Designing High Performance Communication Runtime for GPU Managed Memory: Early Experiences

Dip Sankar Banerjee

Khaled Hamidouche

Dhabaleswar K. (DK) Panda

Network-Based Computing Laboratory
The Ohio State University
Columbus, OH 43210

{banerjee.131, hamidouche.2, panda.2}@osu.edu

ABSTRACT

Graphics Processing Units (GPUs) have gained the position of a main stream accelerator due to its low power footprint and massive parallelism. CUDA 6.0 onward, NVIDIA has introduced the Managed Memory capability which unifies the host and device memory allocations into a single allocation and removes the requirement for explicit memory transfers between either memories. Several applications particularly of irregular nature can have immense benefits from managed memory because of the high productivity in programming that can be achieved owing to the minimal effort involved in the data management and movement. The MVA-PICH2 library utilizes runtime designs such as CUDA Inter Process Communications (IPC) and GPUDirect RDMA (GDR) under the CUDA-Aware concept, to offer high productivity and programmability with MPI on modern clusters. However, integration and interaction of managed memory with these features raises challenges for efficient small and large message communications.

In this study, we present an initial evaluation of managed memory capability and its interaction with existing high performance designs and features available in MVA-PICH2 library. We propose new designs to enable efficient communication support between managed memory buffers. We also perform fine tuning to optimize the transfers between managed memories residing in GPUs. To the best of our knowledge, this is the first evaluation and study of managed memory and its interaction with MPI runtimes. A detailed evaluation and analysis of the performance of the proposed designs is presented. The Stencil2D communication kernel available in the SHOC suite was re-designed to enable the managed memory support. The evaluation shows a 4x improvement in the timings of stencil exchanges on 16 GPU nodes.

1. INTRODUCTION

In the last decade, High Performance Computing has undergone a major revolution with the arrival of massively multi-threaded devices such as GPUs. Owing to the high FLOPS to Watt ratio, GPUs have established themselves as one of the dominant PCIe based accelerators in modern generation clusters. The introduction of commodity high performance interconnects such as InfiniBand, it has become possible to design applications that make use of the raw GPU compute power with greater ease. Programming models such as CUDA have enabled the developers to harness the compute power of GPUs.

Traditionally, the Compute Unified Device Architecture (CUDA) has been the standard programming library used for GPUs. CUDA has evolved massively since the early G80 generation of GPUs to provide higher programmability and efficiency.

CUDA 6.0 onward NVIDIA introduced one of the most dramatic improvements to the CUDA programming model named Managed Memory. It basically creates a pool of memory which is not disjoint between the GPU and the CPU. Also, this memory can be uniquely addressed by both the devices. Managed memory basically removes the added complexity of explicitly allocating memory in either the CPU or in the GPU followed by copying of the data using the popular `cudaMemcpy()` call. This feature provides significant productivity benefits in programming such as doing away with the explicit allocations and transfers. These are now essentially abstracted and the CUDA runtime ensures the locality of the data in the device from which it is touched. This ensures higher code productivity as the users create and manage only one instance of the pointer and treat it as a traditional C pointer.

Since CUDA 4.0 onward, MPI runtimes such as MVA-PICH2 [1] and OpenMPI [3], have been enhanced with the CUDA-Aware concept to allow the runtime to directly and transparently access and send/receive data from/to GPU memories using techniques like CUDA IPC and GDR. CUDA awareness of MPI provides additional productivity to the application developers since they do not have to explicitly perform any copies between the device and the host before performing the MPI Send/Recv operations from the host. These buffers can be directly used for the communications and the underlying runtime will ensure efficient delivery of the buffer. We propose to integrate CUDA managed memory into the MPI runtime so that the runtime can transparently perform data transfers of buffers allocated as managed memories. MPI being the most widely used library for

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

GPGPU-9, March 12-16, 2016, Barcelona, Spain

© 2016 ACM. ISBN 978-1-4503-4195-0/16/03...\$15.00

DOI: <http://dx.doi.org/10.1145/2884045.2884050>

distributed computing, this feature will enable the users to exploit higher productivity in two dimensions. In the first place, using buffers allocated as managed will remove the necessity for explicit intra-node transfers and using managed aware MPI will enable direct transfer of the buffers in an efficient manner.

In Figure 1, we present the design space for CUDA managed memory. In the figure, we broadly present the design alternatives that can be done for CUDA managed memory and their trade offs with the three broad aspects of high performance computing namely, performance, scalability and productivity. The first alternative is the stand alone use of managed memory (shown in black broken line) which is provided by CUDA for single node applications. In this case disjoint device and host memory spaces can be integrated using the Unified addressing provided by the CUDA driver for highly productive application development. This however is not designed for scaling beyond a single node. On a single node it provides good productivity and reasonable performance. It can be used in a distributed setting through non GPU-aware MPI which will provide scalability to some extent but with performance that will be worse than the stand alone use.

