2013 IEEE 7th International Symposium on Embedded Multicore/Manycore System-on-Chip

Auto-Tuning of Data Communication on Heterogeneous Systems

Marc Jordà[1], Ivan Tanasic[1], Javier Cabezas[1], Lluís Vilanova[1], Isaac Gelado[1], and Nacho Navarro[1,2]

[1]Barcelona Supercomputing Center
[2]Universitat Politècnica de Catalunya
{first.last}@bsc.es

***Abstract*—Heterogeneous systems formed by traditional CPUs and compute accelerators, such as GPUs, are becoming widely used to build modern supercomputers. However, many different system topologies (i.e., how CPUs, accelerators, and I/O devices are interconnected) are being deployed. Each system organization presents different trade-offs when transferring data between CPUs, accelerators, and nodes within a cluster, requiring different software implementations to achieve optimal data communication bandwidth.**

In this paper we explore the potential impact of two optimizations to achieve optimal data transfer bandwidth: topology-aware process placement policies, and double-buffering. We design a set of experiments to evaluate all possible alternatives, and run each of them on different hardware configurations. We show that optimal data transfer mechanisms depend on both the hardware topology and the application dataset size. Our experimental evaluation shows that auto-tuning applications to match the hardware topology, and to find the best double-buffering configuration can improve the data transfers bandwidth up to 70% for local communication and is key to achieve optimal bandwidth in remote communication for data transfers larger than 128KB.

I. Introduction

Heterogeneous computing systems that couple general purpose CPUs and massively parallel accelerators are becoming common place in High Performance Computing (HPC) environments. Besides the performance benefits [1], heterogeneous architectures also provides higher energy efficiency [2] than traditional homogeneous systems. Most programming models for heterogeneous systems, such as NVIDIA CUDA [3] and OpenCL [4], assume that the code running in the CPU (i.e., host) and the accelerator (i.e., device) has access to separate virtual address spaces. Therefore, any data communication between the CPU and accelerators, or between accelerators, requires explicit data transfer calls to copy data across address spaces. The optimal implementation of these data transfer calls heavily depends on the underlaying hardware organization. Hence, the behaviour of applications have to be adapted to the hardware topology where they are being executed.

The simplest design of an heterogeneous node includes one multi-core CPU connected to one accelerator through a PCIe bus. Each of these chips is connected to separate physical memories, and Direct Memory Access (DMA) transfers are used to move the data between the CPU and the accelerator. A similar organization attaches several accelerators to the same PCIe bus, making it possible to transfer the data across accelerators using DMA transfers without intervention of the CPU. Alternatively, systems with several CPUs connected through a cache-coherent interconnect (e.g., HyperTransport or QPI) allow attaching accelerators to each CPU through separate PCIe buses. In this scenario, data communication across accelerators requires the intervention of the CPU, since DMA transfers cannot interface with the cache-coherent interconnect. In these systems, the data transfer bandwidth between CPUs and accelerators varies depending on whether the data requires going through the interconnect or not. Besides the effects due to the hardware topology, the performance of data transfers also depends on the software implementation. A widespread technique to hide the cost of data transfers is double-buffering, which requires extra synchronization points that might actually harm the overall performance. Similarly, the performance of data transfers might be boosted by using pre-pinned memory in user applications, since it removes the need for intermediate copies from user buffers to DMA buffers. Pre-pinned memory, however, is a scarce resource and cannot be used for very large data sets.

The large number of design choices makes these codes ideal candidates for auto-tuning. In this paper we study the amenability of auto-tuning techniques to implement data transfers in heterogeneous systems by analyzing the performance characteristics of different data transfers techniques on several heterogeneous architectures. Our experimental results show that auto-tuning can achieve up to 70% speed-ups on data transfers. The first contribution of this paper is a description of different multi-GPU systems and the techniques needed to achieve the highest data communication throughput. The second contribution of this paper is a comprehensive set of benchmarks to measure the performance of the most common data communication patterns in HPC codes.

II. Background and motivation

In this Section we introduce different kinds of GPU-based system organizations and how current programming models expose their different characteristics.

