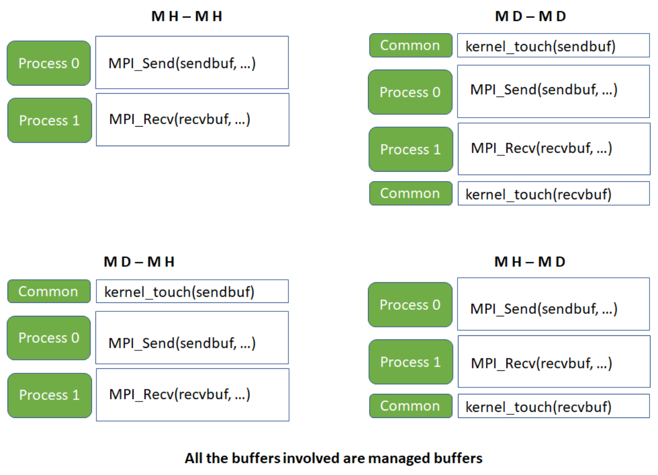


Fig. 3. Data placement when using unified memory for communication

effective location of UM varies when entering the communication runtime. Figure 3 illustrates the four possible data placements for the UM buffer as follows.

1) MH-MH: In this data placement, the effective location of both the send and receive buffers in a point to point operation is set to "host". In other words, data reside on host memory before and after the transaction of the point-to-point operation. Since the default effective location is "host" when allocating a UM (e.g., using *cudaMallocManaged()* without additional setting), the benchmark does not need to explicitly *touch* the buffer in this scenario. This is the only scenario supported in the existing OMB suites [15].

2) MD-MH: For the MD-MH case, the effective location of the send buffer is set to "device" and the effective location of the destination buffer is set to host. This scenario mimics the case where the sender process invokes a kernel computation on GPU before issuing the send operation, where the receiver process does not involve any GPU kernels with the receive buffer.

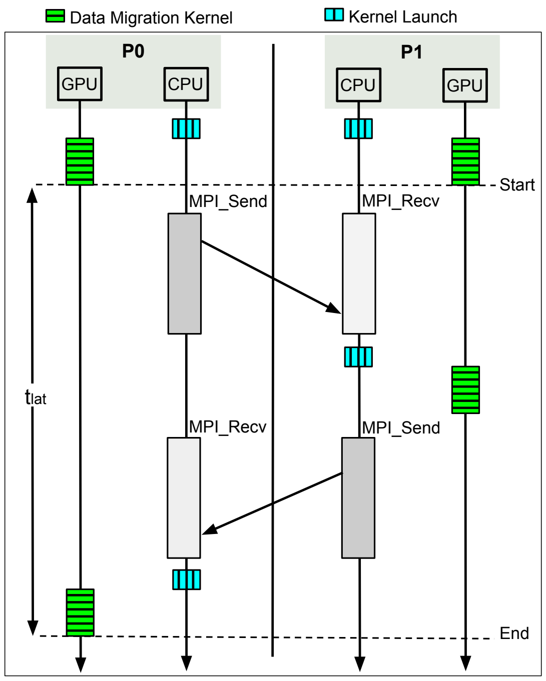
3) MH-MD: This is similar to the MD-MH case, but the roles are reversed. Here, the receiver process invokes computation on GPU after the inter-process communication. In the benchmark, this can be achieved by launching a kernel "after" the receive operations are completed.

4) MD-MD: This case is the most commonly used as shown in Figure 1 and it has both the sender and receiver sides set the effective location to "device" because of the GPU computing kernel before and after the send and receive operations. To mimic this scenario in the benchmark, we can launch a kernel "before" sending the data and also launching a kernel "after" receiving the data.

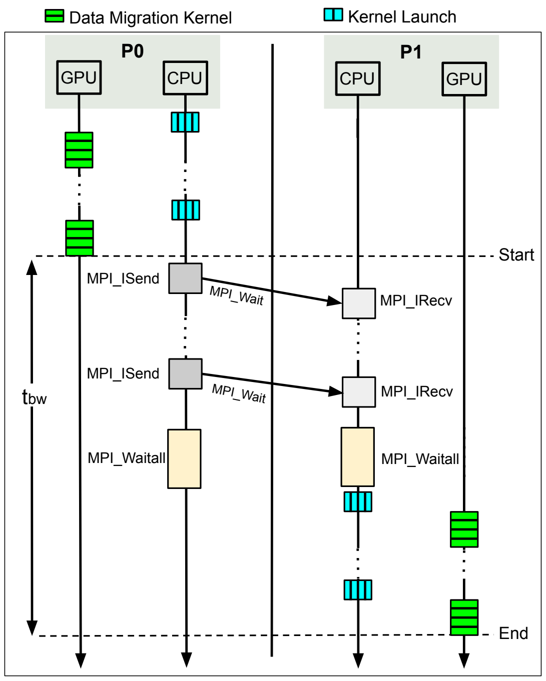
B. Proposed OMB-UM Latency Benchmark

This section describes the proposed OMB-UM latency benchmark, which is based on the existing ping-pong test in OMB [15], for the unified memory. Moreover, the effective location of UM buffers can be set as device or host at both the source and the destination. Here, we use the most common use case of MD-MD as shown in Figure 4(a) to illustrate the proposed design to measure the latency accurately when data migration of unified memory is involved during the communication. Let us consider that P0 be the process with rank 0 and P1 be the process with rank 1 that are involved in the point to point operation with UM buffers. Initially, P0 sets the effective location of the source UM buffer to device by launching a kernel before sending a message to P1 (before *Start* in Figure 4(a)). On the other side, upon P1 receiving the message from P0, it launches a kernel which causes the memory pages of receive buffer to be migrated to GPU memory.