In modern day heterogeneous clusters, it is important to have efficient runtimes that will transparently send and receive data across multiple accelerated nodes. Towards this, we already have CUDA aware MPI implementations such as MVAPICH2 and OpenMPI which allows the users to perform communications on buffers residing on GPU global memories without any additional copy to host. CUDA aware MPI (shown in red broken line) provides very high performance, scalability and productivity. With the advent of managed memory, higher amount of productivity can be achieved due to the unified address space of device and host.

The third scheme that we propose in this work (shown in blue broken line) will provide a CUDA managed memory aware MPI which will provide good scalability and higher productivity. However, there will be a performance degradation due to the fact that the CUDA driver is not maintaining the GPU pages on the device for a sufficiently long time which is necessary to facilitate irregular accesses and RDMA transfers over the underlying interconnects.

Figure 1: Design space of CUDA managed memory in high performance runtimes

Features of CUDA such as non-blocking kernel calls, asynchronous memory copies and concurrent kernels on a single GPU have paved the way for heterogeneous multicore processing where an application can exploit the compute power

of the GPU as well as the significantly powerful CPU. In addition to the generic CUDA APIs, auxiliary features such as GPUDirect help expedite data transfers to/from GPU memory. GPUDirect is a set of features that enable efficient data movement among GPUs as well as between GPUs and peer PCI Express (PCIe) devices. CUDA 5.0 introduced the GPUDirect RDMA (GDR) feature, which allows InfiniBand network adapters to directly read from or write to GPU device memory while completely bypassing the host [11]. However, performing explicit copies of data to the GPU main memory and the final copy back to the host is a necessary step that had to be performed due to the disjoint address spaces of the two devices. For applications with irregular memory accesses, this issue becomes more severe due to the extra synchronization overheads that are incurred in-between the intermediate transfers. So a model which explicitly handles the memory transfers will provide additional help in fine tuning such applications for better performance.

1.1 Motivation

The Managed Memory feature promises a significant impact on the programmability and productivity of the application on GPU nodes as highlighted earlier. However, scientific applications are not pure CUDA applications but rather written with a hybrid programming model that combines CUDA and a high level parallel programming model such as MPI. In some of the earlier works [11, 15], the authors have shown that it is possible to design MPI runtime to exchange data that is residing on the GPU memories in an efficient manner by achieving very low latencies and high bandwidth. The designs are exploiting interesting features such as GDR and CUDA IPC to provide highly efficient communication. This leads us the broad questions: **Can CUDA-Aware MPI runtimes efficiently and transparently take advantage of Managed Memory to provide the productivity for end-users?**

In this paper, we tackle this broad question and discuss various challenges in designing and integrating support to managed memory in the MVAPICH2 library. In particular, we address the following challenges as a part of this work:

- What are the characteristics and limitations of CUDA managed memory on current generation InfiniBand clusters?
- How can managed memory be integrated with a standard MPI library, such as MVAPICH2, so as to efficiently transfer messages that are allocated using `cudaMallocManaged()`?
- What is the nature of interaction of buffers allocated in the managed memory with existing technologies such as IPC and GDR? Is it possible to leverage the existing solutions for creating efficient designs for communication of managed regions as well?
- How does the performance of managed memory compare against the current state of the art solutions for transfers between GPU and host buffers?
- What is the impact on performance of end applications that uses managed memory?

As part of this work, we propose new designs for MPI communications to support managed memory allocation and perform experimental validations to show the viability of those

designs. To the best of our knowledge, this is the first work that studies the impact of CUDA managed memory and its interaction with CUDA-Aware MPI calls. We compare the performance of our results with the existing designs in MVAPICH2. Towards this, we first perform MPI point-to-point communication tests using MPLSend and MPLRecv. We see that using the only possible mechanism for transferring managed data, there is a degradation in performance of about 60% in comparison to messages that are explicitly allocated on the GPUs using cudaMalloc(). The experiments are performed using Kepler K20 GPUs and Mellanox FDR adapter. We also carry out performance evaluation using two benchmarks which are popularly used in several high performance applications. Stencil2D which performs stencil updates followed by symmetric exchange of data is re-designed using managed memory for all allocations and we see that there is a performance gain of approximately 87%. Along these lines we also implement a Stencil3D application which achieves reduction in communication time of up to 75%.

