On the Acceleration of Graph500: Characterizing PCIe Overheads with Multi-GPUs

Mayank Daga

AMD Research
Advanced Micro Devices, Inc., USA
Mayank.Daga@amd.com

Abstract. Graphics Processing Units (GPUs) have fundamentally altered the approach to parallel computing despite the substantial PCIe overheads that they manifest. In order to maximize performance-per-dollar, systems are now being deployed with multiple GPUs in the same node. However, multiple GPUs exacerbate the PCIe overheads by inflicting additional data-movement performance penalties when moving non-local data.
In this paper, we first evaluate the PCIe performance loss that occurs due to improper affinity between CPUs and GPUs, using a PCIeBandwidth benchmark specifically developed for systems with multiple GPUs. Our experiments demonstrate that the performance loss can be up to 2.5× on a single GPU and up to 4.4× when four GPUs are used. We then leverage our learnings from the PCIe studies to optimize and accelerate the Graph500 benchmark on a 4-GPU, multi-socket system. Our optimization techniques include binding the CPU threads to appropriate cores as well as the careful partitioning of data for every GPU. We achieve a speedup of 1.8× over a single GPU implementation.

1 Introduction

The exigent demands of emerging applications to maximize performance while staying under power and thermal constraints have made graphics processing systems (GPUs) ubiquitous [8, 9]. Since GPUs have traditionally resided on PCI Express (PCIe), additional overheads are incurred for host-to-GPU data transfers and vice versa. As a consequence, GPU applications are oftentimes bottlenecked by the PCIe data transfers [6]. Despite this fact, GPUs have achieved immense popularity due to a unique combination of performance and energy efficiency. GPUs have also been recognized to play an important role on the path to extreme scale computing as evident by the fact that half of the top ten supercomputers on the Top500 list use GPUs as accelerators [1].

In order to maximize performance-per-dollar, systems are now being deployed with multiple GPUs. However, multiple GPUs bring in additional challenges particularly with respect to optimal PCIe performance. This is because of the complex mapping between the GPUs which require data across the PCIe and the CPU cores which are responsible for doing the direct memory access (DMA) of data. In such systems, data-transfers occur at full PCIe bandwidth between *local* CPU cores and GPUs. However, data-transfer to a remote GPU is subject to a significantly worse bandwidth because of additional on-chip interconnects. The onus of placing data on appropriate DMA nodes lies on the application developer.

In this paper, we first evaluate the cost of moving data from the host to GPU at various combinations of mapping between the CPU cores and GPUs. For doing so, we use an indigenous `PCIeBandwidth` benchmark which allows us to (i) bind data to a particular DMA node, (ii) bind a CPU thread to a particular core and then (iii) use that thread to transfer data to any particular GPU or multiple GPUs. We then leverage our learnings from the PCIe studies to optimize the Graph500 benchmark [2]. Graph500 uses breadth-first search (BFS) as its main kernel and tracks the fastest data-intensive supercomputers in the world. Almost half of the total execution time in Graph500 is spent in moving data to the GPU [5, 10]. Therefore, managing the data- and thread-bindings is imperative to achieve good performance. Our implementation of Graph500 uses the `hybrid++` algorithm which partitions the computation between CPU and GPU and hence, good PCIe performance is crucial [7, 11]. We implement the following optimization techniques: (i) partition data in chunks for each GPU, (ii) manually map those chunks to local DMA nodes and, (iii) bind CPU threads to a particular core which is local to the GPU.

The PCIeBandwidth benchmark demonstrates that the performance degradation due to incorrect mapping between CPU cores and GPU can be up to 2.5× when a single GPU is used and up to 4.4× when four GPUs are used. Our optimized Graph500 implementation on a 4-GPU multi-socket system achieves a speedup of 1.8× compared to a single GPU implementation.

The rest of the paper is arranged as follows. Section 2 provides a background on the Graph500 benchmark and the algorithm that we use to compute BFS. Section 3 describes our experimental setup followed by the characterization studies of PCIe in Section 4. Section 5 presents the optimizations and evaluation of the multi-GPU Graph500 implementation. Section 6 presents the conclusion of this work.