In MD-MD case, the data is assumed to be present on GPU memory before and after the communication. Therefore, to report the latency of communication accurately, we do not include the time where driver migrates data from system memory to GPU memory before the communication and the duration where we launch the data migration kernel. However,



(a) OMB-UM Latency: MD - MD



(b) OMB-UM Bandwidth: MD - MD

Fig. 4. Proposed OMB-UM Latency and Bandwidth Benchmarks

we do include the data migration time after the communication to guarantee the GPU-resident data because the state-of-the-art CUDA-Aware MPI libraries do not provide such a guarantee. Finally, P0 can calculate the latency as follows.

$$Latency_{MD-MD} = \frac{t_{End} - t_{Start} - 2 \times t_{Kernel\_Launch}}{2} \tag{1}$$

Similarly, other data placement cases can be derived from this case. For example, in MD-MH case, there is no need to launch the data migration kernel after the communication.

C. Proposed OMB-UM Bandwidth Benchmark

The OMB-UM unidirectional bandwidth algorithm for the MD-MD data placement case is depicted in Figure 4(b). Similar to the bandwidth algorithm in OMB, P0, the process with rank 0, issues a window of non-blocking send operations, i.e., window_size times, to P1. In OMB-UM, P0 issue the data migration buffer with UM buffers that used in these non-blocking send operations. It then waits until all the sends are complete through calling MPI_Waitall. On the other side, P1, the process with rank 1, issues a window of non-blocking receive operation corresponding to the send operations in P0. It then waits for all the receives to complete and then sets the effective location of the UM buffers used in these non-blocking receives to device by launching the data migration kernel on every one of them. The total time is calculated from the time P0 started the non-blocking send until the time P1 finished setting the effective location of all the UM buffers to device. The total adjusted time is calculated by subtracting the time taken for kernel launches by both P0 and P1 from the total time. This is then used to calculate the bandwidth.

$$BW_{MD-MD} = \frac{M \times window\_size}{t_{bw} - t_{Kernel\_Launch}} \tag{2}$$

where M represents the message size used.

Other data placement cases can be derived from this case in a similar manner by removing the data migration kernels accordingly. Also, the bi-directional bandwidth benchmark can be simply implemented by introducing send and receive operations on the both processes.

D. Proposed OMB-UM Collective Benchmarks

This section uses broadcast as an example to demonstrate the efficacy of the proposed OMB-UM for benchmarking collective operations. A broadcast operation involves a root process sends the data to every other process in a group, i.e., communicator in MPI context. Thus, the techniques employed in the previously described point to point OMB-UM algorithms can be applied here. For the OMB-UM broadcast with the MD-MD data placement, the root process launches a kernel before sending the data through UM buffer to set its effective location as device. Then the broadcast operation is performed through calling MPI_Bcast. Similarly, all the non-root processes launch the data migration kernel after receiving the data to ensure the memory pages reside on GPU memory. All the processes involved in the broadcast operation would measure the broadcast time, the kernel launch overhead and execution time of data migration, and then calculate the latency of broadcast operation with UM buffer. This can be represented as Figure 4(a) by replacing MPI_Send and

TABLE I
EXPERIMENTAL PLATFORMS

CPU	GPU	Interconnect	NVLink	OS
Sandy Bridge E5-2670	Volta V100	EDR	No	RHEL 7.5.1804
Haswell E5-2687W	Volta V100	EDR	No	RHEL 7.5.1804
OpenPOWER POWER9	Volta V100	EDR	Yes	RHEL 7.6

MPI_Recv operations to just MPI_Bcast operation on multiple processes.

V. EVALUATION

This section presents the evaluation of the proposed OMB-UM benchmarks on various platforms and communication runtimes. First, we describe the experimental platforms and setup. Next, we present the evaluation of the OMB-UM on intra- and inter-node point-to-point communications on Intel x86-based GPU system with PCIe interconnect using MVAPICH2-GDR, a CUDA-Aware library with advanced UM design [6]. Next, we present the point-to-point and collective communications evaluation of OMB-UM on OpenPOWER system with NVLink interconnect using MVAPICH2-GDR, Spectrum MPI and Open-MPI+UCX libraries. We finally conclude the evaluation section with the discussions and insights obtained from the evaluation of OMB-UM on x86 and OpenPOWER systems.