Another contribution of this paper is the redesign of OSU Micro-benchmarks (OMB) suite with the managed memory concept. The existing design of OMB allows benchmarking of different point-to-point and collective operations from/to GPU buffers for the different configurations: Device-to-Device (D-D), Device-to-Host (D-H) and all other possible combinations (H-D, H-H). As a part of this extension, we introduce a new MH configuration. A detailed description and analysis of the new proposed configurations and their interactions with the existing ones are presented in Section 4.

2. BACKGROUND

In this section we will discuss the appropriate background for this work. We shall discuss the GPU architecture, InfiniBand architecture, MVAPICH2 library for GPU based communications and the CUDA managed memory in detail.

2.1 CUDA and GPU Architectures

CUDA is a C based programming library that is provided by NVIDIA for easy application and runtime development for GPUs. In this work we use CUDA version 6.5 for design and evaluation of managed memory. GPUs are co-processors which are attached as peripheral devices on a I/O hub (such as PCI express). There has been several significant developments in the CUDA programming model that has enabled efficient use of GPUs in high performance clusters. One such was the introduction of the CUDA Unified Virtual Addressing feature in CUDA 4.1 [2] which unified address spaces of the GPU and host (CPU) memories in a single node and allowed peer to peer communications between two GPUs without the necessity of going through the host. The CUDA Unified Memory was introduced in CUDA 6.0 [5] which created a single unified address space that removed the necessity of explicitly copying data to and from GPU memory. One limitation of GPU managed memory is that it is not supported when two GPUs are residing on two different I/O hubs in which case the data still has to be explicitly copied through the host memory.

2.2 MVAPICH2 for GPU Clusters

Message Passing Interface (MPI) is the de-facto standard for application development in high performance clusters. MVAPICH2 [1] is a popular open source implementation of MPI over InfiniBand, 10 Gigabit Ethernet/iWARP and

RDMA over converged ethernet (RoCE). MVAPICH2 provides support for transfers between GPU buffers. Several designs inside MVAPICH2 provide efficient transfer mechanisms for point-to-point transfers between GPUs through standard MPI semantics. This design for inter-node communications was first proposed in [16] where the implementation took advantage of the CUDA Unified Virtual Addressing technology that was offered in CUDA 4.0.

MVAPICH2-GDR provides support for GPU based communications through the GPUDirect technology. GPUDirect technology was introduced by NVIDIA since CUDA 4.0 which enabled host regions to be registered by both a network adapter and a GPU device. Registration of memory regions allowed the DMA engines of the two devices residing on PCIe to move data in and out of the registered regions. GPUDirect has also allowed third party devices to directly read and write on the registered locations thus allowing the evolution of GPUDirect RDMA (GDR). It also provides multiple solutions for GPU-GPU intra node transfers which uses shared memory and CUDA Inter Process Communications [12].

MVAPICH2-GDR unified movement of data to and from GPU memories with the standard MPI semantics. The design first proposed in [11] achieves this through the use of Unified Virtual Addressing. MVAPICH2 provides optimized inter-node transfers by pipelining the transfers from GPU to host memory, host to remote memory, and remote memory to remote GPU. These designs were improved in [14] to provide memory transfers by bypassing the host and transferring data directly from the source GPU memory to the destination GPU memory.

Figure 2: Communication Paths between GPUs and between GPU and Host

In Figure 2, we see the different communication paths that can be taken for internode transfers. The path shown in red color shows the path that is taken when implementing the GDR based transfers. One caveat of this design is that there is a strict requirement to register the GPU memory region with the InfiniBand adapter before the transfer can

be initiated. While using the managed memory, performing this registration is not possible as the data pointer cannot be uniquely identified to be residing in either the host or the device. Leading to this we have to investigate the possibility of exchanging the data through the other paths which are shown in Figure 2.

2.3 CUDA Managed Memory

One of the critical aspects of GPU programming using CUDA is the requirement of moving data in and out of GPU memory. Since the introduction of CUDA, this has been the standard practice because the GPU and host memories were disjoint. Since CUDA 6.0, NVIDIA introduced the Unified Memory (UM) feature which removes this explicit requirement, making application development much easier along with higher code productivity. UM unifies a memory pool that is shared between the CPU and the GPU and provides a single pointer that can uniquely identify the host or device and is internally moved. This memory pool is called "managed memory" and is allocated using `cudaMallocManaged()` without the need of using the conventional `malloc()` for host memory or `cudaMalloc()` for device memory allocations. Managed memory hence provides big benefits from the point of view of code development time, quality, effort and cost. More specifically, it provides benefits from the point of development of irregular applications which requires frequent movement of data in and out of the GPU.