2 The Graph500 Benchmark

Graph500 uses a breadth-first search (BFS) kernel to rank the top data-intensive supercomputers in the world. The benchmark provides freedom to the developers to use any custom BFS algorithm for the purposes to computing the score in the form of giga-transferred-edges-per-second (GTEPS). We use the hybrid++ BFS algorithm in our Graph500 implementation [4, 7].

The hybrid++ algorithm is suitable for heterogenous processors as it enables us to choose between a combination of traversal directions (top-down or bottom-up) and the platform of execution (CPU or GPU). The top-down traversal is a serial algorithm that executes best on CPUs. Whereas the bottom-up traversal exposes immense parallelism and is suitable for GPUs. The hybrid++ algorithm uses an online heuristic to seamlessly choose the appropriate algorithm and the suitable core for every iteration of BFS. The heuristic leverages input graph characteristics as well as traversal information from prior BFS iterations to make optimal decisions. A high-level illustration of hybrid++ is shown in Figure 1.

Since, hybrid++ requires a data-copy whenever there is a change in the BFS traversal algorithm, achieving good PCIe performance is imperative for an efficient overall execution of Graph500. Therefore, understanding and characterizing the PCIe effects with multiple GPUs is key to the acceleration of Graph500.

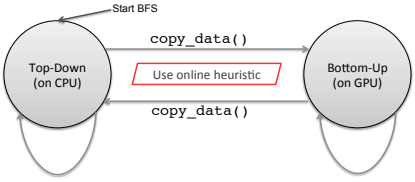


Fig. 1: High-level block diagram illustrating hybrid++ BFS algorithm [7].

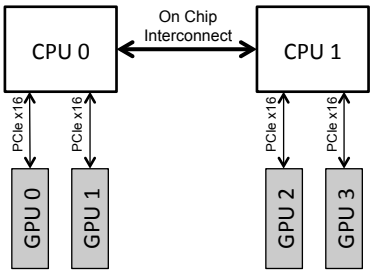


Fig. 2: High-level block diagram illustrating a dual socket multi-GPU system with 4 GPUs connected across PCIe x16 Gen3.

3 Experimental Setup

The schematic diagram of the system we use for our experiments is shown in Figure 2. The system consists of a Intel® Xeon® E5-2667 v3 CPU which has two sockets. There are four PCIe Gen3 x16 lanes with two such lanes connected to each socket. The GPUs reside on the PCIe with one GPU per PCIe. Therefore, as per the figure GPUs 0 and 1 are local to CPU 0 and GPUs 2 and 3 are local to CPU 1. Moving data to/from a remote GPU occurs over the on chip interconnect and hence, adversely affects performance.

We performed all our experiments on an AMD FirePro™ S9150 GPU with ECC disabled. Figure 6 details its important characteristics. The host machine uses 64 GB of DDR3-2133 SDRAM. The GPU was programmed using OpenCL™ v2.0 with the AMD APP SDK v3.0 and AMD FirePro driver v15.20. The operating system was a 64-bit version of CentOS 6.4, kernel version 2.6.32-358.23.2.

The input for Graph500 is a synthetic `rmat` graph [3]. We vary the number of nodes in the graph from 1- to 16-million with an edge-degree of 16.

For measuring the PCIeBandwidth, we use a 512 MB buffer of `float` data-type and the data is moved from host to the GPU. We used 256 threads per workgroup, and all of the performance numbers are an average of 1000 runs.

4 PCIe Bandwidth with Multiple GPUs

We developed a PCIe Bandwidth benchmark to understand the effects of mapping and affinity between the GPUs which need the data and the CPU cores which DMA that data. The pseudocode for the benchmark is shown in Figure 3. Using PCIe Bandwidth we can control the following: (i) mapping data to a particular DMA node, (ii) binding CPU threads to particular cores, (iii) which GPU in the system to use, and (iv) the number of GPUs to use.

