Effective Multi-GPU Communication Using Multiple CUDA Streams and Threads

Mohammed Sourouri*†, Tor Gillberg*, Scott B. Baden‡, and Xing Cai*†
*Simula Research Laboratory
P.O. Box 134, 1325 Lysaker, Norway
Email: mohamso@simula.no, torgi@simula.no, xingca@simula.no
†Department of Informatics, University of Oslo
P.O. Box 1080 Blindern, 0316 Oslo, Norway
‡University of California, San Diego
La Jolla, CA 92093 USA
Email: baden@ucsd.edu

***Abstract*—In the context of multiple GPUs that share the same PCIe bus, we propose a new communication scheme that leads to a more effective overlap of communication and computation. Between each pair of GPUs, multiple CUDA streams and OpenMP threads are adopted so that data can simultaneously be sent and received. Moreover, we combine our scheme with GPUDirect to provide GPU-GPU data transfers without CPU involvement. A representative 3D stencil example is used to demonstrate the effectiveness of our scheme. We compare the performance of our new scheme with an MPI-based state-of-the-art scheme. Results show that our approach outperforms the state-of-the-art scheme, being up to $1.85\times$ faster. However, our performance results also indicate that the current underlying PCIe bus architecture needs improvements to handle the future scenario of many GPUs per node.**

***Keywords*—GPU, Multi-GPU, Dual-GPU, CUDA, OpenMP, MPI, GPUDirect, P2P, Asynchronous Communication, Intra-Node communication**

I. Introduction

Heterogeneous systems have lately emerged in the supercomputing landscape. Such systems are made up of compute nodes that contain, in addition to CPUs, non-CPU devices such as graphics processing units (GPUs) or many integrated core (MIC) coprocessors. The Top 500 list [1] from June 2014 clearly shows this trend. The non-CPU devices, often called *accelerators*, have considerably higher floating-point capabilities than the CPUs, and also a greater power efficiency. The downside is however that more effort is needed for using heterogeneous systems, mostly because programming accelerators can be very different from programming CPUs.

An expected feature of future heterogeneous systems is that each compute node will have more than one accelerating device, adding a new level of hardware parallelism. Some of the current heterogeneous supercomputers have already adopted multiple devices per node. The most prominent example is the world's current No. 1 supercomputer, Tianhe-2 (see [1]), which is equipped with three MIC coprocessors on each node. Having multiple devices per node has its advantages with respect to space, energy and thereby the total cost of computing power.

When multiple accelerating devices are used per node in a cluster, data exchanges between the devices are of two types: inter-node and intra-node. MPI [2] is the natural choice for the first type. However, when it comes to intra-node data exchanges, MPI might not be the best solution. Although it is possible to let one MPI process (or more MPI processes) control each accelerator, a few potential inefficiencies arise with this strategy. First, most MPI implementations do not have a hierarchical layout, meaning that intra-node communication is inefficiently treated as inter-node communication. Second, one MPI process per device will increase the overall memory footprint, in comparison with using one MPI process per node. Third, using multiple MPI processes per device requires the creation of additional process contexts, thus additional overhead on top of the enlarged memory footprint.

Due to the inefficiencies connected with MPI in the context of intra-node communication, recent research such as [3] has therefore focused on utilizing a single MPI process per node while adopting multiple OpenMP threads per node. For example, in [3], a single thread is spawned per device. Despite these efforts, the underlying methodologies are essentially the same and imperfect in efficiency. Hence, we believe that a new approach is needed.

In this work, we explore the intra-node communication between multiple GPUs that share the same PCIe bus – on a variety of multi-GPU configurations - including dual-GPU cards where two GPU devices are placed on the same physical card. To improve the state-of-the-art communication performance, we make use of multiple CUDA streams together with multiple OpenMP threads. To fully benefit from the host-avoiding communication technology, we also adopt direct GPU-GPU transfers.

The primary contributions of this paper are as follows:

- We propose an efficient intra-node communication scheme[1], which lets multiple OpenMP threads jointly control each GPU to improve the overlap between computation and communication. Moreover, for each pair of neighboring GPUs, two CUDA streams are used to enable completely simultaneous send and receive of data.

[1]Source code available at `github.com/mohamso/icpads14`

- We quantify the performance advantage of our new intra-node communication scheme by using a representative 3D stencil example, for which inter-GPU data exchanges have a dominating impact on performance.

The rest of this paper is organized as follows: Section II provides the reader with background information on heterogeneous computing, GPUDirect and multi-GPU programming. The current state-of-the-art scheme is presented in Section III. A detailed examination of our new communication scheme is presented in Section IV. We discuss our performance results in Section VI. Section VII surveys related work. Finally, in Section VIII, we conclude with some remarks and describe future work.

II. Background

Apart from aggregating the computation capacity, using multiple GPUs is also motivated from a memory perspective. In comparison with a compute node's system memory that belongs to the CPU host, a GPU's device memory has considerably smaller capacity. For example, 32-64 GB is a typical size for the system memory, whereas a single GPU has between 4 and 12 GB device memory. When dealing with large-scale scientific applications, the size of the device memory may thus become a limiting factor. One way of overcoming this barrier is to make use of multiple GPUs, by either interconnecting multiple single-GPU nodes or installing multiple GPUs within a single compute node. The latter configuration is the focus of this study.

