Optimizing GPU to GPU Communication on Cray XK7

Jeff M. Larkin
NVIDIA
Santa Clara, CA, USA
jlarkin@nvidia.com

***Abstract*—When developing an application for Cray XK7 systems, optimization of compute kernels is only a small part of maximizing scaling and performance. Programmers must consider the effect of the GPU's distinct address space and the PCIe bus on application scalability. Without such considerations applications rapidly become limited by transfers to and from the GPU and fail to scale to large numbers of nodes. This paper will demonstrate methods for optimizing GPU to GPU communication and present XK7 results for these methods.**

KEYWORDS: MPI,GPU, CUDA, OpenACC, XK7, XK6

I. Introduction

Performance optimization of GPU kernels has been widely discussed in numerous venues and formats. While these optimizations are important, especially when running on individual GPU-enhanced servers, they are only one consideration when programming for large-scale HPC systems, such as the Cray XK7 [1]. On such systems it is important not only to consider the performance of GPU computation and effective management of distinct CPU and GPU memory address spaces on each node, but also the effective communication between nodes using MPI. Without such considerations, overall application performance can quickly become dominated by the communication time, making further GPU optimizations fruitless. This paper will discuss multiple approaches to GPU-to-GPU (G2G) communication via MPI and their relative strengths and weaknesses.

II. G2G Communication Basics

Before discussing GPU communication strategies, it's important to understand what is meant by G2G communication. A Cray XK7 node is comprised of one AMD Interlagos CPU[2] with its associated memory, one Nvidia K20X (Kepler) GPU [3], and a Cray Gemini network ASIC (shared between 2 nodes). Figure 1: GPU to GPU Communication between Cray XK7 Compute Nodes.. illustrates two XK7 nodes. It is assumed for this paper that during the course of an MPI application's runtime, it will be necessary to exchange data that resides in the memory of GPU0 with GPU1 via MPI. This is conceptually represented by the arrow drawn between GPU0 and GPU1. While this arrow is conceptually correct, the actual path the data takes between these two GPU memories and how the programmer expresses this transfer could take several forms.

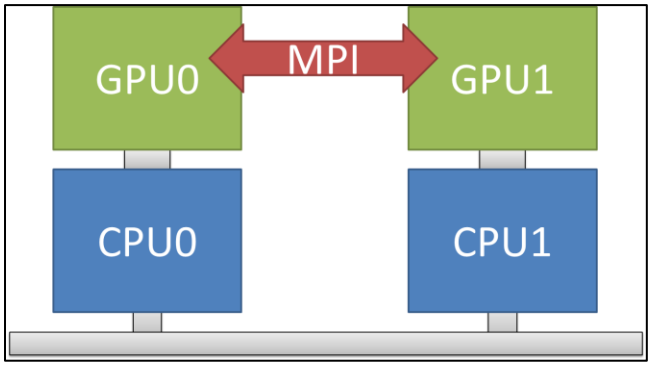

Figure 1: GPU to GPU Communication between Cray XK7 Compute Nodes.

A. CUDA Unaware MPI Transfers

Traditionally, MPI libraries had no concept of distinct CPU and GPU memory spaces and could not deal with accessing GPU memory directly. Since the library was ignorant to GPU memory, it was necessary for the programmer to directly copy GPU data for transfer into a CPU buffer, which was then passed to the MPI library. On the receiving end of this transfer, the programmer would provide the MPI library with a CPU memory buffer and complete the transfer by copying the resulting data to the GPU after it is received. Figure 2 illustrates this approach to G2G communication.

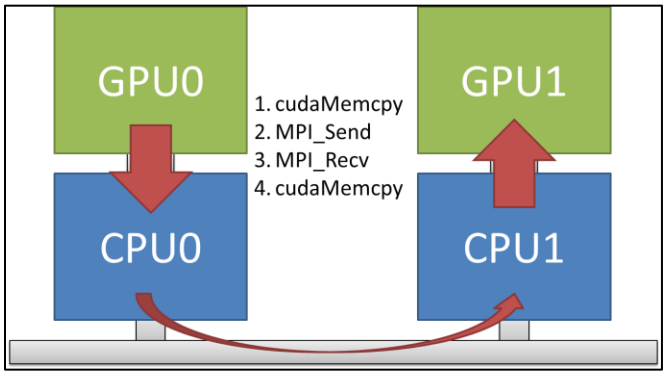

Figure 2: GPU to GPU Communication Through CPU Memory Buffers

While this approach is conceptually straightforward and can be implemented with multiple GPU programming techniques (CUDA, OpenACC, OpenCL, …), it has several shortcomings. From a performance standpoint, it is