We compute the PCIe bandwidth achieved using various mappings of CPU cores and GPUs and characterize the effects of moving local and non-local data across the PCIe in a multi-GPU system. For example as per Figure 2, if data is mapped to CPU 0 and moved to GPU 0 then the transfer is local but if it is moved to GPU 2 then the transfer is remote and occurs via the on-chip interconnect.

```
function AllocateAndRun() {
   // create one thread per gpu
   for g ∈ num_gpus do
      std::thread new_thread (ThreadAllocateAndRun, /* function arguments */ );
   end for
}
function ThreadAllocateAndRun( /* function arguments */ ) {
   // bind this thread to a CPU core
   pthread_setaffinity_np(pthread_self(), /* core to bind */ );
   // allocate host buffer
   TYPE *hostMem = new TYPE[SIZE];
   // allocate device buffer
   cl_mem devMem = clCreateBuffer( /* function arguments */ );
   // move data across PCIe and measure the bandwidth
   clEnqueueWriteBuffer(..., devMem, hostMem, ...);
}
```

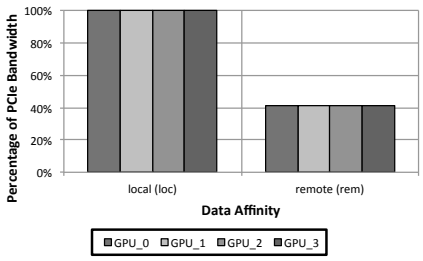
Fig. 3: Pseudocode for PCIe Bandwidth benchmark.

Figure 4 demonstrates the PCIe bandwidth achieved with both local and remote data-transfers when one, two or four GPUs are used. Using a single GPU and moving data to a local GPU, we achieve a unidirectional bandwidth of 12.3 GB/s. We normalize our results to this number, which is the best-case, and present them in Figure 4a. From the figure, we note that performance is consistent no matter which GPU among the four GPUs in the system are used. The difference between local and remote data-transfers is 2.5×. This is because a remote data-transfer adds the latency of on-chip interconnect.

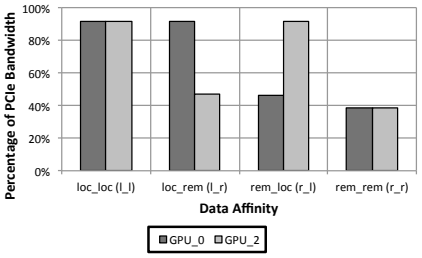
Figure 4b illustrates the PCIe bandwidth achieved when using two GPUs residing on two different nodes, e.g., one among GPUs 0 or 1 and one among GPUs 2 or 3. To achieve the best bandwidth with two GPUs, both the GPUs need to do local data-transfers. This means that the data has to be partitioned and allocated half on each memory node. From the figure, we note that the local bandwidth achieved with two GPUs is 91% of the bandwidth achieved with one GPU due to inherent system overheads. When either of the two GPUs is doing a remote transfer, its bandwidth reduces significantly just as in the case of a single GPU. When both the GPUs are doing remote transfers, the bandwidth achieved is 2.6× lower than the maximum possible.

Figure 4c illustrates the PCIe bandwidth achieved when using two GPUs residing on the same node, e.g., either GPUs 0 and 1 or GPUs 2 and 3. Best bandwidth is achieved when both the GPUs are doing local transfers. However, the bandwidth achieved by both the GPUs is not equal; GPU 3 achieves 6% lower bandwidth than GPU 2. Bandwidth achieved when either of the two GPUs is remote is also erratic. From the figure, when GPU 3 is remote, bandwidths achieved by GPUs 2 and 3 are 81% and 41% of peak, respectively. However, when GPU 2 is remote the bandwidths achieved are only 48% and 57% of peak, respectively. The reason for this is the contention of resources on the same memory node while carrying out the DMA to GPUs. When both the GPUs are remote, the bandwidth achieved is 2.6× lower than than achieved by a single GPU. Therefore, if two GPUs are required to be used, the application developer should ensure that both the GPUs reside on different sockets in a multi-socket system.