The current generation of GPUs targeted at the HPC market does not share the same memory space with its CPU host. A programmer thus has to explicitly transfer data between the device and host. Although the most recent version of CUDA can abstract the data transfer calls, the physical data motion still takes place. Using multiple GPUs per node adds to the complexity. Independent of the direction, data transfers incur a performance penalty when moving across the high-latency PCIe bus that connects a GPU with its host. Therefore, one of the main objectives of our new communication scheme is to better hide the costs of data transfers.

The latest research activities have focused on improving the efficiency of inter-node MPI communication. Researchers have extended several open-source MPI implementations to support GPU-accelerated systems [4], [5]. Implementations such as OpenMPI, MVAPICH, and MPICH now have specific interfaces for interacting with CUDA devices (Nvidia GPUs). This group of CUDA friendly implementations is commonly collected under the umbrella term *CUDA-Aware MPI*. The main focus of these implementations has so far been to provide a more user friendly MPI interface for directly passing pointers to GPU memory.

A. GPUDirect

The concept of peer-to-peer (P2P) memory access between two GPUs was introduced in GPUDirect v2. P2P allows direct data transfers between two GPUs on the same PCIe bus, completely avoiding copies via the system memory. A precondition for a P2P setup is that the two devices must be located on the same I/O hub or PCIe switch. GPUDirect v2 has two modes: *direct access* and *direct transfers*. In the first mode, a GPU can directly read/store data from/to the device memory of another GPU. In the second mode, data is explicitly moved from the device memory of one GPU to that of another GPU, using CUDA's `cudaMemcpyPeerAsync` commands.

Using P2P transfers in MPI requires either a CUDA-aware MPI implementation that is able to detect multiple GPUs sharing the same PCIe bus or manually through the use of CUDA Inter Process Communication (IPC). The latter is a specific interface that gives a remote process direct access to the memory of a GPU. CUDA-aware MPI implementations usually rely on CUDA IPC in order to transfer data directly between two peers. Consequently, there is an overhead connected to the use of CUDA IPC. This overhead comes on top of the overhead of passing messages. However, these two MPI-related overheads can be avoided if threads are used to control the GPUs within a node. A quantification of the overheads of CUDA IPC is presented in [6].

B. Multi-GPU Programming

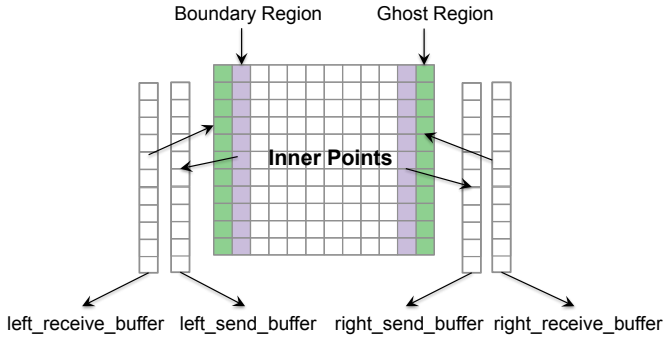


Fig. 1. A simple two-neighbor example. The send buffers correspond to the boundary region (purple), and the receive buffers match the ghost region (green).

As shown i Figure 1, a *subdomain* is the responsible computation area of a GPU. The data values that are needed by the neighbors constitute the so-called *boundary region*, whereas the data values that are to be provided by the neighbors constitute the so-called *ghost region*.

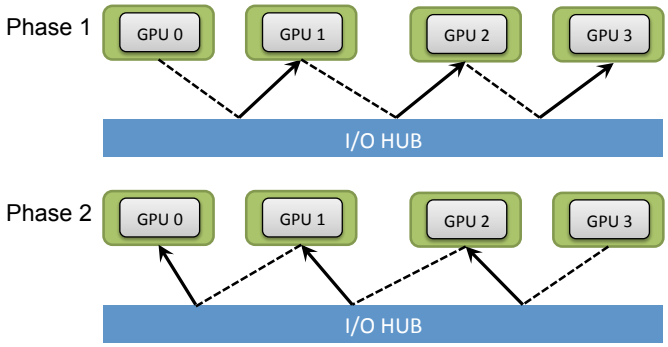
Between each pair of neighboring GPUs, the *data exchange* process consists of first copying data from the "outgoing" buffer of a GPU to the host, and then from the host into the "incoming" buffer of a neighboring GPU. Alternatively, P2P can be used (by `cudaMemcpyPeerAsync`) to directly transfer the content of each outgoing buffer to a matching incoming buffer on a receiving GPU.

Computation is launched first in the boundary region, followed by data exchange of the boundaries. Concurrently with the data exchange, computation of the inner points is performed. If P2P is used in asynchronous implementations, the aforementioned `cudaMemcpyPeerAsync` command should be used.

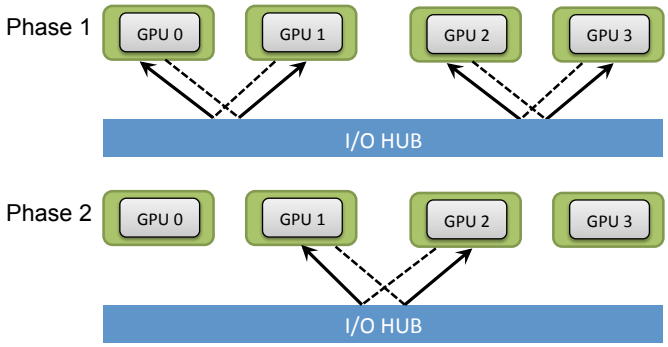
III. State of the Art

This section describes how boundary data exchanges (communication) and computation are handled in the current state-of-the-art intra-node communication scheme, which uses an

asynchronous solution as exemplified in [3], [7]–[12]. The state-of-the-art scheme uses one MPI process or OpenMP thread to control each device, and two CUDA streams per device to overlap communication with computation.