A. Experimental Platforms and Design

The experiments were performed on three types of system architectures as shown in Table I: i) Intel x86 machine: a local cluster that has nodes with two NVIDIA Volta V100 GPUs connected by PCIe Gen3 x16 and Intel Sandy Bridge E5-2670 processors running at 2.60 GHz. Each node has two sockets with eight cores per socket connected using Intel QPI. The nodes run Red Hat Enterprise Linux Server release 7.5.1804 (Core) with a kernel version of 3.10.0-862.14.4. Mellanox OFED 4.5-1.0.1.0 and CUDA toolkit 10.1 were used. Mellanox Infiniband EDR networks are used to connect these nodes. It is used for obtaining x86 intra-node numbers ii) Intel x86 machine: a local cluster that has nodes with two generations of NVIDIA GPUs: Pascal P100 and Volta V100, both are connected by PCIe Gen3 x16 to Intel Haswell E5-2687W processors running at 3.1 GHz. Each node has two sockets with ten cores per socket connected using Intel QPI. The nodes run Red Hat Enterprise Linux Server release 7.4 (Core) with a kernel version of 3.10.0-693.21.1. Mellanox OFED 4.4-2.0.7 and CUDA toolkit 9.2.14x86 8 were used. Mellanox Infiniband EDR networks are used to connect these nodes. It is used for obtaining x86 inter-node numbers ii) IBM OpenPOWER machine: The node is equipped with two 22-core IBM POWER9 CPUs and 6 NVIDIA Volta V100 GPUs, where NVLink2 is used to connect CPU and GPU to provide advanced UM features as described in Section III-A. The nodes are connected via Mellanox InfiniBand EDR networks. The system has 256GB memory, and each GPU has 16GB GDDR5 memory. The node runs Red Hat Enterprise Linux Server release 7.6 (Maipo) with a kernel version of 4.14.0-115.8.1. CUDA toolkit 10.1.168 along with Mellanox OFED 4.5-2.2.9.0 is installed on this system.

On each of the above mentioned architectures, the extended point-to-point communication type benchmarks in the proposed OMB-UM such as latency (osu_latency), unidirectional bandwidth (osu_bw) and bidirectional bandwidth (osu_bibw) are first evaluated using different CUDA-Aware MPI libraries. The four data placements for UM effective location: 1) MH-MH 2) MD-MH 3) MH-MD 4) MD-MD discussed in Section IV are evaluated.

To gain insights into "UM-Awareness" of existing CUDA-Aware MPI libraries and cutting-edge interconnect technology, we performed the above evaluation using different MPI libraries such as MVAPICH2-GDR, OpenMPI and Spectrum MPI on the OpenPOWER-based GPU machine. Finally, we evaluated the extended UM-based collective benchmark using broadcast (osu_bcast) as the test candidate to show the efficacy of the proposed OMB-UM benchmark.

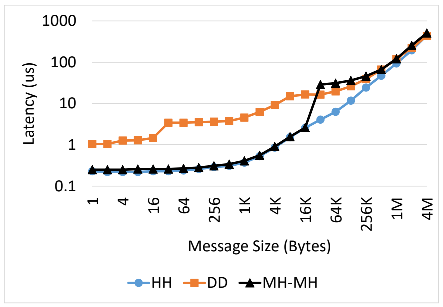
B. Intra-node Evaluation on Intel x86 System

In this section, the extended point-to-point benchmarks of OMB-UM are evaluated on a single node using MPVAICH2-GDR MPI library. We conducted experiments using the four extended data placement options for specifying the locality of UM buffers namely 1)MH-MH 2)MD-MH 3)MH-MD 4)MD-MD. Moreover, we compared these new options to host-to-host (labeled as *HH*) and device-to-device (labeled as *DD*) communication to understand the performance differences when moving from the explicit memory allocation to the implicit unified memory.

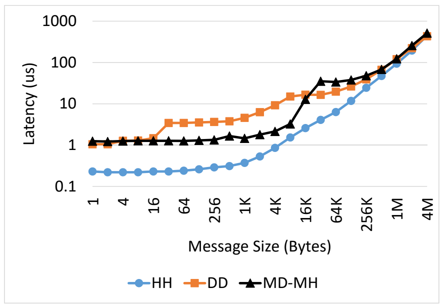
1) Latency(MH-MH): Figure 5(a) shows the latency curve for MH-MH data placement. This is the default data placement scenario in existing OMB suite. It shows that the curve aligns with the host (HH) path until the medium message range and then it suddenly increases and aligns with the device (DD) path. Since the UM buffer's effective location is set as host, it initially follows the host path. However, in the medium message range the advanced IPC designs for UM buffers as described in Section III-C kick in and change the effective location of UM buffer as device. Thus for medium and large message range it aligns with the device's path. This behavior of MH-MH curve needs to be fixed to align MH-MH curve with the HH curve.

Thus these data placement options to the UM buffer provide various knobs which an MPI designer can use to get more insights into the performance of UM-aware MPI library thereby improving its performance. Since these extended options are not available in current OMB benchmark where the only available option is MH-MH implicitly, it might sometimes mislead the UM-aware MPI developer with incorrect performance characteristics. This is also another motivation behind the development of OMB-UM.