impossible to overlap the PCIe transfer from GPU memory of a given buffer with its corresponding MPI transfer over the interconnect. This introduces an inherent serialization of these transfers. From a programming standpoint, while it's possible to use asynchronous PCI transfers and MPI routines, the coordination of such a scheme is difficult and error prone. This method for G2G transfers, while the most limited in terms of performance and flexibility, is the most portable method at time of publication.

B. *CUDA-aware MPI Transfers*

Recent MPI implementations, including Cray's Message Passing Toolkit (MPT) version 5.6.3 and newer, enhance the MPI interface to be able to accept CPU or GPU pointers and act appropriately for each. This ability was enabled by the addition of Unified Virtual Addressing (UVA) in CUDA 4.0. Prior to UVA, CPU and GPU memory both began at address 0x0 and could use the same integer addresses to refer to memory in different physical locations. With UVA, the CPU and all GPUs on a node will share a single virtual memory space, so a given address can only point to a single memory location. This makes it possible to determine where a pointer's memory resides and to act on that information. Using UVA to make an MPI library CUDA-aware means that it's possible to remove the necessity for a programmer to explicitly copy data to and from the GPU around MPI communications, the MPI library can handle this itself.

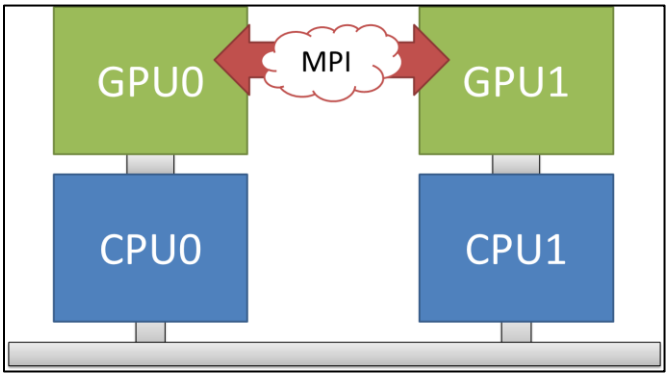

Figure 3: Using CUDA-aware MPI, the programmer can program as if transferring directly between GPU memories.

While one may now represent in the application code the concept of copying data directly between two GPUs via MPI, there is no guarantee that this is actually how the MPI library will transfer the data between the two GPUs. The library may still choose to use a temporary buffer in CPU memory, or it may choose to send the data directly using a remote DMA (RDMA) action, or it may choose to do something else entirely, such as chunking and pipelining large messages. Just as with the details of how CPU-to-CPU MPI messages are sent are generally not vital to the application programmer, it is not critical that the programmer know how the library is transferring a given message in order to begin using CUDA-aware MPI. Ideally, the programmer can trust the MPI implementer to write the library to be as efficient as possible for a given hardware configuration.

It is worth noting that there is not currently a standard way to specify that a given MPI transfer should work on a particular CUDA stream (or related asynchronous construct). Cray's implementation, for instance, uses one CUDA stream for host to device (H2D) transfers and another for device to host (D2H). MVAPICH [4] similarly manages its own pool of CUDA streams for MPI transfers. Because of this, it may be possible at times to achieve better overlapping of GPU computation, PCIe transfers, and MPI by hand-coding these asynchronous activities. The most straightforward way to achieve such overlapping at time of publication is to implement it directly, via CUDA, OpenACC, or the like.

III. Bandwidth and Latency of G2G MPI

For this section, I will use the OSU Micro Benchmarks (OMB) to obtain the MPI latency and bandwidth between two GPU nodes. For each test, I will measure the value for transferring between CPU memories as a baseline, using CPU memory buffers, using pinned (page-locked) CPU memory buffers, enabling G2G communication in the library, and using automatically pipelined G2G communication.