(a) The Left-Right Approach on Four GPUs



(b) The Pairwise Approach on Four GPUs

Fig. 2. Two variations of a multi-GPU communication scheme, both involving two stages as discussed in [13].

There are two variations of data exchanges in the state-of-the-art scheme. They are presented in [13] as the *left-right* and *pairwise* approaches. In both approaches, data exchange is done in two phases. During the first phase of the left-right approach, see Figure 2(a), data is sent to the right neighbor and received from the left neighbor. The direction of data movement is reversed during the second phase. Furthermore, in the pairwise approach, see Figure 2(b), the first phase consists of exchanging border data between some pairs of GPUs. In the second phase, data is exchanged between the remaining neighboring pairs. Another difference between these two approaches is that communication is *uni-directional* in the left-right approach, while it is *bi-directional* in the pairwise approach.

CUDA streams can be used to overlap communication and computation on Nvidia GPUs. Streams are sequence of operations that are executed in the order they are issued. The operations are queued. A queue manager picks an operation from the front of the queue, before feeding it to the device. The overall idea of streams is to increase concurrency by executing multiple operations simultaneously.

The state-of-the-art communication scheme relies on using two CUDA streams per GPU to overlap communication and computation. As Figure 3 reveals, the first stream is dedicated to computing the boundary points and exchanging data, while the second stream is dedicated to computing the inner points.

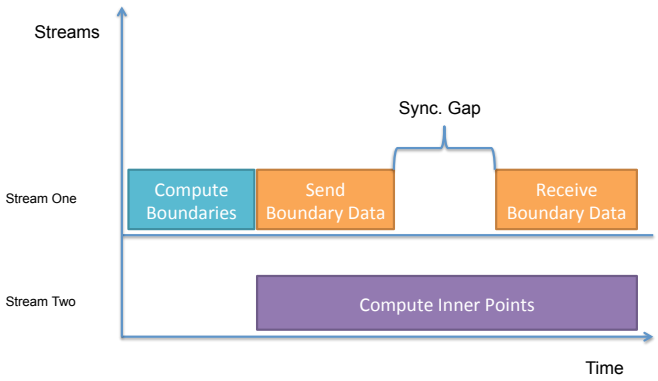


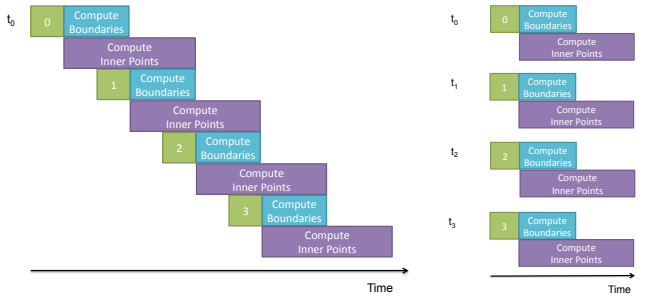
Fig. 3. Two CUDA streams are used in the state-of-the-art scheme to overlap communication and computation. Due to space constraints, only the data blocks for one side of the subdomain is shown for Stream One.

In the Send Boundary Data block of Figure 3, data from the GPU memory is copied to a data buffer on the host, followed by a CUDA stream or device synchronization. Synchronization is necessary to ensure that the data transfer from the GPU to the host is indeed completed, before data can be copied from the host to a buffer of the target device. If MPI is used for intra-node data exchange, an additional layer of process communication and synchronization is added. As a result, the synchronization gap depicted in Figure 3 is widened further.

IV. A New Communication Scheme

We propose a new intra-node communication scheme that targets multiple GPUs sharing the same PCIe bus. In this section, we describe the two fundamental components of our scheme, namely multiple OpenMP threads per GPU and multiple CUDA streams per pair of neighboring GPUs, and how these two components can be combined to outperform the state-of-the-art communication scheme.

A. Multi-Threading



(a) Single-threaded execution sequence

(b) Multi-threaded execution sequence

Fig. 4. Using a single host thread or multiple host threads to launch kernels on multiple GPUs, where t_i denotes the thread number. Each green bar represents the time for switching from one device to another, each purple bar corresponds to computing the boundary region, and each blue bar corresponds to computing the inner points.

Our context of study is intra-node communication between multiple GPUs that share the same PCIe bus. When a thread-based programming model is used, a single or multiple threads

can be used to control the GPUs. Although the single-threaded approach is easier to implement, it has distinct disadvantages, because the application execution is serialized, including the kernel launches. Figure 4(a) depicts the situation if all the actions of the host are executed serially. That is, the kernels on GPU 0 is launched first, then on GPU 1, and so on. As the figure shows, the overhead of using a single thread is considerable. For this reason, we choose not to use a single thread to control multiple GPUs in our scheme. Instead, we choose to use multiple OpenMP [14] threads.

Similar to the state-of-the-art scheme, we let one OpenMP thread control each GPU as the *main thread*. The benefit of the one-thread-per-GPU approach is that the different kernels can be launched in parallel, and thus, eliminating kernel launch overheads. Figure 4(b) illustrates this in greater detail.