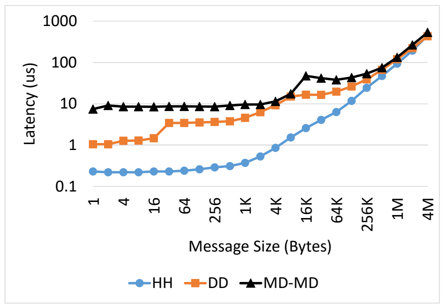
2) Latency(MD-MH and MH-MD): Figure 5 shows the latency curve for MD-MH data placement. This data placement is also called as the sender side kernel design as only a sender launches a kernel to move the source UM buffer in the host to the device. Since the UM buffer is moved to the device we see that the MD-MH latency curve initially aligns with the device



(a) Latency MH MH



(b) Latency MD MH



(c) Latency MD MD

Fig. 5. x86 intra-node latency on MVAPICH2-GDR using OMB-UM

(DD) and in the middle it is in-between device and host (HH) and finally it aligns with both the host and the device.

Similarly in MH-MD data placement one process sets the effective location of source UM buffer as host and another process sets the effective location of the destination UM buffer as device. Since OMB-UM benchmarks are designed to start the timing measurement after the source UM buffer is brought to the device, this MH-MD curve is similar to MD-MD curve and hence it is omitted here for space.

3) Latency(MD-MD): Figure 5(c) shows the latency curve for MD-MD data placement. Here, both the processes involved in the point to point operation launch a kernel to move the source and destination UM buffer from host to device. From the graph we could see that the MD MD curve has mostly aligned with the device (DD) curve for medium and large messages. Since MD-MD data placement sets the effective location of source and destination UM buffer to be on device consistently, this is semantically similar to host (HH) and device (DD) curves where their buffers also lie either at the host or device consistently during the entire point to point operation. Thus in all the subsequent graphs host (HH), device (DD) and MD-MD data placement is shown for consistency.

4) Bandwidth and Bi-Bandwidth(MD-MD): Figure 6 shows the bandwidth and bi-bandwidth curves for the MD-MD data placement. Both the processes involved in the bandwidth and bi-bandwidth benchmark launch a kernel to move the source and destination UM buffer to the device. From both the graphs we could see that UM buffers are having a low bandwidth compared to the device (DD) and host (HH) counterparts for small to medium message range. For large message range bandwidth and bi-bandwidth curves aligns with the host (HH) and device (DD) curves. This characteristic obtained from OMB-UM reflects the usefulness of MVAPICH2-GDR's advanced IPC design as explained in Section III-C for large messages.

C. Inter-node Evaluation on Intel x86 System

While the previous sections detailed the intra-node evaluation on X86, this section covers the inter-node evaluation. The latency, bandwidth and bi-bandwidth benchmarks are run on a couple of nodes with the UM data placement set as MD-MD and their results are presented in figure 7. From the latency curve we could see that MD-MD curve is slightly ahead of device (DD) curve as managed memory latency is expected to be higher than device (DD) curve due to underlying data migration costs. Furthermore, for the bandwidth and bi-bandwidth curves the performance of MD-MD curve is worse by an order of magnitude. This insight by the OMB-UM benchmark can be used by the UM-aware MPI designer to come up with better UM-aware MPI designs for improving the bandwidth and bi-bandwidth.

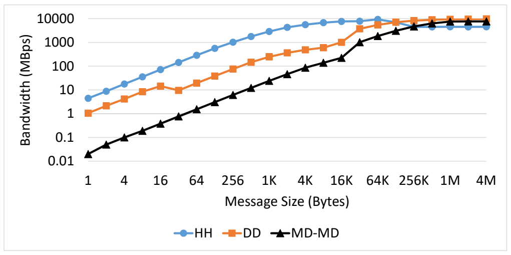
D. NVLink-enabled POWER9 intra-node and inter-node evaluation

To evaluate the effect of UM on the latest NVLink-enabled GPU systems, we conducted experiments using the proposed OMB-UM on a couple of GPU-enabled POWER9 nodes. Here, we perform point-to-point communication between two GPUs with direct NVLink and report the latency, uni-bandwidth, and bi-bandwidth. We compared the performance between three CUDA-Aware MPI libraries: 1) MVAPICH2-GDR 2.3.2 (*MVAPICH2-GDR*), 2) OpenMPI 4.0.1 with UCX 1.6 (*OpenMPI+UCX*), and 3) SpectrumMPI 10.3.0.01 (*SpectrumMPI*).

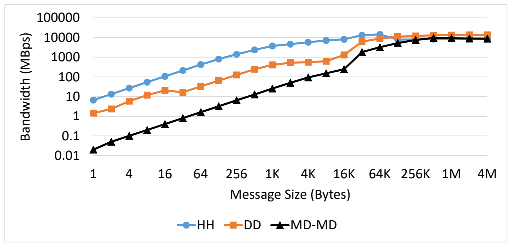
In Figure 8(a), we can see that MVAPICH2-GDR and OpenMPI+UCX provide the lowest latency and SpectrumMPI delivers high latency where data migration is occurring for small messages. For large messages, MVAPICH2-GDR significantly outperforms OpenMPI+UCX and SpectrumMPI due to the advanced designs for UM. For bandwidth and bi-bandwidth as shown in figures 8(b) and 8(c), MVAPICH2-GDR and SpectrumMPI perform similar until advanced IPC designs in MVAPICH2-GDR takes effect in larger message sizes. The bandwidth of OpenMPI+UCX is limited by explicit data movement between host and GPU memory.