A. *Cray-Mpich2 GPU-Awareness*

GPU-awareness for Cray's MPICH2 library is enabled by setting the `MPICH_RDMA_ENABLED_CUDA` environment variable at runtime. Not setting this environment variable will result in application failure when passing GPU memory to an MPI routine. This feature is available in cray-mpich2 version 5.6.3 and newer. Version 5.6.4 adds an additional environment variable, `MPICH_G2G_PIPELINE`, which, when set, will allow large messages to be chunked and pipelined so that transfers between the CPU and GPU may be overlapped with network transfers. The default number of pipelines, when enabled, is 16, but for OMB the best results are achieved with a value of 64. All tests were run between 2 nodes on separate compute blades.

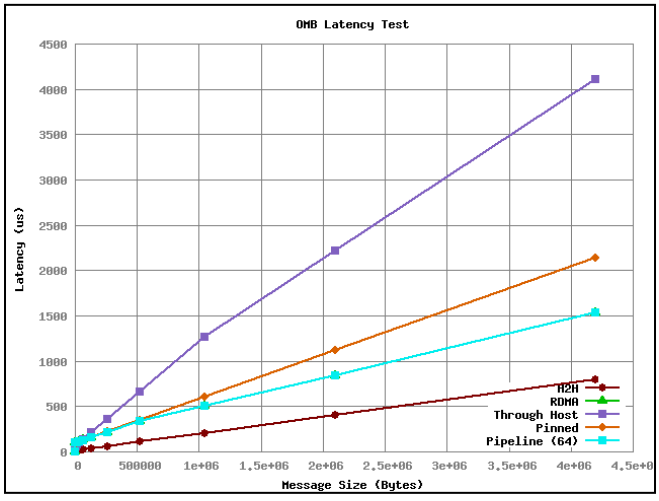

Figure 4: OMB Latency Benchmark (Lower is Better)

Figure 4 shows the measurements for MPI point to point latency for varying message sizes. The lines from highest (worst) to lowest (best) are copying through host memory, copying through pinned memory, RDMA and pipelined (overlapping), and standard host-to-host. These results are as expected. Because standard host-to-host transfers do not require a hop over the PCIe bus, they have the lowest latency, which explicitly buffering through host memory has the highest, due to the latency involved in explicitly copying the data to the host. Both RDMA and pipelined RDMA do not require a copy to a host memory buffer, so they are the second best performing. Lastly, copying through pinned memory falls between RDMA and host-buffered because of the increased PCIe performance achieved from using pinned memory.

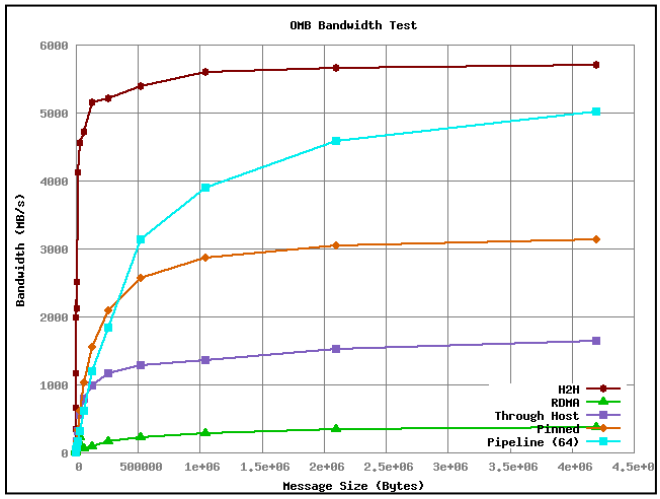

Figure 5: OMB Bandwidth Benchmark (Higher is Better)

Figure 5 shows the MPI point-to-point bandwidth performance for varying message sizes between the same two neighbors as the above latency test. The lines from best to worst are direct host-to-host, pipelined (`MPICH_G2G_PIPELINE=64`), copied through pinned memory, copied through pageable host memory, and direct RDMA. The most striking thing to observe from this figure is the poor performance of direct RDMA transfers on the XK7 platform. This is due to hardware limitations, the details of which are outside of the scope of this paper. By enabling automatic pipelining, however, the library was able to intelligently overcome this limitation and achieve performance nearing that of cpu-to-cpu transfers. The value of 64 was chosen because OMB issues 64 sends or receives before waiting for all of them to complete, so 64 is the optimal number of messages to have in flight simultaneously. Intermediate values were also tried, as illustrated in Figure 6. While it is not surprising that 64 is the ideal value, it was a surprise to see that no other value improved the performance. This result is not intuitive and warrants further exploration.