978-0-7695-5086-2/13 $26.00 © 2013 IEEE
DOI 10.1109/MCSoC.2013.40

A. GPU-based systems

First programmable GPUs (e.g., NVIDIA G80) appeared in the form of discrete devices attached to the system through an I/O bus like PCIe. A discrete GPU has its own high-bandwidth RAM memory (e.g.,GDDR5). Data must be copied from host (i.e., CPU) to the GPU memory before any code on the GPU accesses it and results must be copied back to be accessed by the CPU code. While last generation I/O buses deliver up to 16 GBps (PCIe 3.0 [5]), CPU↔GPU transfers can easily become a bottleneck. Later, designs that integrate CPU and GPU in the same die have been proposed (e.g., NVIDIA ION, AMD Fusion, Intel Sandy/ Ivy Bridge). In these designs, CPUs and GPUs share the same physical memory, thus eliminating the need for data transfers. However, the memory bandwidth delivered by the host memory is an order of magnitude lower than memories used in discrete GPUs (30 GBps vs 200 GBps). Therefore, in this paper we focus on systems with discrete GPUs.

HPC systems commonly include several GPUs in each node to further accelerate computations and handle large datasets. GPUs can be directly connected to a PCIe root complex or to a PCIe switch. Typically, each CPU socket in the system has its own I/O controller that contains a PCIe root complex.

Direct communication between GPUs connected to different PCIe root complexes (e.g., in a multi-socket system) is not currently supported (due to limitations in the CPU interconnect) and intermediate copies to host memory are needed. Figure 1a shows a single-socket system in which all GPUs are connected to the same PCIe bus. In this system, all memory transfers go through the PCIe root complex found in the I/O controller, that may become a bottleneck. Figure 1b shows a more complex topology that uses two PCIe switches. While CPU↔GPU transfers still suffer from the same contention problems, GPU↔GPU transfers can greatly benefit, since transfers between different pairs of GPUs can proceed in parallel. Figure 1c shows a dual-socket system with one I/O controller per socket. In this configuration, two CPU↔GPU memory transfers can execute in parallel, one in each I/O hub. However, the NUMA (Non-Uniform-Memory-Access) nature of the system makes the management of memory allocations used in data transfers more difficult [6].

There are several techniques to communicate GPUs with I/O devices (e.g., disks, network cards). The most simple implementation is using an intermediate copy to host memory. This guarantees compatibility with all device drivers at the cost of extra memory transfers. NVIDIA introduced the GPUDirect [7] technology to reuse host memory buffers across different device drivers to avoid unnecessary copies, that was mainly adopted by Infiniband network interface vendors. The second version of the technology allows GPUs to directly communicate with devices connected to the same PCIe root complex thus completely eliminating intermediate copies to host memory.

B. Programming multi-GPU systems

Programming models for GPUs provide a host programming interface to manage all the memories in the system. CUDA [3] allows programmers to allocate memory in any GPU in the system and provides functions to copy data between them. Since copies between host and GPU memories are performed using DMA transfers, pinned host memory buffers must be used. This ensures that pages are not swapped out to disk by the Operating System during the transfer. Memory transfers that use regular user (non-pinned) buffers are internally copied to pinned memory buffers by the runtime.

Applications that run on multi-node systems usually use the MPI (Message Passing Interface[8]) programming model to communicate across nodes. MPI functions take host pointers as input/output buffers and, therefore, transferring data between GPUs in different nodes requires using intermediate copies to host memory. However, latest versions of some MPI implementations allow the use of GPU memory pointers. This enables direct GPU↔GPU data transfers between MPI processes running on the same node, and GPU to NIC transfers when data is transferred through the network.