E. NVLink-enabled POWER9 intra-node collective communication evaluation

Finally, we present the performance comparison of MPI libraries performing UM-based broadcast on the POWER9 machine as a proof-of-concept of completeness of the proposed

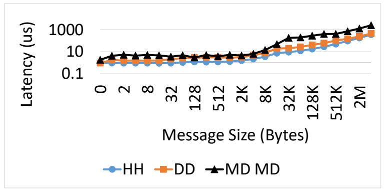


(a) Bandwidth MD MD

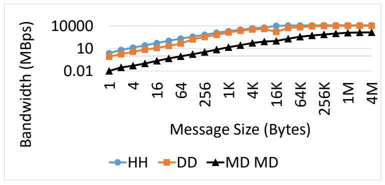


(b) Bi-bandwidth MD MD

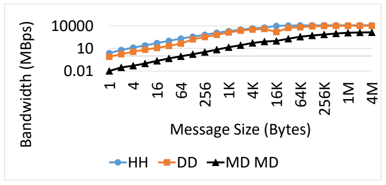
Fig. 6. x86 intra-node bandwidth and bi-bandwidth on MVAPICH2-GDR using OMB-UM



(a) Latency

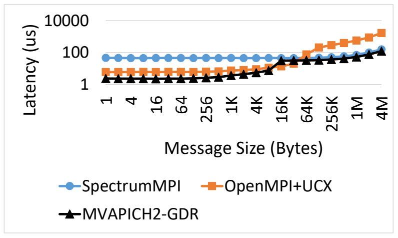


(b) Bandwidth

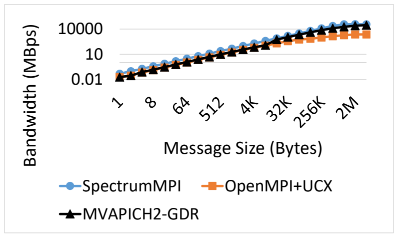


(c) Bi-Bandwidth

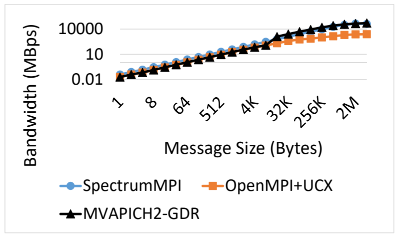
Fig. 7. x86 inter-node point to point benchmarks on MVAPICH2-GDR using OMB-UM



(a) Latency MD MD



(b) Bandwidth MD MD



(c) Bi-Bandwidth MD MD

Fig. 8. Comparison of intra-node point-to-point communications between CUDA-Aware MPI libraries using OMB-UM on POWER9 and Volta over NVLink2

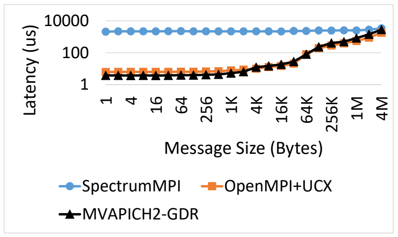
OMB-UM. Figure 10 depicts the latency of broadcast operation obtained by using MVAPICH2-GDR, OpenMPI+UCX, and SpectrumMPI across 6 GPUs within a POWER9 machine. Similar to point-to-point shown in Figure 8(a), MVAPICH2-GDR, and OpenMPI+UCX yield comparable performance for small message sizes (i.e., smaller than 16KB) and outperform SpectrumMPI by a magnitude of two. For large message sizes, MVAPICH2-GDR achieves up to 2X and 7.5X lower latency compared to SpectrumMPI and OpenMPI+UCX, respectively, due to the advanced IPC design for UM.

VI. Discussion

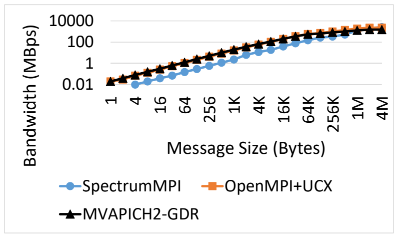
This section discusses some interesting insights obtained by using the OMB-UM benchmark on various CUDA-Aware MPI libraries such as MVAPICH2-GDR, SpectrumMPI and OpenMPI on both x86 and OpenPOWER architectures. These insights from OMB-UM benchmarks provide valuable information to identify the bugs and MPI design issues.