However, unlike the state-of-the-art scheme, we take one step further and make use of multiple threads per GPU for improving the communication performance. The neighboring sides of a subdomain are independent of each other. If decoupled, each part can be processed in parallel. The decoupling process consists of splitting each neighboring side of the subdomain into two parts: a send and a receive part. After the split, each neighbor contains a send and a receive part. Next, we spawn one *assistant thread* per neighbor, meaning that the total number of OpenMP threads per GPU equals: $3\times$ the number of GPUs.

B. Multi-Streaming

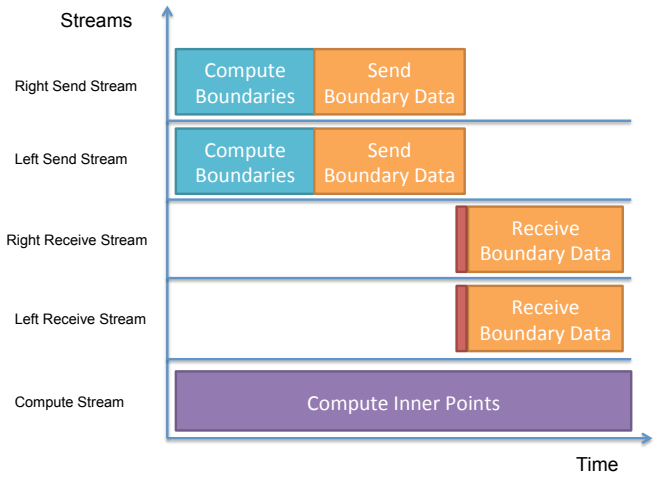


Fig. 5. Five streams used in our scheme to achieve better overlap of communication and computation.

The previous section demonstrated how a group of threads can increase scheme concurrency, and realize bi-directional communication. In this section, we demonstrate how multiple threads in tandem with multiple streams can be used to reduce additional overheads.

In the state-of-the-art scheme, two CUDA streams are created per GPU to overlap communication with computation. To recap, the first stream is used for computing the boundaries and performing communication, while the second stream is responsible for computation of the inner points. Despite being two independent operations, the second stream is launched after the computation of the boundaries has finished in the first stream [8]. Even if the boundary kernels were launched simultaneously as the inner points kernel, the use of a single thread/process per device would result in a delay between the start of the inner points kernel.

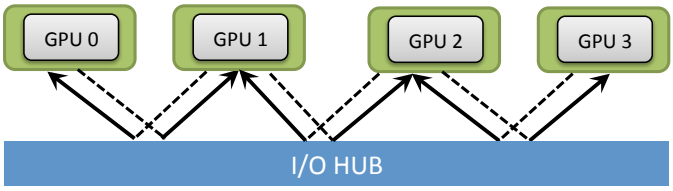


Fig. 6. Using separate CUDA streams can allow simultaneously sending and receive data between multiple GPUs.

We repeat the subdomain splitting process mentioned in Section IV-A. In addition we create a group of streams per thread. These streams are created for the send, receive and compute phases, as shown in Figure 5. In other words, the number of streams per GPU is $2\times$ number of neighbors of the GPU.

The different assistant OpenMP threads are associated with each group of streams. The use of multiple streams has two major advantages. One, it enables us to decouple the data transfers going in different directions within each subdomain, and as a result, the two phases depicted in Figure 2 can now be merged into a single phase. Figure 6 illustrates our single phase scenario. Two, by creating a group of streams and attaching the assistant threads to each stream group, we are able to mitigate the delays and overheads that arise in the state-of-the-art scheme.

Another motivation for letting each assistant thread control one group of streams is that, in the scenario where only a single thread is used, i.e. the state-of-the-art scheme, synchronization will stall the master thread. By using additional threads, we avoid stalling the main execution thread. This is especially of importance in real-world applications where the running thread needs to attend to other tasks as well.

Multiple streams also express the independence that exists between tasks more clearly. We found this property to be especially useful on the Fermi GPUs, where placing multiple streams inside a loop can lead to false dependencies. Further information on this topic can be found in the documentation about HyperQ [15].

C. Summary

Our scheme is built upon two principles: multi-threading and multi-streaming. These two techniques are combined to create a more efficient intra-node communication scheme that increases the overall concurrency.

Multiple threads are used to reduce kernel launch overhead, avoid stalling the running host thread and improve application performance by reducing the gap between computation and communication. We also believe that multiple threads can also be useful for architectures that do not support streams.

Multiple streams are used to stack communication so that data exchange can occur simultaneously on both sides of a

subdomain. The use of multiple streams gives us a more fine-grained control of the different operations, enabling us to launch all groups of streams at the same time, resulting in bi-directional data transfers in a single phase. Moreover, the combination of multiple threads and multiple streams reduces the synchronization overhead that is needed between the send and receive streams. Additionally, P2P can be adopted for the purpose of reducing the overhead directly related to data transfer.

V. Experiments and Measurements

A. Experimental Platforms

GPU Name	Tesla K20	Tesla C2050	GeForce GTX 590
Architecture	Kepler	Fermi	Fermi
# SMs	13	14	2 x 16
# cores/SM	192	32	32
register file/SM	65K	32K	32K
storage/SM	64KB	64KB	64KB
memory size	5GB	3GB	2 x 1.5GB
bandwidth (GB/s)	208	144	2 x 163.8
SP, GFLOPs	3520	1030	2 x 1243
DP, GFLOPs	1170	515	2 x 155

TABLE I. An overview of the GPUs used. SM denotes streaming multi-processor, whereas storage/SM means shared memory plus L1 cache per SM.