However, inside the CUDA runtime the implementation of managed memory is still in a naive state as the unification of heterogeneous memories is still an open challenge. Bottlenecks in terms of memory registration essentially prevent the use of existing designs in MVAPICH2 for integration of managed memory and extracting high performance. In this paper we have optimized designed for these bottlenecks so as to efficiently use managed memory in the standard MPI semantics. These are discussed in the next section.

3. PROPOSED DESIGNS

In this section we are going to discuss the different designs proposed to efficiently implement and integrate CUDA managed memory into the CUDA-Aware MVAPICH2 library. The main objective behind the design is to provide efficient and transparent support for managed memory through the standard MPI semantics. From the point of view of application development it should be possible to perform send/receive of data that has been allocated using managed memory without the requirement of any additional code.

3.1 CUDA-Aware Managed memory allocation

One of the fundamental basis of the CUDA-Aware concept is based on the Unified Virtual Addressing (UVA) feature available since CUDA 4.0 Using UVA, the runtime can dynamically identify if a buffer is allocated on the host (`malloc`) or on the GPU (`cudaMalloc`). The internal handling at the CUDA driver level of memories allocated as managed is closed. Hence, it is not possible for examining how the memory pages are handled or moved by the CUDA driver.

In order to support managed memory under the umbrella of CUDA-Aware concept, this detection logic needs to be extended to handle the managed memory allocation through the `cudaMallocManaged`. The `cudaGetPointerAttributes()` call can return both Host or Device. For the `Type` attribute, we use the same call with `is_managed` attribute to under-

stand if the buffer is allocated with managed memory or not and return the appropriate value. Further we also need to pay attention to the case that managed memory buffers can co-exist with the standard host and/or device allocations. With this extension, depending on the memory allocation MVAPICH2 automatically detects if it is a host buffer (0), a device buffer (1) or a managed buffer (2).

3.2 Interaction of Managed memory with CUDA IPC and/or GDR

The current support of managed memory in the latest CUDA 7.5 release does not allow the memory to be pinned. In other words, a managed memory cannot be registered with any device. The limitation is inherited from the core concept of managed memory itself. Indeed, managed memory allows the memory pages to move between host and GPU dynamically as needed. For instance, when the user starts a CUDA kernel the pages are dynamically swapped out from host pages and swapped into the GPU pages. Due to this page swapping the managed memory cannot be pinned.

As a consequence of the inability to pin memory, neither CUDA Inter Process Communication (IPC) nor GDR can be used with the managed memory. IPC requires the pinning of the memory with the GPU and GDR requires the memory to be registered with the IB Host Control Adapter (HCA). Thus we expect a significant degradation in performance for small message with internode communication where GDR is used. This can also be expected for intranode communications involving large message sizes where IPC delivers good performance.

3.3 Design for MPI point-to-point communication

It is a common practice to adopt different strategies to exchange messages according to their sizes. The reason for this being that while sending small messages often incur lesser memory overheads, the larger messages create a higher amount of overhead in terms of control messages that are required to be exchanged before the actual data buffer could be sent. It is observed in the most popular MPI implementations that small messages use the standard Eager protocol while larger messages are handled using the Rendezvous protocol. One of the things noteworthy is the fundamental difference between these protocols. The sending of smaller messages is mostly dominated by the copy between the user buffers and the runtime internal buffers. Whereas large messages are limited by the control message exchange. This behavior can be observed in the GPU communications as well.

3.3.1 Short Message Communication

Taking advantage of the fact that each time a managed memory buffer is touched by the CPU, the CUDA driver will automatically bring the data from the GPU to the host, we enhanced the eager protocol of MVAPICH2 for small message managed memory transfers. When MVAPICH2 detects a managed memory buffer, it first touches the buffer from the host which enables the CUDA driver to swap in the pages to the host. Once the data is on the host the runtime follow the original host based eager protocol. In order to reduce the overhead of touching all the data on host by an extra loop of dummy reads, we directly use the memory between the managed memory and the MVAPICH2 internal eager buffers (known as Vbuf) to emulate the touching procedure as the

memcpy is performed by the CPU.

Note that the same protocol is been used for intranode and as well as internode communication. The only difference is the channel used to transfer the data form sender's Vbuf to receiver's Vbuf. For intranode, it uses the shared memory channel whereas for the internode it uses the IB channel.

Using device buffers MVAPICH2 ensures the complete path for data movement from device buffer at the sender to the device buffer at the receiver. With managed memory, the runtime ensures that the data has reached the host's destination managed memory. However, it does not ensure the final destination buffer. In other words, with managed memory a device-to-device (D-D) transfer is equivalent to a device-to-host (D-H) from the runtime point of view. This problem is however simplified by the CUDA runtime which ensures that the data resides in the GPU global memory when the end application touches it through a kernel invocation or simply accessing from the GPU.