A. x86 Intra-node MH-MH latency bump in MVAPICH2-GDR

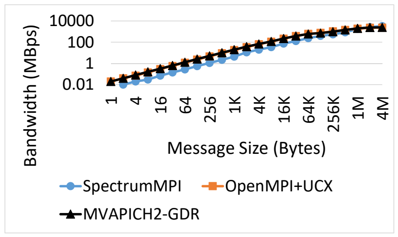
In Figure 5(a) which refers to the MH-MH latency curve, the effective location of the sender and receiver side UM buffer is set to host. Hence no kernel is launched either on the sender side or on the receiver side. Hence the MH-MH curve is expected to align well with pure host to host (HH) curve. But from Figure 5(a) we see that the MH-MH curve aligns well HH curve from small till medium message range. For large message range there is a sudden increase in latency. This is due to the activation of advanced IPC designs explained in Section III-C in MVAPICH2-GDR. This sudden increase in latency needs to be addressed to make the MH-MH curve align well with HH curve by intelligently activating and deactivating the advanced IPC designs.



(a) Latency MD MD



(b) Bandwidth MD MD



(c) Bi-Bandwidth MD MD

Fig. 9. Comparison of inter-node point-to-point communications between CUDA-Aware MPI libraries using OMB-UM on POWER9 and Volta over NVLink2

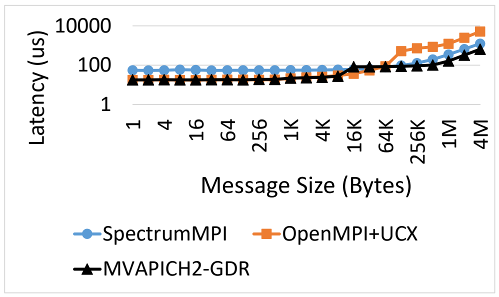


Fig. 10. Comparison of UM-based Broadcast operation between CUDA-Aware MPI libraries using OMB-UM on POWER9 and Volta over NVLink2

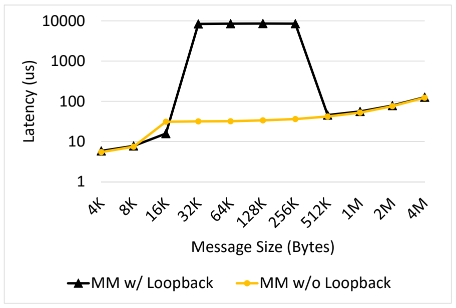


Fig. 11. Loopback issue with intra-node point to point communication in MVAPICH2 on OpenPOWER systems

B. *x86 Intra-node MD-MD large message bandwidth and bi-bandwidth benefited from MVAPICH2-GDR advanced IPC designs*

From Figure 6(a) and Figure 6(b) it is seen that even though bw and bibw are really low for managed buffers for small and medium messages, they are on par with host and device buffers for large message sizes. Thanks to the advanced IPC designs present in the MVAPICH2-GDR which provides these benefits. Another point worthy of mention is that these advanced IPC designs were developed during the Kepler era NVidia GPUs which had very preliminary support for managed memoey and those designs are still valid on the latest Volta and Pascal GPUs which boast advanced hardware features to support managed memory.

C. *x86 Intra-node and inter-node MD-MD small to medium message bandwidth and bi-bandwidth needs improvement in MVAPICH2-GDR*

From Figure 6(a), Figure 6(b), Figure 7(b) and Figure 7(c) we could see that the bw and bibw lag behind their host (HH) and device (DD) counterparts by an order of magnitude in MVAPICH2-GDR. This may be due to excessive moving of managed buffer between host and device but needs to be investigated further.

D. *OpenPOWER Intra-node point to point communication using MVAPICH2-GDR suffers from loopback design*

MVAPICH2-GDR employs high performance communication designs such as loopback [16] for communication within a node. The infiniband HCA adapter is involved in the communication. It receives the data from the sender and then it routes the data back to the receiver within the same node. This mechanism is normally exercised on host (HH) and device buffers (DD) in an intra-node setting. On the other hand when loopback is employed on managed buffers (MD-MD), performance degradation on the orders of magnitude is observed as shown in Figure 11. Further investigation needs to be done to identify the root cause of the degradation.

TABLE II
INSIGHTS ON POWER9 SYSTEMS WITH VOLTA GPUS

MPI Library	Comm type + bench	GPU Page Faults	CPU Page Faults
OpenMPI+UCX	Intra-node bibw	284557	295680
SpectrumMPI	Intra-node bibw	1248	—
SpectrumMPI	Inter-node latency	351864	390526
OpenMPI+UCX	Inter-node latency	70445	74020

E. *OpenPOWER Intra-node bibw on OpenMPI needs improvement*

Figure 8(c) shows that bi-bandwidth of OpenMPI is significantly lower than that of SpectrumMPI and MVAPICH2-GDR for large messages. Further profiling of both OpenMPI and SpectrumMPI shows that OpenMPI designs result in larger CPU and GPU page faults in the Vota GPUs resulting in poor performance. The page fault data are tabulated in the Table II. The design choices can be revisited in the OpenMPI designs to avoid unnecessary movement of managed data due to touching it either from host or device.