All tests were conducted on three different systems. The first machine contains a single Intel Xeon E5620 CPU, equipped with two Nvidia GeForce GTX 590 GPUs. These are dual-device cards, i.e., two GPU devices are fused together on the same physical print-circuit board. Therefore, the total number of GPU devices is four. The second machine is a dual-socket server containing two Intel Xeon E5-2650 CPUs with four Nvidia Tesla K20 GPUs. The third and last machine is a special multi-GPU node from NERSC's GPU testbed, Dirac. This node is equipped with two Intel Xeon 5520 CPUs with four Nvidia Tesla C2050 GPUs. A more detailed technical overview of the different GPUs is shown in Table I. All calculations used double precision floating point with CUDA version 5.5. Due to the lack of error-correction code (ECC) support on the GTX 590, ECC was turned off on all GPUs. Because of technical difficulties, we could not get MVAPICH2 v1.9 to work on one of our test systems, hence, we have used OpenMPI 1.6.5.

B. Bandwidth Measurements

Communication is typically identified as one of the main performance bottlenecks in multi-GPU applications. Hence, the speed of the communication plays a vital role. We have measured the bandwidth by taking the time it takes to transfer data under the following circumstances:

- From one device to another device via the host (DtD)
- From one device to another device using P2P

Due to the nature of the GTX 590, we are also able to transfer data directly between the two on-board devices. A special PCIe bridge on the GTX 590 enables direct data transfers between the two devices so that communication does not have to go through the main PCIe bus. Figure 7 shows how the bandwidth scales with data sizes up to 32 MB. Notice that moving data directly between two devices (P2P) is almost twice as fast as moving the same data through the host. The performance is especially evident on the dual-GPU GTX 590. If data motion occurs via the host on GTX 590, communicating between on-board devices is equally fast as using the two physically separated devices. Thus, the results for transferring data from device 0 to device 2 (DtD [0-2]) have been omitted from Figure 7.

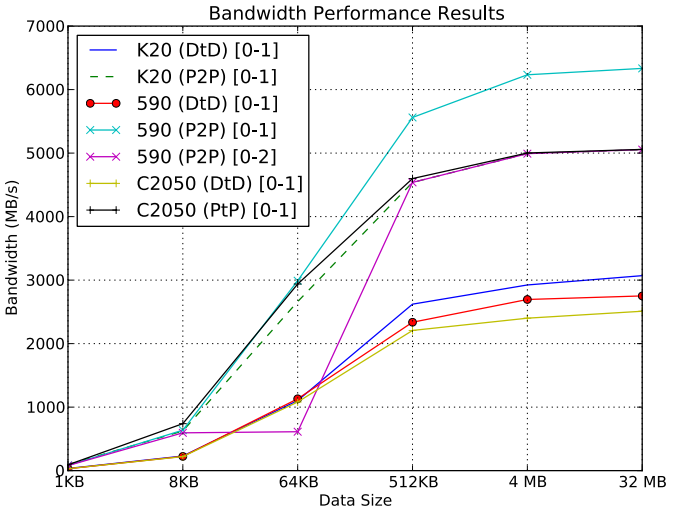


Fig. 7. Measured bandwidths for different configurations. The numbers in the bracket refer to the devices used, e.g., [0-1] means a transfer from device 0 to device 1.

The four Tesla C2050 cards are connected pairwise to two CPUs. Devices 0 and 1 constitute the first pair, while devices 2 and 3 constitute the second pair. In contrast to the four GTX 590 devices, where P2P is available across all devices, P2P is only available within each pair, because an I/O hub separates the two pairs. In practice, this means that data motion across the two pairs has to travel through an additional I/O hub. There is thus a small performance penalty associated with this configuration.

The four-way Tesla K20 server resembles the configuration of the Tesla C2050 system, with one important distinction. That is, there is only a single I/O hub. P2P is available between devices 0 and 1, and between devices 2 and 3. In order to communicate between e.g. device 1 and 2, data transfers must be staged through the host.

P2P transfers between two directly connected P2P devices on the Tesla C2050 and the K20 system are equally fast. Transferring data via the host is slightly faster on the K20 platform, possibly due to faster QPI link speed. Our measurements also reveal that the host-to-device bandwidth for device 2 and 3 is approximately 1.25x slower than device 0 and 1 on the Tesla C2050 system. Moreover, the device-to-host bandwidth for device 2 and 3 is about 1.4x slower compared to device 0 and 1 on the same system. We did not observe the same bandwidth difference on the K20 system.

C. Benchmark Stencil Computation

Stencil computations constitute one fundamental tool of scientific computing. They are typically used to discretely solve a differential equation using finite difference methods,

which in turn give rise to a stencil calculation. For this paper, we have chosen the following 7-point stencil that sweeps over a uniform 3D grid:

$$U^{n+1}_{i,j,k} = \alpha U^{n}_{i,j,k} + \beta \left(U^{n}_{i+1,j,k} + U^{n}_{i,j+1,k} + U^{n}_{i,j,k+1} + U^{n}_{i-1,j,k} + U^{n}_{i,j-1,k} + U^{n}_{i,j,k-1} \right)$$

where α and β are two scalar constants. This 3D stencil computation can arise from discretizing the Laplace equation (using $\Delta x = \Delta y = \Delta z$) over a box and solving the resulting system of linear equations by Jacobi iterations. In the literature, this 3D stencil is thus widely referred to as the 3D Laplace stencil. (The same stencil can also arise from solving a 3D heat equation by a forward-Euler time stepping method.)