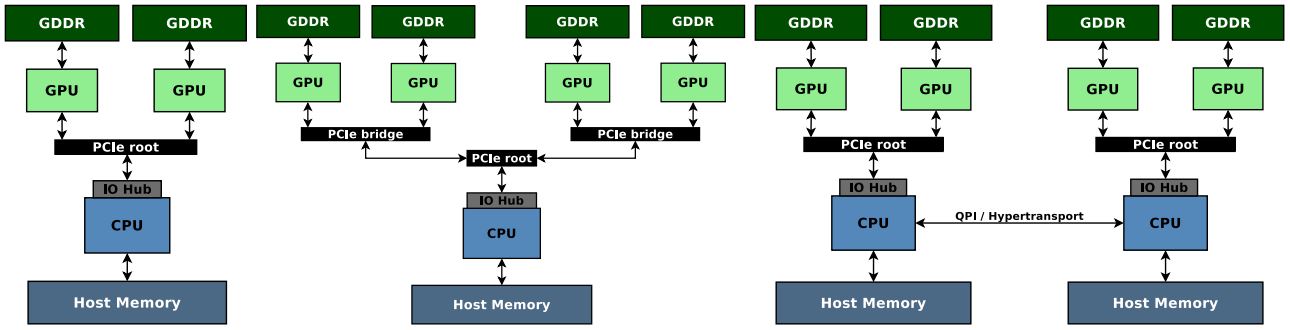
III. Experimental methodology

A. Description of the Experiments

We design experiments to measure the impact of locality and double-buffering on the performance of data communication across CPUs, GPUs, and I/O devices. We cover three possible scenarios: communication inside a node (intra-node), communication between nodes (inter-node) and I/O performance.

Communication inside the node can happen between the CPU (host) and a GPU or between two GPUs. For the CPU-GPU transfers, we measure the bandwidth of single transfers, and two concurrent transfers in opposite directions. Concurrent transfers are common in applications that overlap the transfer of the output data of a previous kernel execution with the transfer of the input for the next one. Host threads are pinned to one CPU to force the allocation of host memory buffers in the closer memory partition. We measure the throughput of data transfers between a CPU and a GPU connected to the CPU's I/O hub, and a GPU connected to the I/O hub of another CPU.

For GPU to GPU communication we also measure the throughput of both single and concurrent bidirectional transfers between two GPUs. In this case is to simulate a data exchange pattern commonly found in multi-GPU applications. The GPU to GPU experiments are performed using two GPUs sharing the same PCIe bus, and using two GPUs installed in different PCIe buses, to quantify the advantage of using peer-to-peer data transfers, when available, over transfers requiring intermediate copies to the host memory.



(a) System with two GPUs. (b) System with four GPUs. (c) System with a four GPUs in a NUMA system.

Figure 1: Multi-GPU system topologies.

To simulate cluster level applications, we also measure the performance of intra-node GPU to GPU copies when each GPU is managed by a different MPI process and the transfer is performed using the MPI primitives. We use two different versions of the OpenMPI library [9] (1.4.5 and 1.7) to perform the inter-node GPU to GPU transfers, to evaluate the advantages of using a GPU-aware MPI implementation.

Communication between the nodes (inter-node) is performed over the network. In our experiments, we focus on the MPI over Infiniband network as the most representative case in multi node GPU machines. We study the performance of the transfers between GPUs of different nodes, and between the host memory of one node and a GPU of another node.

Many applications do file I/O, either to store the input data and the results, or as a backing store for big datasets. We measure the performance of transfers between GPU and the disk for both traditional Hard Disks (HDD) and Solid State Drives (SSD). To avoid the interference of the Linux operating system's file caches in our measures, we use the `O_DIRECT` flag of the POSIX `open` function to ensure that data is actually read from and written to the disk instead of the cache.

Depending on the scenario, different types of transfers are possible. In experiments that transfer data between a CPU and a GPU, or between two GPUs on the same PCIe bus (including transfers using GPU-aware version of the OpenMPI library), the data are transferred to the destination directly. Whenever these direct transfers are not possible, staging is performed through the host memory, either by copying the whole memory structure to the host and then to the destination memory, or by performing pipelined transfers through host (or hosts) using the double-buffering technique.

B. Evaluation Systems

Table I shows the systems used in our evaluation. *System A* is a single-node NUMA machine with two quad-core CPUs running at 2.4 Ghz and four GPUs. CPUs are interconnected with Intel QPI and each one is connected to a I/O hub shared by two GPUs (Figure 1c). The system's hard disk is connected to the I/O controller of one of the CPUs.