3.3.2 Large Message Communication

For the large messages, the MVAPICH2 runtime uses a handshaking based approach to implement the rendezvous protocol and an RDMA operation which requires the memory registration. In the advanced GDR design the data is communicated directly from the host sender (after staging from GPU) to the destination GPU without the involvement of the receiver host memory. Since the data pointer of managed memory cannot be registered, we propose an alternative design for sending large messages using a more reliable extension of the eager protocol.

In the first step there is a Request to Send (RTS) message that is sent to the receiver process with information about the data buffer that is about to be exchanged including the type of buffer (managed) and the protocol to be used. The receiver process on the other hand sends back a reply with the information of its intermediate buffers using a Clear To Send (CTS) message. At the end of this step both the sender and receiver have agreed on the rendezvous protocol to be used.

Based on the fact that we cannot register the application buffers, we fall back to use intermediate buffers (Vbuf) similar to the eager protocol. However, unlike the eager protocol, copying large messages from the user buffer to the Vbufs can be expensive. To avoid the expensive copy of large message sizes we propose a pipeline design based on a packetization scheme. The buffer is packetized into smaller sized chunks that can fit into these intermediate buffers. Once the packetization is done, the transfer is performed in a pipeline manner as shown in Figure 3. Here we can see that to hide the overhead of the copies, we overlap them with the IB transfers. As the verbs API are non-blocking, we can easily overlap the transfers between the intermediate buffers in the two processes. This offers good advantages over the non-overlapped design. Once all packets have completed transfers, the receiver responds with a Finish (FIN) message.

3.4 Design for MPI Collectives

MPI collectives such as alltoall, allgather, allreduce are advanced operations which are implemented over MP_Send/Recv operations. Since the managed memory feature offered by NVIDIA is still at a nascent stage we design the collectives using the naive send receive mechanism that we discussed in the previous designs. With the future advances of the managed memory technology, we will be able to implement more

Figure 3: Message pipelining between the different transfers

efficient designs dedicated towards efficient MPI collectives. The current implementation of CUDA aware MVAPICH2 allows one GPU per process. In accordance, all the results provided towards evaluation are performed with one GPU per process.

4. EXTENDING OSU MICRO-BENCHMARKS

We choose the latency and bandwidth tests which are present in the OSU Micro-benchmark (OMB) suite and perform certain extensions to the existing OMB tests for GPU. The existing features in OMB allow to experiment on point-to-point communications using standard MPI semantics where we can send and receive data directly from the GPU memory. For using managed memory, the buffers allocated using `cudaMalloc()` now needs to be allocated using `cudaMallocManaged()`. Hence, we added this functionality inside OMB where we can mention explicitly when running OMB tests that the GPU buffers be allocated using managed memory. The device to device transfers denoted by D-D in the following figures show the allocations done using conventional `cudaMalloc()` while the transfers denoted by MH-MH denote the transfer of buffers that are allocated using `cudaMallocManaged()`.

It is important at this stage to clearly distinguish between the managed memory that ensures that the data lies in the host or the data lies in the GPU. In managed memory, the localization of the data pointer that has been allocated through `cudaMallocManaged()` is not trivial as in the case of `cudaMalloc()`. The CUDA runtime states that for any data buffer that is allocated through managed memory resides in either the host or device depending on the location from where the data buffer was touched. If it is touched from a GPU kernel, the runtime performs an implicit `cudaMemcpy()` in order to ensure that the data is residing on the GPU when the latency test starts. In a similar way when the data is accessed from the host, the data is moved from GPU to host in case it was not residing there already. From this perspective, it is important to ensure that the data resides in the said location while performing the point-to-point tests.

Towards this, we envisioned the MD-MD transfer scenario which will require a call to a dummy kernel that touches the GPU buffer on the device side and ensures the data is on the device before it is sent out. This however is not feasible due to the fact that in every subsequent call to the kernel there is a necessity to do a synchronization so that the data buffer can be re-used. This synchronization call makes the CUDA driver to move the data to the host in every call due to which we cannot ensure that the data buffer is residing on the GPU global memory. This fact was also evident from

the subsequent experimental analysis which showed that the data was in fact sent out from the host and the results were not reflecting the time required for a `cudaMemcpy` that will be needed to move the data from the GPU to the host before being pushed out over the network.

We do a test between two managed allocations done on host (MH-MH) using a similar strategy where we ensure that the data buffer is accessed (through a `memset` operation) from the host before the send/receive is initiated.