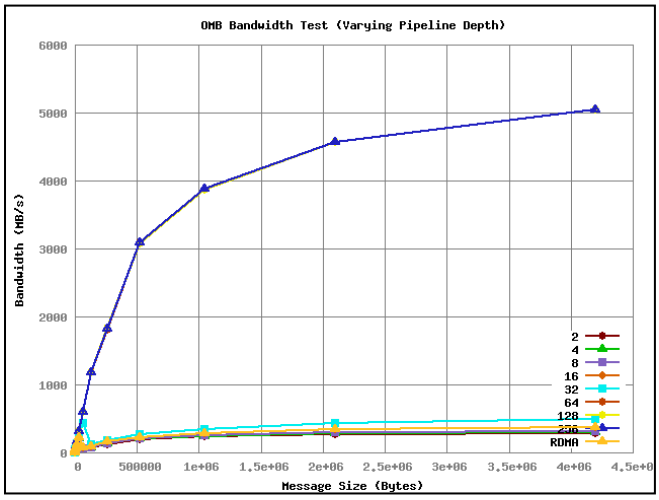

Figure 6: OMB Bandwidth test with varying pipeline depth.

B. Manual Message Pipelining

While the pipelining of individual messages is something that can be done by MPI library developers, application developers may be able to do additional overlapping when sending multiple MPI messages to different neighbors. Taken a step farther, in the case of certain communications, such as a stencil operation, it may be possible use the GPU to pack a message for one neighbor, while copying data to be sent to another, and performing the MPI transfer to yet another. Such a scheme requires a way to expose the dependencies between these operations, such as CUDA streams or OpenACC asynchronous handles, which is not currently available from MPI libraries. To establish a baseline for such a pipeline, the OMB bandwidth benchmark was modified to allow individual messages to be chunked and pipelined. Unlike Cray's automatic chunking, which always breaks the message into 512KB chunks and keeps a configurable number of these chunks in flight, this

implementation divides the message into equal chunks by dividing the total length by pipeline length. While this is not identical to how the MPI library behaves when pipelining is enabled, it is how a user might choose to implement manual pipelining of messages. Best performance was achieved with a pipeline length of 4 and the performance was consistently around 4 GB/s, which falls between staging through pinned memory and using library pipelining. With additional tuning, it may be possible to increase this performance further, but such pipelining is best performed by the MPI library implementer.

IV. Exploiting Communication Concurrency

As discussed above, while automatic pipelining by the MPI library handles individual messages very well, the MPI interface lacks a way to expose the inherent concurrency of some communication patterns. Once such communication pattern widely used in scientific computing, is the stencil operation. When performing a stencil update, some number of "outside" elements from the local data domain must be exchanged with the same number of elements on a neighboring process in order to keep some level of consistency between the two processes. In fact, in certain algorithms, these elements that are exchanged represent an overlap in the two local domains. Since each neighbor exchange on a particular node is independent, except possibly in the case of corner elements, which are shared with multiple neighbors, it is possible to setup a pipeline of exchanges on the node in addition to simply chunking and pipelining a particular message. In [5] the authors discusses how such a technique is applied to the HOMME application and in **Error! Reference source not found.** the authors discuss implementing something similar in the S3D application.

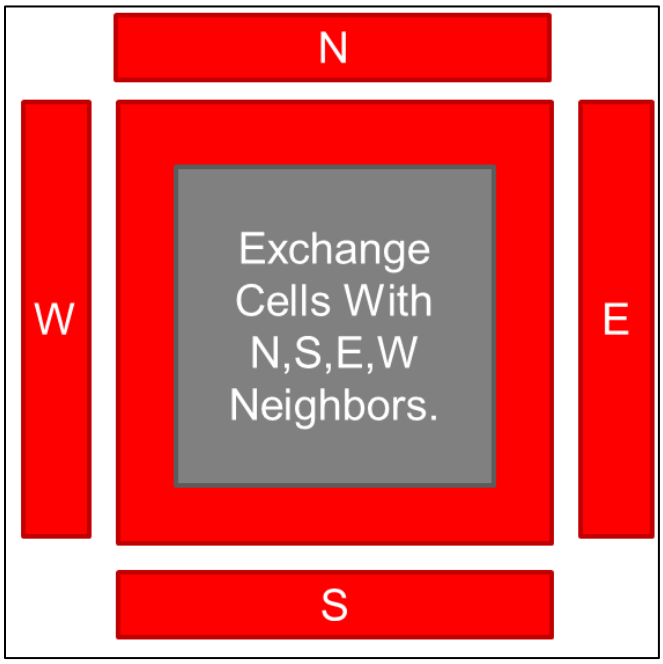

Figure 7: Example of a possible stencil-like boundary exchange with inherent parallelism in each direction.