We chose, for simplicity reasons, to decompose our problem domain along the z direction, resulting in a 1D decomposition. One benefit of using a 1D decomposition is that kernels developed for a single-GPU can be used without any additional modification. Nevertheless, we acknowledge that using a 1D domain could be considered as suboptimal due to the communication overhead that arises when working on extremely large problem sizes or across many nodes. However, we believe it is sufficient for our study, as we deal with neither extremely large problem sizes nor many nodes. Previous studies such as [16] have shown that 1D decomposition with similar problem size, provides linear scaling up to 16 nodes.

Each result is divided into four scenarios, determined by the type of implementation used: Baseline, MPI, OpenMP, OpenMP w/P2P. The baseline version constitutes a naïve implementation where a single host thread is used to control all GPU devices, and where all data transfers go via the host.

Our MPI implementation of state-of-the-art scheme utilizes the Left-Right approach described in Figure 2(a). This turned out to be marginally faster than Pairwise approach.

The GPUDirect v2 mode of the direct access has been omitted for this paper, mainly due to two reasons. One, the performance associated with direct access is considerably lower than that associated with direct transfers. Two, only one of the three experimental platforms supports direct access.

The size of the 3D grid ranges from 64^3 to 512^3 inner points. Due to memory size limitations it was not possible to run experiments for the largest problem size on a single GTX 590 device.

VI. Results

A. Experiments on Kepler

As Figure 8(a) shows, our scheme is able to outperform the state-of-the-art scheme in every case by a considerable margin. Interestingly, our scheme is faster for smaller problem sizes when using two Kepler GPUs, and for the larger problem sizes when using four Kepler GPUs. However, if the computation part is big enough, communication can be entirely hidden. This explains the good performance of the state-of-the-art scheme for the largest problem sizes.

Using asynchronous P2P transfers is faster than the pure OpenMP implementation. This is possible because the CUDA subsystem overlaps the staged P2P transfers. The performance advantage for the P2P implementations comes from the direct P2P transfers that occur on the edges of the domain, that is, between device 0 and 1 and device 2 and 3.

B. Experiments on Fermi

The performance results on the Fermi systems (see Figure 8(b) and Figure 8(c)) are similar to the results observed on the Kepler platform.

For the Tesla C2050 system, direct P2P transfers are staged but overlapped between device 1 and 2. However, as the performance results indicate, thanks to faster direct P2P transfers on the edges of the domain, the P2P solutions are able to outperform the other implementations.

We are also able to outperform the state-of-the-art scheme on the Fermi GPUs. The performance gaps are especially large when four GPUs are involved. For example, our scheme is 58 percent faster for the largest problem size on the Tesla C2050 and 40 percent faster for the largest problem size on the GTX 590.

On the GTX 590 system, we can observe a larger performance gap between the P2P and the other implementations, because true P2P transfers are available between all devices on this system.

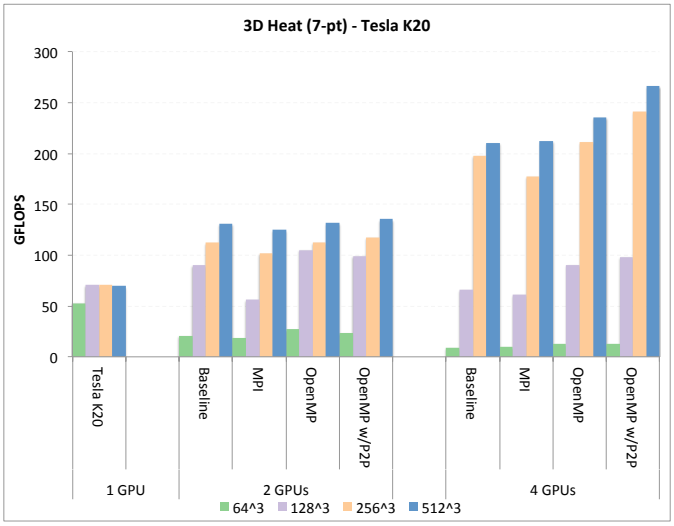
Figure 8(d) shows a direct comparison of our scheme versus the state-of-the-art scheme implemented in MPI on the Tesla C2050 system. The same benchmark application is used. We observe that our scheme is able to outperform the state-of-the-art scheme quite considerably when two GPUs are used. On the other hand, when four GPUs are used, the difference between the two different schemes is not visible until we reach the larger problem sizes. For larger problem sizes, our scheme is better at hiding communication than the state-of-the-art scheme.