From the point of view of end applications, often this might be the case that there are consecutive `MPI_Send/Isend` operations that take place from the same location be it on the device or on the host. In case the data is currently located on the host and there are successive send operations performed on the same device buffer, then there will be no requirement to copy the same data from the GPU device to the host multiple number of times. In order to monitor such kind of occurrences, we take a similar approach while re-designing the bandwidth test of OMB. We first initiate a transfer from the GPU which is not touched again from the GPU in the successive calls. This is unlike the strategy that we adopt for the latency tests where before every send, the buffer is accessed at least once from the GPU through a dummy kernel so as to ensure that the data resides on the GPU. These two scenarios give a better understanding of the performance tradeoffs that an application may have to consider before using managed memory.

5. EXPERIMENTAL EVALUATION

In this section we provide the experimental results of integration of managed memory under the CUDA-Aware concept. Towards this we use the extended OSU Micro-benchmark suite to evaluate point-to-point performance along with performance with an end application communication kernel.

5.1 Experimental Setup

For experimentation we used the Wilkes cluster deployed at University of Cambridge. Wilkes is a GPU cluster where there is support for GPUDirect RDMA. We used two nodes in the cluster which are composed of Intel Ivybridge processors. Each node has 12 Xeon E5-2630 2.60 GHz cores which is 2 processors having 6 cores each in two sockets. The Wilkes cluster, which was deployed in November 2013, is one of the fastest clusters in the United Kingdom and achieves 230 TF performance for High Performance Linpack. A subset of the nodes in the Wilkes cluster have two K20 GPUs in each node that are connected through FDR IBs. The K20 GPUs have a global device memory of 12 GB while the entire node has 64 GB main memory.

5.2 Evaluation of point-to-point communications

We now evaluate the performance of the GPU proposed design through the point-to-point communication benchmarks. Before we move to the discussion on the results in details we re-iterate the strategy which we discussed in Section 3.1. While performing the latency tests using OMB, we test three different buffers using managed allocations. The transfers between managed buffers residing on device (using a dummy kernel), managed buffers on host (MH) and conventional allocations done using `cudaMalloc()` (D) and `malloc` (H). For MH buffers, we touch the data from the host using a `memset` operation. Another evaluation that we do is that of latency

performance with a dummy kernel call. This is analogous to the MD configuration that we had discussed in Section 4. We create a dummy buffer, and call that before each of the data buffers are pushed out over the network. This way, the results reflect the extra time that it will need to copy the data from the GPU to the host memory and a synchronization time.

In Figure 4 we see that latency performance for small messages. We see that there is a performance degradation in the communication times by almost 28% on an average over the H-H communications. In Figure 5, we see the performance of the medium to large sized messages shows an effective degradation upto 21%. The bandwidth performances are shown in Figure 6 and Figure 7 for small and large message sizes, respectively. We observe that there is a degradation of around 58% overall against the H-H buffers. Additionally we perform the experiments to observe the impact of `cudaMemcpy()` on the overall performance. In Figure 5(b) and Figure 7(b) we see that there is a significant overhead that is imposed by a dummy kernel call. On an average we see a 55% overhead for latency and 68% overhead for bandwidth performance. This is due to the kernel calls that are put before the data buffers are sent out through the network. Also, there is a kernel call that is put after the receive so as to mimic the case where the buffer will be residing on the GPU memory at the destination. This additional two copies and subsequent stream synchronizations lead to the overhead that is observed.

5.3 Evaluation of Collectives

As the MPI collective operations are built on top of the MPI point-to-point calls, we now perform the evaluation of the collective operations using the similar configurations that we presented for point-to-point. In this case, we broadly experiment with the two buffers where one is on the host that is allocated using the CUDA managed memory allocation. We also add a configuration with a dummy kernel call that will reflect the overhead for an additional memory copy from the GPU and a required synchronization. As mentioned earlier, we experiment with each process running on one GPU and the results show similar trends with number of processes varying from 2 to 128.

In Figures 8 and 9, we have presented the results for some of the widely used MPI collectives. We observe that on an average there is an additional 45% time that is required for the dummy kernel call and synchronization. Overall the collectives perform 58% worse than the pure GPU buffers residing on the device. The broad cause for this degradation can be attributed to the data buffers being exchanged over the InfiniBand send/recv protocol and the inability to use the RDMA semantics. The RDMA semantics are not possible to be used due to the fact that the memory pages allocated using the CUDA managed memory is not pinnable to a particular memory.