Figure 7 shows at a very high level how a stencil-like operation may occur within an application, where boundary data must be exchanged to the N, E, S, and W neighbors. When sending the boundary data, each of the four directions is completely independent, so one could build a pipeline where one boundary is packing a contiguous buffer for MPI, another is copying data to or from the GPU memory, and meanwhile MPI messages are being sent and received. With such a pipeline, the cost of PCIe transfers and MPI communication can be completely hidden. Figure 8 shows how such a pipeline may look. Please see the presentation associated with this paper for a pseudocode implementation of such a communication. This example is intentionally very high level and does ignore some complexities of such an algorithm, such as how to deal with overlapping corners and how to efficiently handle compressing non-contiguous boundaries into contiguous memory buffers. It was my intention to build a sample mini-app for such an operation, but I was unable to complete this by the conference deadline.

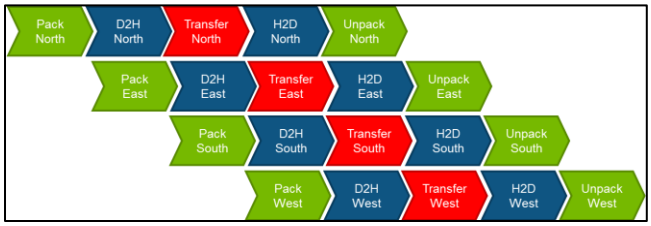

Figure 8: Example pipeline for N/E/S/W boundary exchange.

V. Conclusions and Future Work

In conclusion, careful consideration must be given for communication on an XK7 or other distributed hybrid architecture machines. The addition of the PCIe bus for the

attached accelerator has a negative effect on both latency and bandwidth when communicating between two GPUs. This performance degradation is made worse by hardware limitations on the Cray XK7 platform. MPI library developers have begun to support the passing of GPU pointers to the MPI library, where they can they perform optimizations to maximize single message performance. When an application has additional concurrency around communication that can be exploited to hide the cost of the data transfers, the application developer should expose that parallelism, effectively eliminating the performance cost of PCIe.

Future activities for this research will include additional optimizations to hand-pipelined messages, investigating the non-intuitive results for varying the number of messages in flight from Cray's automatic pipelining, and completing the sample mini-app for demonstrating pipelining of boundary communications.

ACKNOWLEDGMENT

Thank you to Nick Radcliffe and Kitrick Sheets of Cray for their help in understanding and testing Cray MPT. Thank you also to Jiri Krauss of Nvidia for his help in gathering data for this paper.

REFERENCES

[1] http://www.cray.com/Assets/PDF/products/xk/CrayXK7Brochure.pdf

[2] http://www.amd.com/us/products/server/processors/6000-series-platform/6200/Pages/6200-series-processors.aspx

[3] http://www.nvidia.com/content/PDF/kepler/Tesla-KSeries-Overview-LR.pdf

[4] http://mvapich.cse.ohio-state.edu/

[5] Norman, Matthew, et al. Porting the Community Atmosphere Model - Spectral Element Code to Utilize GPU Accelerators. Presented at CUG2012. Cray Users Group; 2012. 2012 April 29 – May 03; Stuttgart Germany.

[6] John M. Levesque, Ramanan Sankaran, and Ray Grout. 2012. Hybridizing S3D into an exascale application using OpenACC: an approach for moving to multi-petaflops and beyond. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (SC '12). IEEE Computer Society Press, Los Alamitos, CA, USA, , Article 15 , 11 pages.

[7] Krauss, Jiri. S3047 - Introduction to CUDA-aware MPI and NVIDIA GPUDirect™. Presented at GTC 2013. GPU Technology Conference; 2013 March 20; San Jose, CA, USA.