VII. Related Work

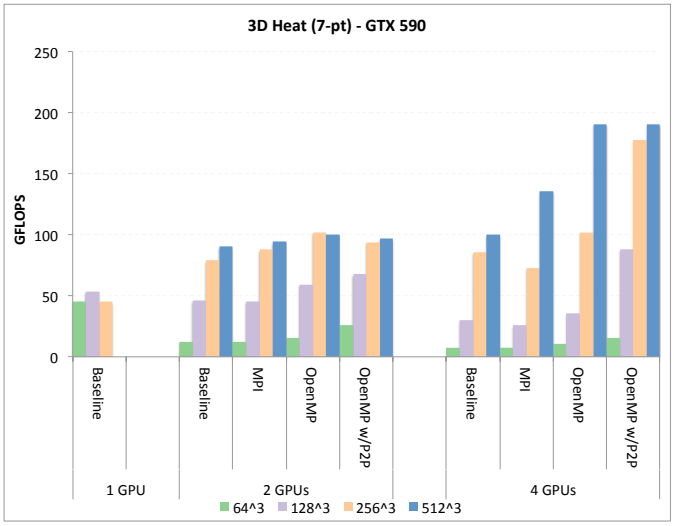
Paulius Micikevičius [12] has investigated how a fourth-order wave equation can be solved on a single compute node with up to four GPUs using MPI. Micikevičius reports linear scaling in the number of GPUs used for all but one case. The domain is decomposed along the z-axis and computation is overlapped with communication.

We decompose the domain in the same manner and interleave computation with communication. However, we rely on the use of threads and not MPI processes to control multiple GPUs. We also make use of GPUDirect to avoid potential host intervention, as well as a new communication scheme to increase the overlap. Depending on the setup and the data communication method, we also observe the same linear and superlinear speedup that Micikevičius reports. The superlinear speedup seen in Figure 8(d) can be explained by reduced TLB miss rate.

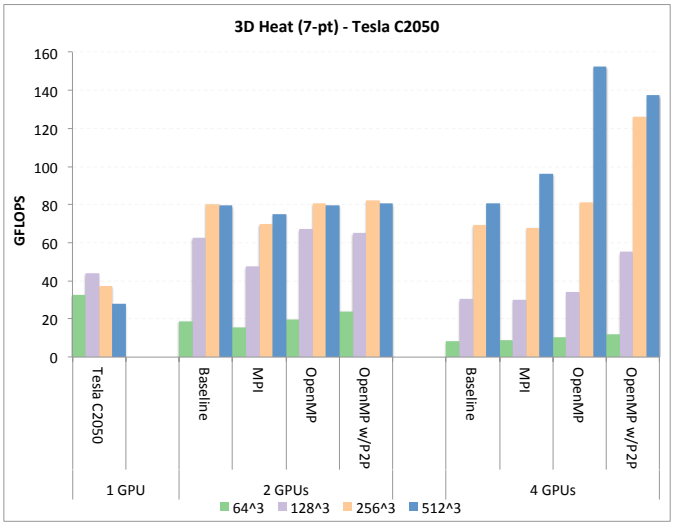
Thibault et al. [17] develop a multi-GPU Navier-Stokes solver in CUDA that targets incompressible fluid flow in 3D. In the study, one Pthread is spawned per GPU. The code runs on a single Tesla S870 GPU server with four GPUs. Depending on the number of GPUs and problem size, Thibault et al. report speedup between $21 - 100\times$ compared to a single CPU core. The implementation presented in Thibault et al. does not



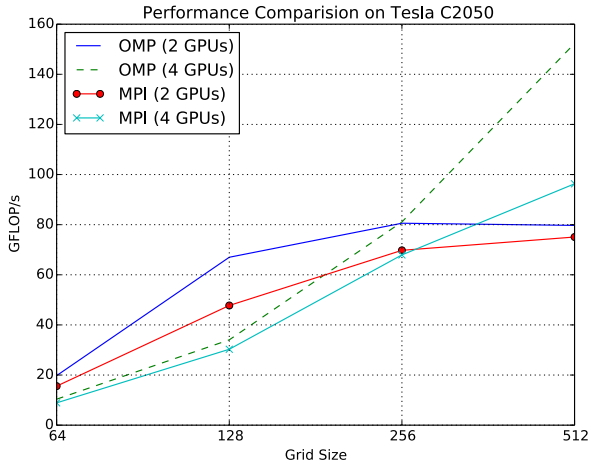
(a) Performance results on Tesla K20



(b) Performance results on GTX 590



(c) Performance results on Tesla C2050



(d) Speedup comparison between state-of-the-art and our scheme

Fig. 8. The sustained performance results measured in GFLOPs on the three experimental platforms.

overlap computation with communication and CUDA streams are not used. Results reported are also in single precision.

Thibault and Senocak's work is extended by Jacobsen et al. [11]. Major changes include the use of CUDA streams to interleave computation with communication, and the use of MPI processes in preference of Pthreads. One MPI process was created per device. All experiments were conducted on a cluster containing 128 GPUs. Each node was equipped with two Tesla C1060 GPUs, putting the number of compute nodes at 64. The implementation using all 128 GPUs is 130× faster than a multicore implementation running on only two CPU cores on a dual-socket Intel Xeon E5530 quad-core processor operating at 2.4 GHz.

In Bernaschi et al. [7], a CUDA-Aware MPI implementation is used to study the inter-node communication performance by measuring the time it takes to update a single spin of the 3D Heisenberg spin glass model. The scheme used in Bernaschi et al. uses two streams, one for compute and one for data exchange. P2P is used between two nodes (each equipped with two different Fermi GPUs). Inter-node P2P is possible thanks to the use of APEnet+, a custom 3D Torus interconnect that can access the GPU memory without going through the host.