	System A	System B	System C
CPUs	2 Intel E5620	2 Intel E5649	1 Intel i7 3820
GPUs	4 Nvidia C2070	2 Nvidia M2090	4 Nvidia C2050
GPUs per PCIe	2	1	1
Disk	7200 rpm HDD SATA 3 6 Gbps	SSD SATA 3 6 Gbps	7200 rpm HDD SATA 3 6 Gbps
DRAM	1333 Mhz	1333 Mhz	1333 Mhz
NICs	n/a	MT26428 40 Gbps	n/a
CUDA Driver	304.88	285.05.09	304.88
Linux Kernel	3.2.41	2.6.32	3.2.41
GCC	4.6.3	4.6.1	4.6.3
NVCC	4.2	4.1	5.0

Table I: System configurations

System B is a GPU cluster. Each node has two six-core CPUs, running at 2.53 GHz, also connected with a QPI bus. They have two GPUs, and two Infiniband network adapters. The system has two PCIe 2.0 buses, each shared by one GPU and one network adapter (similar to Figure 1c, but replacing one GPU for the NIC). This system uses a SSD disk, which is able to deliver a higher throughput than traditional HDDs.

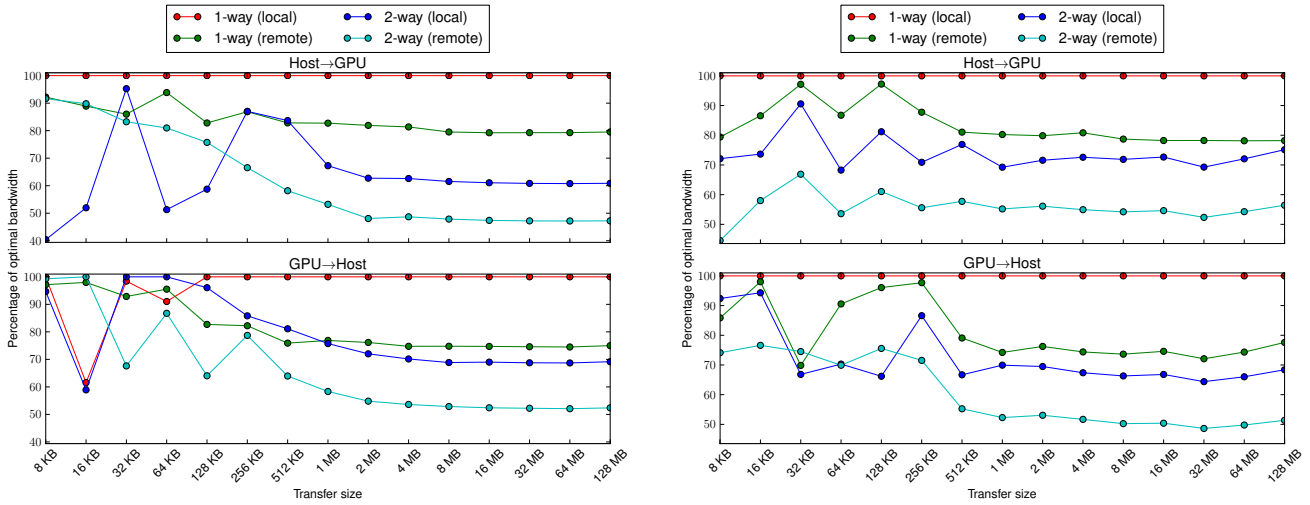
System C is a single-node machine with one quad-core Intel Core i7 3820 CPU clocked at 3.6 Ghz. The main difference of this system is that it has 4 GPUs that are connected in pairs to two different PCIe switches, connected to a single PCIe root complex (Figure 1b). This allows peer-to-peer communication between all the GPUs in the system.

IV. Locality Evaluation

To compare the different locality configurations, we normalize the results to the best achieved bandwidth for each data size. This allows us to show the percentage of the optimal bandwidth achieved by each configuration.

A. Locality-aware Intra-node GPU Communication

First, we study how locality affects data transfers between a GPU and the host memory. We measure the bandwidth of data transfers to/from a GPU connected to the PCIe hub local to the CPU, and a GPU connected to the PCIe hub of the other CPU (remote). We measure both single data



(a) System A.

(b) System B.