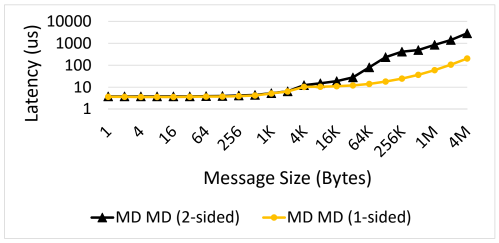


Fig. 12. Issue with two sided protocols in MVAPICH2-GDR for inter-node point to point communication on OpenPOWER systems

F. OpenPOWER Inter-node point to point communication in MVAPICH2-GDR suffers due to two sided protocols

MVAPICH2-GDR employs 2 sided protocols when communicating between managed buffers between a pair of nodes. But when the nodes are of type OpenPOWER significant degradation is seen as shown in the Figure 12. The degradation is alleviated when single sided protocols are employed in the place of 2 sided ones.

G. OpenPOWER Inter-node latency on Spectrum MPI needs improvement

Figure 9(a) shows that the latency of Spectrum MPI is on the order of magnitude worse than OpenMPI for small to medium messages. On further profiling it is found that the CPU and GPU page faults in Spectrum MPI are almost five times that of OpenMPI. Hence design choices in Spectrum MPI needs to be revisited to avoid the data migration of managed buffers thereby realizing high performance.

VII. Related Work

In [17], Chao et al. developed a test benchmark suite and nine applications to evaluate and compare the performances of pageable memory, pinned memory, and unified memory. In [18], the authors developed a UM benchmark in the context of OpenMP. They evaluated the performance of these benchmarks to identify where GPU memory management can be optimized. In [19], Jablin et al. developed a system for optimizing CPU-GPU communication. They evaluate the various issues with CPU-GPU communication and propose an automated approach to GPU memory management. While this work evaluates GPU data movement, the system does not consider UM.

IMB [20] is the benchmark developed by Intel to conduct performance and validation tests for all kinds of MPI communication APIs including MPI-1 functions, one-sided communications, MPI input/output (I/O), non-blocking (NBC) collectives and MPI3-RMA communications. MPICH test suite [21] provides the validation tests for MPI-1, MPI-2, and MPI-3 standard. Nevertheless, there is no performance measurement, and it is not GPU-aware. OMB [15], [5] provides performance measurement for all communication APIs, and it is GPU-aware and UM-Aware. However, UM support is naive and cannot evaluate the scenarios addressed in this paper.

Manian et al. [22] provided an in-depth characterization of UM-Aware MPI libraries on three generations of GPU architectures including Kepler K-80, Volta V100, and Pascal P100. OMB is modified to launch a sender side CUDA kernel similar to MD-MH to evaluate the UM-Aware MPI's performance. Manian et al. work lacks the options for all the four data placement options for setting UM buffer's effective location along with a full fledged benchmarking of UM buffer on various architectures and CUDA-Aware MPI libraries.

In [23], Knap et al. evaluate the performance of Unified Memory with data prefetching and memory oversubscription for various CUDA applications on Volta and Pascal GPUs to determine the influence of specific architectures on performance.

VIII. Conclusion

GPUs are becoming ubiquitous in the HPC world, and CUDA-aware MPI is one of the most preferred programming model to utilize the power of GPU clusters. The latest advancement in CUDA Unified Memory (UM) improves the productivity of the programmers by giving a unified memory space between CPU and GPU. Since the UM buffer can effectively lie either in the CPU or GPU, the micro-benchmarks written for CUDA-aware MPI such as OMB does not capture all the data placements possible for UM buffer. Thus this work proposes OMB-UM a UM-aware micro-benchmark that provides options for explicit data placement of UM buffers for point to point and collective operations. MPI designers can take advantage of new options provided by OMB-UM to fully characterize the UM-aware MPI libraries for finding insights to issues or for evaluating high performance designs. In addition to the design and implementation an in-depth performance evaluation of the proposed OMB-UM benchmark on a wide variety of systems and MPI libraries is provided. Furthermore, the insights gained by using the OMB-UM benchmarks on the issues found during the evaluation of various MPI libraries on multiple systems is provided to showcase the efficacy of the proposed benchmarks.

IX. Acknowledgements

This work was supported by the NSF under the ACI-1450440, ACI-1664137, CNS-1513120, and CCF-1565414 grants.

References

[1] J. A. Anderson, C. D. Lorenz, and A. Travesset, "General purpose molecular dynamics simulations fully implemented on graphics processing units," *J. Comput. Phys.*, vol. 227, no. 10, pp. 5342–5359, May 2008. [Online]. Available: http://dx.doi.org/10.1016/j.jcp.2008.01.047

[2] A. A. Awan, J. Bédorf, C.-H. Chu, H. Subramoni, and D. K. Panda, "Scalable Distributed DNN Training using TensorFlow and CUDA-Aware MPI: Characterization, Designs, and Performance Evaluation," in *The 19th Annual IEEE/ACM International Symposium in Cluster, Cloud, and Grid Computing (CCGRID 2019)*, 2019.