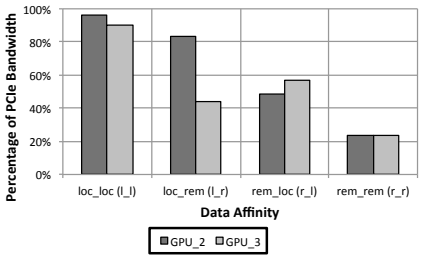
Figure 4d illustrates the PCIe bandwidth achieved when using all four GPUs in the system. As in other cases, the best bandwidth is achieved when all the GPUs are accessing local data. The bandwidth achieved by a single GPU when all the GPUs are active is 88% of what is achievable when only one GPU is active. From the figure, we



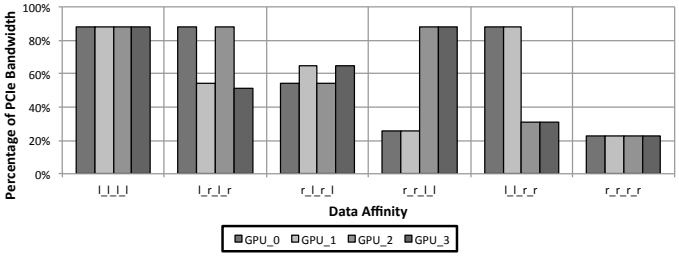
(a) Using 1 GPU.



(b) Using 2 GPUs residing on two different nodes.



(c) Using 2 GPUs residing on the same node.



(d) Using 4 GPUs.

Fig. 4: PCIe bandwidth achieved as measured by the `PCIeBandwidth` benchmark using various combinations of 4 GPUs. All the results are normalized to the bandwidth achieved by a single GPU when moving data local to its node. Local (or `loc (l)`) means the GPU closer to data is used. Remote (or `rem (r)`) means that the GPU farther away from the data is used.

note that inconsistent bandwidths are achieved at different combinations of local and remote data transfers due to the underpinnings of the system which are hidden from the application programmer. When all the GPUs are remote, the bandwidth achieved is worst and is 4.4× lower than that achieved by a single GPU.

From the above results, it is clear that manually managing the data and thread bindings is vital to extract efficient performance when using multiple GPUs. Not controlling the data binding allows the runtime and operating system to freely modify the bindings without programmer knowledge, thereby resulting in suboptimal performance, as shown in Figure 4. A particular feedback to the runtime developers is to make the process of binding threads and data easier by providing APIs to do so.

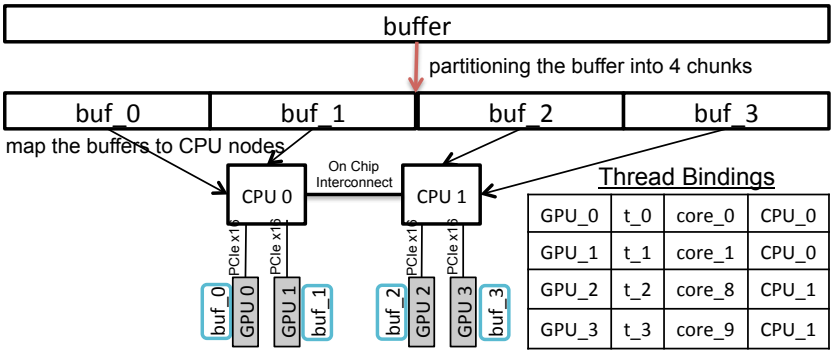


Fig. 5: Partitioning and mapping of the BFS tree buffer in Graph500 to achieve efficient PCIe performance. The buffer is divided into 4 chunks because we are using 4 GPUs in the system.

5 Graph500: Optimization and Evaluation

In this section we describe our optimization strategies for the Graph500 benchmark. Graph500 requires the BFS tree that is generated as part of the search to be preserved as final output. The buffer containing the BFS tree is copied to and from the host in order to keep an updated copy of the resulting output.