Figure 2: Percentage of the best achieved bandwidth for different locality configurations when transferring data between a GPU and host memory (2-way: two concurrent transfers in opposite directions).

transfers (one-way) and two concurrent transfers in opposite directions (two-way). Figure 2 shows the results obtained in systems A and B. System C is not considered for this test, because all GPUs are connected to the same PCIe hub. Both systems follow a similar trend, where the impact of locality increases with the transfer size until 8 MB, where all the configurations reach a steady state. As expected, one-way local transfers are the best performing in both systems, while one-way remote transfers reach from 70% to 80% of the optimal bandwidth for large transfer sizes, depending on the direction of the copy. In system A, host to GPU copies are faster than their respective copy from GPU to host, however in system B GPU to host copies have better bandwidths. Even though they have similar GPU and CPU models, they have different hardware characteristics that can favor one direction or the other. Two-way data transfers are the slowest ones, obtaining up to 70% of the optimal measured bandwidth.

Figure 3 shows the results obtained for intra-node GPU to GPU data transfers in systems A and C. Since system B has only two GPUs, it cannot be used to study the effects of locality for GPU to GPU transfers. In this case, the local and remote meaning is not the same in both systems. In system A, local transfers are between GPUs installed in the same PCIe hub, and remote transfers are done using two GPUs connected to different PCIe hubs. Thus, in this system peer-to-peer copies are only available for local data transfers. In system C, all four GPUs are connected to the same PCIe hub, allowing peer-to-peer copies between all the GPUs. In this case, we consider transfers between two GPUs in the same PCIe bridge to be local, and transfers across bridges to be remote. Besides the GPU to GPU copy functions provided by CUDA, we also measure the bandwidth of our own GPU to GPU copy implementation, using a double-buffering technique through host memory.

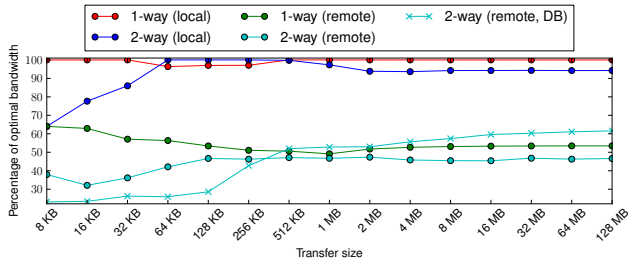
The advantage of peer-to-peer copies is clear in both systems. In system A, the best bandwidth obtained by remote transfers is only 65% of the optimal one, because they have to be implemented as a two step copy using intermediate stagging in the host memory. In system C, since all the copies can use peer-to-peer transfers, the bandwidth difference is small (up to 15%).

Most HPC applications are programmed using MPI to run on modern clusters. To model this scenario, we also measure how intra-node transfers perform when the GPUs are managed by different MPI processes. Figure 4 shows the percentage of the optimal bandwidth achieved for local MPI process communication in two different hardware configurations. In both configurations, locality-aware policies provide the optimal bandwidth when MPI processes are running in the same node. These experimental results show that if processes are not correctly placed (i.e., the MPI process uses the CPU and GPU in the same PCIe domain), local MPI inter-processes communication can be up to 75% slower than when topology-aware placement is used.

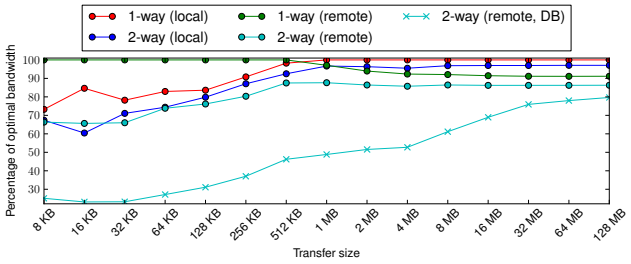
Results in Figure 4 also show that ensuring the MPI processes locality also provide large benefits for MPI inter-process communication when each process is assigned to a GPU from different PCIe domains. There is a potential bandwidth penalty of up to 70% if MPI communication requires crossing PCIe domains several times on each data communication operation. This shows the viability of auto-tuning policies to assign GPUs in the same PCIe domain to MPI processes that most frequently exchange data.