5.4 Evaluation of Stencil Kernels

Stencil kernels are popular benchmarks that are often used in parallel computing in order to measure the performance of data exchanges. Stencils are a class of iterative kernels that are commonly used for solving partial differential equations, in image processing and geometric modelling. As a part of this work we evaluate two stencil computations which are done in two and three dimension respectively. The motiva-

(a) Small Messages MH-MH vs H-H

(b) Small Messages MH-MH with dummy kernel and without

Figure 4: Comparison of latency performance for small messages for different OMB configurations on Wilkes

(a) Large Messages MH-MH vs H-H

(b) Large Messages MH-MH with dummy kernel and without

Figure 5: Comparison of latency performance for medium and large messages for different OMB configurations on Wilkes

(a) Small Messages MH-MH vs H-H

(b) Small Messages MH-MH with dummy kernel and without

Figure 6: Comparison of bandwidth for small messages for different OMB configurations on Wilkes

Figure 7: Comparison of bandwidth performance for medium and large messages for different OMB configurations on Wilkes

Figure 8: Comparison of collectives with managed memory buffers run using 16 nodes on Wilkes

Figure 9: Variation of number of GPU nodes for Alltoall from 2 to 16 nodes on Wilkes

tion being that it will enable us to observe the behavior of the managed memory in CUDA when the computations are performed across GPU buffers.

We have redesigned the Stencil benchmark available in the SHOC suite [4], to enable the usage of Managed memory. The evaluation includes Stencil2D and Stencil3D using different message sizes and different system sizes ranging from 4 to 16 nodes.

Figure 10 shows that in addition to the productivity benefits, managed memory delivers better performance than the CUDA-Aware version. The managed version reduces the execution time of the Stencil2D by a factor of 4x on average. The reasons behind these benefits are directly related to the communication pattern in the benchmarks. In this stencil, the same buffer at the sender is used for the communication with all the neighbors. Thus in managed version, CUDA driver copies the data from the device to the host only in the first transfer and using the host version for other sends. In other words, in managed version only the first send is from device buffer and the rest of the sends are from the host. In the CUDA-Aware version all the transfers take place from GPU buffers. The stencil applications have negligible use of collective operations.

The results of Stencil3D are shown in Figure 11. We evaluate this application by varying the number of elements that are exchanged in three dimensions. The results shown in Figures 11(a), 11(b), and 11(c) show the results for variations along X,Y and Z dimensions, respectively. We see a benefit in performance by up to 65% when averaged over the benefits gained over all the three dimensions. In this case too we can attribute the advantages to the fact that there is only a single transfer that takes place from the GPU to the host which is then reused in the following iterations.

6. RELATED WORK

The use of accelerators in high performance clusters is becoming a common practice in today's world. Given the advantages that accelerators offer in terms of raw compute power and power efficiency, they have found usage in significant clusters that are featured in the Top500 list. GPUs have always been a shared memory processor with the global memory forming the core communications medium for the thread blocks.

One of the early works of the use of GPUs in MPI calls was proposed by Miyoshi et al. [10] where the authors proposed embedding of MPI calls inside GPU kernels. The first work to demonstrate the use of GPUDirect RDMA was proposed by Potluri et al. in [11]. In [11], the authors showed how GPUDirect RDMA can offer high performance along with good amount of programming productivity. This work was targeted at optimizing small message transfers and was further extended by Shi et al. in [14] where the authors showed how some of the new techniques such as NIC loopback and Fastcopy could enable faster transfer of eager messages with higher performance. In a recent work done by Landaverde et al. in [8], the authors have done a performance evaluation of the CUDA managed memory from an applications perspective. The authors state that even though the programming productivity is high due to the on-demand fetching of data, the performance of managed memory is poor which severely restricts its flexibility and adding future optimizations. In all of the current work, there has not been any studies pertaining to the study of CUDA managed mem-

ory on distributed systems. Managed memory offers exciting opportunities from the applications development stand point due to its high productivity and scalability. We distinguish our work from the related efforts by providing a solution for using managed memory across distributed memory systems that are commonly present in modern high performance clusters. We performed studies on GPU managed memory which benefit through the host assisted communications.

Another direction in accelerator based MPI communications is the optimization of MPI derived data types with GPU based packing and unpacking. This work was done by Shi et al. in [13]. The clear advantages obtained in the work was evaluated using the same stencil computations that we have used in our work for the evaluation of the managed memory kernels. Stencil computations using accelerators was shown in [9, 6, 7]. In [6], the authors proposed a generic library for stencil computations through the optimization of targeted nested vector and subarray packing/unpacking routines at the application level.