All the GPUs computing BFS access the `BFS tree`. However, due to the data-parallel nature of bottom-up BFS they access different regions of the buffer thereby, allowing the buffer to be partitioned among the GPUs. As we note in Section 4, for efficient PCIe performance each chunk of the buffer should be copied to the local GPU. Therefore, we first create as many chunks of the `BFS tree` buffer as the number of GPUs and then map each chunk to the closest CPU node which will do the DMA. For mapping the chunks on the host, we use Pthreads to create a new host thread for every GPU and set its affinity to the core closest to that GPU. For example, a thread `t_0` is created for GPU 0 and its affinity is set to `core_0`. Similarly, for GPU 2, a thread `t_2` is created and its affinity is set to `core_8` because each CPU has 8 cores in our system. Hence, `t_2` is bound to CPU 1. Once the affinities are set, the same threads are used to allocate the GPU buffers using OpenCL APIs and then DMA the data to their local GPU. Since the threads are manually bound to CPU cores, they use the DMA engines on the same node as the CPU thereby, ensuring that the data is moved to the local GPU. Figure 5 illustrates this optimization process.

In Figure 7, we plot the time taken to move the data to and from the GPU, i.e., the copy time, and the time taken to do the actual search on the GPU, while varying the number of GPUs. The GPU time is measured using OpenCL event APIs and copy time is measured using `clock_gettime()` on the host. From the figure, we note that the copy time reduces by 3× when four GPUs are used. This is because as we increase the number of GPUs, the amount of data required to be moved becomes smaller due to partitioning of data, as shown in Figure 5. The scaling of copy time is not perfectly linear because of the inherent runtime and operating system overheads, as outlined in Section 4. Similarly, the GPU time is reduced by 3.5× when using four GPUs thereby,

CPU	Intel® Xeon® E5-2667v3
Cores	16 (8 on each socket)
GPU	AMD Firepro™ S9150
Compute Units (CU)	44
Core Clock Rate	930 MHz
GDDR5 Memory Clock Rate	1250 MHz
Memory Size	16 GB
Peak Memory Bandwidth	320 GB/s

Fig. 6: Overview of the test platform.

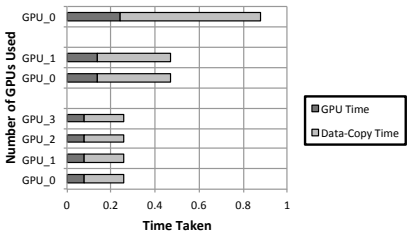


Fig. 7: Scaling of time spent on the GPU and to move data with increasing number of GPUs. This data is computed using the `rmat` graph with 8M nodes and 96M edges.

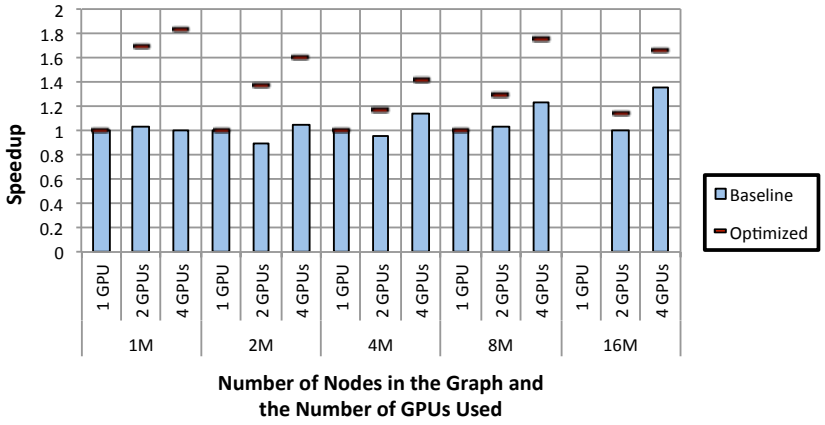


Fig. 8: Effect of optimizations on Graph500. Graph with 16M nodes could not be executed on one GPU due to memory limitations. Baseline is assumed to be performance of a single GPU but for the 16M node-graph, baseline is performance of 2 GPUs

demonstrating almost linear scaling. Therefore, our optimizations are quite effective in improving the performance of multi-GPU implementation of Graph500.