Our proposed scheme uses up to five streams, four streams for data exchange (two per each side of the subdomain) and one for the inner points. In our scheme, the computation of the inner points and the boundaries are scheduled to run at the same time. Moreover, since our scheme uses threads, we are able to easily pass pointers between GPUs with minimal overhead, whereas exchanging GPU pointers using MPI processes involves explicit message passing.

VIII. Conclusions

In this work we have studied the current state-of-the-art intra-node communication scheme. Based on our findings, we have proposed a new scheme that is faster than the existing

state-of-the-art scheme. The main ingredient of our scheme is to combine multiple OpenMP threads with multiple CUDA streams for a more efficient overlap of communication with computation.

First, we make use of multiple OpenMP threads per device to increase the scheme concurrency. This is in stark contrast to the state-of-the-art where each device is controlled by a single thread or process. Then, we create a group of CUDA streams for each stage of the communication and computation, whereas the state-the-art scheme uses only two CUDA streams. Finally, we combine the two techniques together to create a more efficient intra-node communication scheme that is able to perform bi-directional communication with lower synchronization overhead.

Depending on the test platform, results indicate that our scheme is able to outperform the state-of-the-art scheme with quite a noticeable margin. The best speedup on the GTX 590 platform was $1.4\times$, $1.6\times$ on the C2050 platform and $1.85\times$ on the K20 platform.

Our investigations also show that there are scenarios where P2P can further reduce communication overhead to some degree. However, the importance P2P is limited. Our results demonstrate that P2P is also impeded by todays PCIe bus.

We have three immediate extensions planned for the future. The first extension involves studying the effect of larger ghost regions. Currently, our implementations use the thinnest possible ghost region width. A thicker ghost region can potentially benefit the computation of wider stencils such as a 19-point stencil or in combination with time unrolling where two sweeps are performed per time step. The use of time unrolling increases the computation while at the same time reducing the number of boundary data exchanges. Thus, time unrolling can be regarded as an important feature with respect to computation of smaller data sets where the majority of the time is spent on communication. In this study we have looked at a traditional stencil code, however, our scheme is by no means limited to stencil code. We are in the process of exploring the use of our strategies in connection with applications from the real world. Finally, work is also underway to extend our scheme to other architectures such as Intel's Xeon Phi.

IX. Acknowledgments

This work was supported by the FriNatek program of the Research Council of Norway, through grant No. 214113/F20. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant No. OCI-1053575. Computer time on the Dirac system was provided by the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] "Top500 supercomputers sites," http://www.top500.org/lists/2014/06/.

[2] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard, Version 3.0*. High Performance Computing Center Stuttgart (HLRS), September 2012.

[3] T. Shimokawabe, T. Aoki, and N. Onodera, "A high-productivity framework for multi-GPU computation of mesh-based applications," *HiStencils 2014*, p. 23, 2014.

[4] 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*, vol. 26, no. 3-4, pp. 257–266, Apr. 2011.

[5] F. Ji, A. M. Aji, J. Dinan, D. Buntinas, P. Balaji, W.-C. Feng, and X. Ma, "Efficient intranode communication in GPU-accelerated systems," *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pp. 1838–1847, May 2012.

[6] S. Potluri, H. Wang, D. Bureddy, A. K. Singh, C. Rosales, and D. K. Panda, "Optimizing MPI communication on multi-GPU systems using CUDA inter-process communication," in *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, May 2012, pp. 1848–1857.

[7] M. Bernaschi, M. Bisson, and D. Rossetti, "Benchmarking of communication techniques for GPUs," *Journal of Parallel and Distributed Computing*, vol. 73, no. 2, pp. 250–255, 2013.

[8] M. Rietmann, P. Messmer, T. Nissen-Meyer, D. Peter, P. Basini, D. Komatitsch, O. Schenk, J. Tromp, L. Boschi, and D. Giardini, "Forward and adjoint simulations of seismic wave propagation on emerging large-scale GPU architectures," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 38:1–38:11.

[9] D. Playne and K. Hawick, "Asynchronous communication for finite-difference simulations on GPU clusters using CUDA and MPI," in *Proc. International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA11), number PDP2793, Las Vegas, USA*, 2011.

[10] E. H. Phillips and M. Fatica, "Implementing the Himeno benchmark with CUDA on GPU clusters," in *Proceedings of 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, 2010, pp. 1–10.

[11] D. A. Jacobsen, J. C. Thibault, and I. Senocak, "An MPI-CUDA implementation for massively parallel incompressible flow computations on multi-GPU clusters," in *Proceedings of the 48th AIAA Aerospace Science Meeting*, 2010, pp. AIAA 2010–522.

[12] P. Micikevicius, "3D finite difference computation on GPUs using CUDA," *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units - GPGPU-2*, pp. 79–84, 2009.

[13] ——, "Multi-GPU programming," http://www.nvidia.com/docs/IO/116711/sc11-multi-gpu.pdf.

[14] OpenMP Architecture Review Board, *OpenMP Application Program Interface Version 4.0*, July 2013.

[15] T. Bradley, "Hyper-Q example," http://docs.nvidia.com/cuda/cuda-samples/index.html#simplehyperq, Aug. 2012.

[16] 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 *roceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. IEEE, 2011, pp. 11:1–11:12.

[17] J. C. Thibault and I. Senocak, "CUDA implementation of a Navier-Stokes solver on multi-GPU desktop platforms for incompressible flows," in *Proceedings of the 47th AIAA Aerospace Sciences Meeting*, 2009, pp. AIAA 2009–758.