7. CONCLUSIONS AND FUTURE WORK

In this work we performed a preliminary analysis of the CUDA managed memory feature and studied its interaction with the CUDA-Aware MPI runtimes. We proposed novel designs that integrate and enable the support of managed memory into the CUDA-Aware concept in the MVAPICH2 MPI library. Even though managed memory offers good productivity in programming, the current support in CUDA 7.5 suffers in performance due to its inherent limitation due to its limited usability with IPC and GPUDirect RDMA. Despite these limitations, for stencil communication patterns where the same buffer is transferred to different processes, our design of the managed memory shows an improvement by a factor of 4X on 16 GPU nodes.

With upcoming CUDA releases, we anticipate the maturity of managed memories and heterogeneous memories in accelerator environments to grow. As a result, our future goals will focus on further investigations of the designs into such implementations for MPI communication (point-to-point and collective) operations. Additionally studies pertaining to applications involving irregular memory accesses and different communication patterns is of significant interest.

8. ACKNOWLEDGMENTS

This research is supported in part by National Science Foundation grants #OCI-1148371, #CCF-1213084 and #ACI-1450440. We would like to thank Filippo Spiga from University of Cambridge for providing access to the Wilkes testbed.

9. REFERENCES

- [1] MVAPICH2: MPI over InfiniBand, 10GigE/iWARP and RoCE. <http://mvapich.cse.ohio-state.edu/>.
- [2] NVIDIA Unified Virtual Addressing. <http://developer.download.nvidia.com/CUDA/training/cuda-webinars-GPUDirect-uva.pdf>.
- [3] Open MPI: Open Source High Performance Computing. <http://www.open-mpi.org/>.
- [4] The SHOC Benchmark Suite. <https://github.com/vetter/shoc>.
- [5] Unified Memory in CUDA 6. <http://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/>.

Figure 10: Evaluation of Stencil2D kernels

Figure 11: Evaluation of Stencil 3D application kernel

- [6] M. Bianco and U. Varetto. A Generic Library for Stencil Computations. CoRR:arXiv:1207.1746, 2013.
- [7] J. Cecilia, J. Abellán, J. Fernández, M. Acacio, J. García, and M. Ujaldón. Stencil Computations on Heterogeneous Platforms for the Jacobi method: GPUs versus Cell BE. *The Journal of Supercomputing*, 62(2):787–803, 2012.
- [8] R. Landaverde, T. Zhang, A. Coskun, and M. Herbordt. An investigation of Unified Memory Access performance in CUDA. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, Sept 2014.
- [9] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka. Physis: An Implicitly Parallel Programming Model for Stencil Computations on Large-scale GPU-accelerated Supercomputers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11*, pages 11:1–11:12, New York, NY, USA, 2011. ACM.
- [10] T. Miyoshi, H. Irie, K. Shima, H. Honda, M. Kondo, and T. Yoshinaga. FLAT: A GPU Programming Framework to Provide Embedded MPI. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units, GPGPU-5*, pages 20–29, New York, NY, USA, 2012. ACM.
- [11] 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 *42nd International Conference on Parallel Processing (ICPP)*, pages 80–89, Oct 2013.
- [12] S. Potluri, H. Wang, D. Bureddy, A. Singh, C. Rosales, and D. Panda. Optimizing MPI Communication on Multi-GPU Systems Using CUDA Inter-Process Communication. In *IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW)*, pages 1848–1857, May 2012.
- [13] R. Shi, X. Lu, S. Potluri, K. Hamidouche, J. Zhang, and D. Panda. HAND: A Hybrid Approach to Accelerate Non-contiguous Data Movement Using MPI Datatypes on GPU Clusters. In *43rd International Conference on Parallel Processing (ICPP)*, pages 221–230, Sept 2014.
- [14] R. Shi, S. Potluri, K. Hamidouche, J. Perkins, M. Li, D. Rossetti, and D. Panda. Designing efficient small message transfer mechanism for inter-node MPI communication on InfiniBand GPU clusters. In *21st International Conference on High Performance Computing (HiPC)*, pages 1–10, Dec 2014.
- [15] H. Wang, S. Potluri, M. Luo, A. K. Singh, X. Ouyang, S. Sur, and D. K. Panda. Optimized Non-contiguous MPI Datatype Communication for GPU Clusters: Design, Implementation and Evaluation with MVAPICH2. *CLUSTER '11*, pages 308–316, Washington, DC, USA, 2011. IEEE Computer Society.
- [16] H. Wang, S. Potluri, M. Luo, A. K. Singh, S. Sur, and D. K. Panda. MVAPICH2-GPU: Optimized GPU to GPU Communication for InfiniBand clusters. *Computer Science - Research and Development*, 26(3):257–266, 2011.