In Figure 8, we demonstrate the impact of our optimizations as we increase the nodes and edges of the input graph, while varying the number of GPUs. For a single GPU, both baseline and optimized numbers are the same because all of our optimizations are targeted towards multiple GPUs. From the figure, we note that the optimizations always improve the performance. Speedup achieved by the optimizations alone can be up to 1.9× as shown for the 4 GPU run of the 1M-node graph. Overall, the speedup achieved compared to a single-GPU implementation is 1.8×, on average.

6 Conclusions

GPUs have become immensely popular for accelerating applications despite the PCIe overheads between the CPU and GPU. Nowadays, systems are being deployed with

multiple GPUs on a single to maximize performance-per-dollar. However, multiple GPUs magnify the performance penalties of PCIe due to the possibility of moving data to non-local resources.

In this paper, we develop a novel PCIeBandwidth benchmark to characterize the PCIe overheads in a multi-GPU system. We also demonstrate the mechanisms for the efficient use of multi-GPU systems. Our experiments portray that a performance loss of up to 4.4× can occur while moving data to four GPUs using incorrect data- and thread-bindings. We then optimize the Graph500 benchmark on a multi-GPU, multi-socket, NUMA system and achieve a speedup of 1.8× over a single GPU.

References

[1] The Top500 Supercomputer Sites, http://www.top500.org
[2] The Graph500 Benchmark (2012), http://www.graph500.org
[3] Bader, D.A., Meyerhenke, H., Sanders, P., Wagner, D. (eds.): Graph Partitioning and Graph Clustering - 10th DIMACS Implementation Challenge Workshop, Georgia Institute of Technology, Atlanta, GA, USA, February 13-14, 2012. Proceedings. Contemporary Mathematics (2013), http://dblp.uni-trier.de/db/conf/dimacs/dimacs2012.html
[4] Beamer, S., Asanović, K., Patterson, D.: Direction-optimizing Breadth-first Search. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. pp. 12:1–12:10. SC '12, Los Alamitos, CA, USA (2012), http://dl.acm.org/citation.cfm?id=2388996.2389013
[5] Checconi, F., Petrini, F.: Traversing Trillions of Edges in Real-time: Graph Exploration on Large-scale Parallel Machines. In: IEEE 28th International Symposium on Parallel Distributed Processing (IPDPS). IEEE (2014)
[6] Daga, M., Nutter, M.: Exploiting coarse-grained parallelism in b+ tree searches on an apu. In: High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:. pp. 240–247 (Nov 2012)
[7] Daga, M., Nutter, M., Meswani, M.: Efficient Breadth-first Search on a Heterogeneous Processor. In: Proceedings of the 2014 IEEE International Conference on Big Data (Big Data) (Oct 2014)
[8] Daga, M., Feng, W., Scogland, T.: Towards Accelerating Molecular Modeling via Multi-Scale Approximation on a GPU. In: Proceedings of the 1st IEEE International Conference on Computational Advances in Bio and medical Sciences (2011)
[9] Owens, J.D., Houston, M., Luebke, D., Green, S., Stone, J.E., Phillips, J.C.: GPU Computing. Proceedings of the IEEE 96(5), 879–899 (May 2008), http://www.idav.ucdavis.edu/publications/print_pub?pub_id=936
[10] Ueno, K., Suzumura, T.: Highly Scalable Graph Search for the Graph500 Benchmark. In: Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing. pp. 149–160. HPDC '12, New York, NY, USA (2012), http://doi.acm.org/10.1145/2287076.2287104
[11] Yasui, Y., Fujisawa, K., Goto, K.: NUMA-Optimized Parallel Breadth-first Search on Multicore Single-node System. In: BigData Conference. pp. 394–402. IEEE (2013), http://dblp.uni-trier.de/db/conf/bigdataconf/bigdataconf2013.html#YasuiFG13