[3] A. A. Awan, K. Hamidouche, J. M. Hashmi, and D. K. Panda, "S-caffe: Co-designing mpi runtimes and caffe for scalable deep learning on modern gpu clusters," *SIGPLAN Not.*, vol. 52, no. 8, pp. 193–205, Jan. 2017. [Online]. Available: http://doi.acm.org/10.1145/3155284.3018769

[4] K. S. Khorassani, C.-H. Chu, H. Subramoni, and D. K. Panda, "Performance Evaluation of MPI Libraries on GPU-enabled OpenPOWER Architectures: Early Experiences," in *International Workshop on Open-POWER for HPC (IWOPH 19) at the 2019 ISC High Performance Conference*, 2018.

[5] D. Bureddy, H. Wang, A. Venkatesh, S. Potluri, and D. K. Panda, "OMB-GPU: A Micro-benchmark Suite for Evaluating MPI Libraries on GPU Clusters," in *EuroMPI'12*, 2012, pp. 110–120.

[6] K. Hamidouche, A. A. Awan, A. Venkatesh, and D. K. Panda, "CUDA M3: Designing Efficient CUDA Managed Memory-Aware MPI by Exploiting GDR and IPC," in *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*, Dec 2016, pp. 52–61.

[7] NVIDIA, "Whitepaper: NVIDIA Tesla P100, section 'Unified Memory'," 2016, Accessed: October 18, 2019. [Online]. Available: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

[8] N. Sakharnykh, "Everything You Need to Know About Unified Memory," March 2018. [Online]. Available: http://on-demand.gputechconf.com/gtc/2018/presentation/s8430-everything-you-need-to-know-about-unified-memory.pdf

[9] H. Wang, S. Potluri, D. Bureddy, C. Rosales, and D. K. Panda, "GPU-Aware MPI on RDMA-Enabled Clusters: Design, Implementation and Evaluation," *IEEE Transactions on Parallel and Distributed Systems*, Oct 2014.

[10] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, 9 2004, pp. 97–104.

[11] Network-Based Computing Laboratory, The Ohio State University, "MVAPICH: MPI over InfiniBand, Omni-Path, Ethernet/iWARP, and RoCE," 2001, Accessed: October 18, 2019. [Online]. Available: http://mvapich.cse.ohio-state.edu/

[12] Jiri Kraus, "An Introduction to CUDA-Aware MPI," March 2013, Accessed: October 18, 2019. [Online]. Available: https://devblogs.nvidia.com/introduction-cuda-aware-mpi/

[13] NVIDIA, "NVIDIA GPUDirect," March 2011, Accessed: October 18, 2019. [Online]. Available: https://developer.nvidia.com/gpudirect

[14] D. S. Banerjee, K. Hamidouche, and D. K. Panda, "Designing High Performance Communication Runtime for GPU Managed Memory: Early Experiences," in *Proceedings of the 9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit*, ser. GPGPU '16. New York, NY, USA: ACM, 2016, pp. 82–91. [Online]. Available: http://doi.acm.org/10.1145/2884045.2884050

[15] Network Based Computing Laboratory, "OSU Micro-benchmarks," 2019, Accessed: October 18, 2019. [Online]. Available: http://mvapich.cse.ohio-state.edu/benchmarks/

[16] R. Shi, S. Potluri, K. Hamidouche, J. Perkins, M. Li, D. Rossetti, and D. K. 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)*, Dec 2014, pp. 1–10.

[17] C. Liu, J. Bhimani, and M. Leeser, "Using high level gpu tasks to explore memory and communications options on heterogeneous platforms," in *Proceedings of the 2017 Workshop on Software Engineering Methods for Parallel and High Performance Applications*, ser. SEM4HPC '17. New York, NY, USA: ACM, 2017, pp. 21–28. [Online]. Available: http://doi.acm.org/10.1145/3085158.3086160

[18] A. Mishra, L. Li, M. Kong, H. Finkel, and B. Chapman, "Benchmarking and Evaluating Unified Memory for OpenMP GPU Offloading," in *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC*. New York, NY, USA: ACM, 2017, pp. 6:1–6:10. [Online]. Available: http://doi.acm.org/10.1145/3148173.3148184

[19] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August, "Automatic CPU-GPU Communication Management and Optimization," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 142–151. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993516

[20] "Intel MPI Benchmarks (IMB)," https://software.intel.com/en-us/articles/intel-mpi-benchmarks, 2017, [Accessed: October 18, 2019].

[21] Argonne National Laboratory, "MPI Test Suites," http://www.mcs.anl.gov/research/projects/mpi/mpi-test/tsuite.html, 2017, [Accessed: October 18, 2019].

[22] K. V. Manian, A. A. Ammar, A. Ruhela, C.-H. Chu, H. Subramoni, and D. K. Panda, "Characterizing CUDA Unified Memory (UM)-Aware MPI Designs on Modern GPU Architectures," in *Proceedings of the 12th Workshop on General Purpose Processing Using GPUs*, ser. GPGPU '19. New York, NY, USA: ACM, 2019, pp. 43–52. [Online]. Available: http://doi.acm.org/10.1145/3300053.3319419

[23] M. Knap and P. Czarnul, "Performance evaluation of unified memory with prefetching and oversubscription for selected parallel cuda applications on nvidia pascal and volta gpus," *The Journal of Supercomputing*, Aug 2019. [Online]. Available: https://doi.org/10.1007/s11227-019-02966-8