B. *Locality-aware Disk I/O*

We measure the performance of disk reads and writes in System A (HDD disk) and System B (SSD disk), using

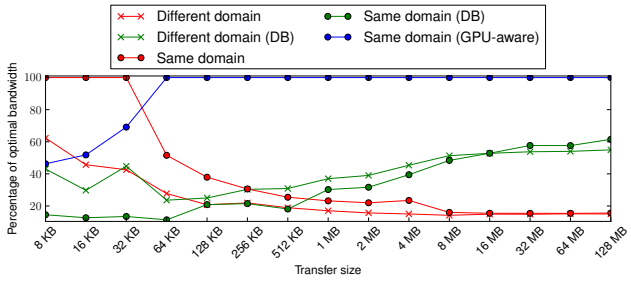


(a) System A

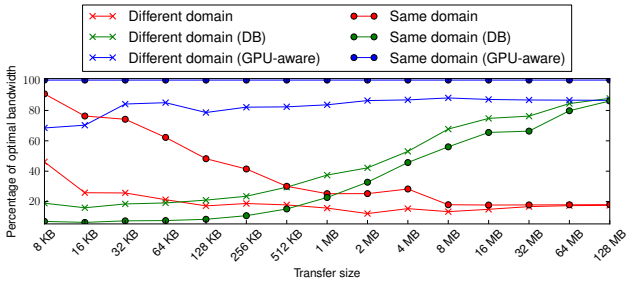


(b) System C

Figure 3: Percentage of the best achieved bandwidth for different locality configurations in GPU to GPU data transfers. (2-way: two concurrent transfers in opposite directions, db: hand-written double-buffered implementation).



(a) System A. A domain is a PCIe hub.



(b) System C. A domain is a PCIe bridge.

Figure 4: Percentage of the best achieved bandwidth for different locality configurations in intra-node GPU to GPU data transfers using MPI (each GPU managed by a different MPI process).

different locality configurations for both the disk and the GPU, with respect of the CPU where the host thread is running. System C is not studied because it can be treated as a subset of System A for these experiments. For reasonable big files (1 MB or more), both systems show a small impact of locality on the performance of GPU-Disk data transfers (less than 20%). For file writes smaller than 1 MB, considerable variability is seen due to inability to isolate disk traffic generated by our experiments from the traffic generated by the operating system services.

V. Data Transfers Double-Buffering Evaluation

A. Intra-node GPU to GPU communication

Figure 5 shows the two-way transfers between remote GPUs in System B. One labeled DB is our hand implementation of the double-buffered transfer while the other uses the CUDA provided call, which is also internally double buffered, but with the fixed buffer size. We compare the two methods to conclude that for the transfer sizes of 512 KB or more, CUDA provided implementation achieves from 85% to 95% of the auto-tuned transfer performance, while small transfer sizes achieve higher performance than double-buffering due to the extra overhead of synchronization and inefficiency of issuing small transfer sizes.

B. Disk I/O

The bandwidth provided by disks is much lower than that of the PCIe interconnect. However, results show that using

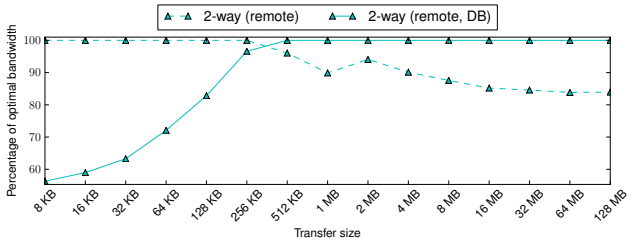


Figure 5: Efficiency of the auto-tuned and non-tuned double-buffered transfers between GPUs.

double buffered transfers is still beneficial compared to simple transfers staging all the data in the host memory. Double-buffered transfers are slower for transfer sizes smaller that 512 KB. After this point, double-buffering delivers 10-20% more throughput than simple transfers for both transfer directions. Tests on the SDD disk exhibit the same behavior.

The effect of the size of the buffers used in the implementation of the double-buffered transfer follows a similar trend in both systems. The bandwidth increases (from 20% of the optimal bandwidth) as the buffer size increases, reaching the maximum achieved bandwidth at a specific buffer size. After this point, the bandwidth achieved by larger buffer sizes stays close the optimal value. The buffer size at which the optimal bandwidth is reached depends on the disk type and the transfer size. The HDD disk reaches the optimal bandwidth with smaller buffer sizes, due to its limited

transfer rate compared to the SSD one (e.g., 128 KB vs. 4 MB in 128 MB transfers).

C. Inter-node GPU Communication

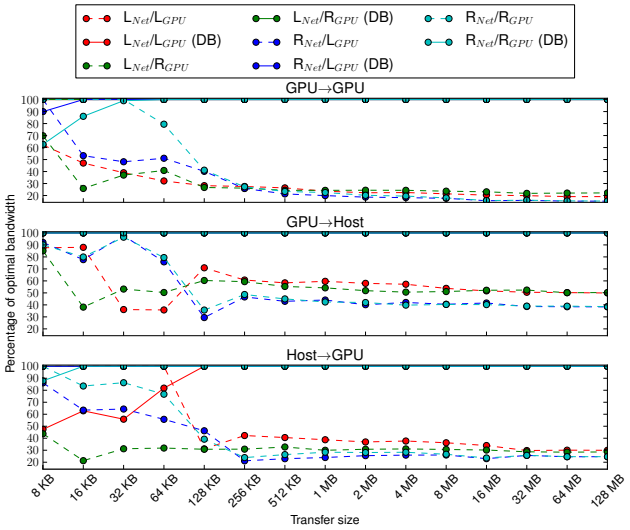


Figure 6: Performance of MPI transfers

Figure 6 shows the performance of inter-process MPI communication for data transfers across CPU and GPU memories. These results show that topology-aware placement of MPI processes barely affects the performance of remote MPI data transfers. The results in Figure 6 also shows the benefits of double-buffering remote communication; in all cases double-buffering provides the optimal bandwidth for data transfer sizes larger than 128 KB. The only case where double-buffering performs worse than direct transfers is when communicating data smaller than 128KB from the network to the GPU memory, if both network interface and the GPU are attached to the same PCIe bus. The optimal buffer size in all cases is 2 MB. When larger buffers are used, the bandwidth slowly decreases, while smaller buffers provide sub-optimal bandwidths due to the overhead of the staging.

VI. Conclusions

In this paper we have analyzed the importance of auto-tuning applications to match the system topology and select the most adequate data transfer mechanism to achieve optimal bandwidth on data transfers between CPUs, GPUs, and nodes within a cluster. To evaluate the potential impact of each of these optimizations on different scenarios, we have developed and executed synthetic benchmarks that stress common data transfer scenarios. We have presented the experimental results of running these benchmarks on three different systems that represent most of the existing heterogeneous architectures currently used in HPC environments.

Our experimental results highlight the benefits of auto-tuning applications to match the node topology. By ensuring that applications run on the CPU and GPUs connected to the same PCIe bus, the data transfer bandwidth can improve up to 70%. We have shown, that locality-aware policies have little impact when communicating MPI processes running on different nodes. We have also shown that double-buffering is key to achieve optimal performance on inter-process communication when processes are running on the same node as well as when running on separate nodes. However, double-buffering only provides marginal gains when transferring data between GPUs and I/O devices, such as disks, due to the low read/ write bandwidth of existing I/O devices.

The results presented in this paper show that auto-tuning is key to optimize data transfers in existing heterogeneous HPC systems. Our experimental data also shows that simple auto-tuning policies, such as different code paths depending on the data transfer size, can provide impressive performance gains.

Acknowledgements

This work is supported by the European Commission through TERAFLUX (FP7-249013) and Mont-Blanc (FP7-288777) projects, NVIDIA through the CUDA Center of Excellence program, and the Spanish Ministry of Science and Technology through Computacion de Atlas Presteciones V and VI (TIN2007-60625 and TIN2012-34557).

References

[1] "TOP500 list - November 2012," 2012. [Online]. Available: http://top500.org/list/2012/11/100/

[2] "The green 500 list, november 2012 edition," 2012. [Online]. Available: http://www.green500.org/lists/green201211

[3] *CUDA C Programming Guide*, NVIDIA, 2012.

[4] *The OpenCL Specification*, 2009.

[5] *PCI Express Base 3.0 Specification*, PCI-SIG, 2010.

[6] V. Kindratenko *et al.*, "Gpu clusters for high-performance computing," in *CLUSTER '09*.

[7] "NVIDIA GPUDirect," NVIDIA. [Online]. Available: https://developer.nvidia.com/gpudirect

[8] *MPI-3: A Message-Passing Interface Standard*, Message Passing Interface Forum, 2012.

[9] E. Gabriel *et al.*, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *European PVM/MPI Users' Group Meeting*, 2004.