Design Space Exploration of GPU Accelerated Cluster Systems for Optimal Data Transfer Using PCIe Bus

Janki Bhimani
Northeastern University
Email: bhimani@ece.neu.edu

Miriam Leeser
Northeastern University
Email: mel@coe.neu.edu

Ningfang Mi
Northeastern University
Email: ningfang@ece.neu.edu

***Abstract*—Use of accelerators such as GPUs is increasing, but efficient use of GPUs requires making good design choices. Such design choices include type of memory allocation and overlapping concurrency of data transfer with parallel computation. Performance varies with the application, hardware version such as generation of GPU, and software version including programming language drivers. This large number of design decisions makes it nearly impossible to obtain the optimal performance point by directly porting any application. This emphasizes the need for high level design decision guidelines for GPU accelerated cluster systems, applicable to a broad class of applications rather than any specific application.**

This paper proposes novel design guidelines for GPU accelerated cluster systems, to optimize the data transfer from host (CPU) to device (GPU) using the PCIe bus. In particular, we consider design choices offered by NVIDIA GPUs. Our main contribution is to build design guidelines that are applicable to a broad class of applications. We design 27 different versions of the same micro benchmark, where the design choices made by each version is unique. We observe that a speedup of 2.6x can be obtained just by making good design choices.

***Keywords*—Design Guidelines, Data Transfer, PCIe bus, NVIDIA GPUs, Micro Benchmark Versions**

I. Introduction

The use of GPUs for accelerating applications on compute clusters is growing. A well known problem reported in many research projects is that there is no magical performance improvement just by porting application code to accelerators [1]. Many specific, appropriate design choices need to be made to achieve good speedup. There exist many projects which focus on how to obtain the maximum speed-up for a particular application on a particular platform [2]–[5]. But to maintain a proper balance between performance, portability and programmablility, it is necessary to have design guidelines which can be applied to a class of applications. The current state of the art gives good guidelines on how to optimize the kernel (i.e., computation on accelerator) and shared resources while computing [6], [7], but in most applications, performance can also be improved by improving data transfer (i.e., communication between CPU and accelerator).

Any parallel application running on GPU accelerated cluster systems has two main phases: (i) communication of the data between host (CPU) and device (GPU) and (ii) computation on the device. We assume that the GPU (device) is attached to the host (CPU) via a PCIe bus and describe choices for NVIDIA GPUs in rest of the paper. A parallel application is either limited by performance of the accelerator (compute bound) or by PCIe bus throughput (communication bound). Sometimes an application is neither compute nor communication bound and we call this scenario *balanced*. Applications with the same bound are expected to behave similarly, because they contend for similar resources. We can easily determine the bound nature of an application either by using the roofline model [8], or by executing the real application and measuring the time spent in computation versus communication.

A particular bound nature will benefit from a specific type of host memory allocation. Different available options in the NVIDIA tool set include pinned, pageable or unified memory allocation. To achieve better application performance, implementing an application using the appropriate memory allocation technique is important. We explain each type of memory allocation technique and explore their effects on performance for the three types of bounds in Section IV.

Improvement in performance can be obtained by increasing concurrency. This can be achieved by using streams of data. In order to derive performance improvement through the use of more streams, it is important that sufficient resources are available. Use of more streams does not guaranty performance improvement and may even cause a decrease due to resource contention. In Section V, we explore the performance impact of increasing the number of streams. As a stream is a sequence of operations that execute in issue-order on the GPU, overall performance is also dependent on the issue-order of the instructions of each stream. This issue-order management is called stream scheduling. In Section V, we also explore different types of stream scheduling schemes.

With the availability of new hardware, a user may be unsure whether an investment in newer versions will give improvement in performance. There exists a design choice between different versions of hardware; in particular, we consider upgrading of the PCIe bus version. This decision is closely related to memory mapping and in Section VII, we explore this design decision.

The major contributions of this paper are:

- Development of twenty-seven versions of the same micro benchmark with each using a unique set of design choices.
- Performance analysis of applications with respect to their bound nature.
- Performance analysis for different types of available memory allocation techniques for each bound class of application.
- Performance analysis of the effect of increasing concurrency by increasing the number of streams and different stream scheduling schemes for each bound class of application and each type of memory allocation technique.

978-1-5090-3525-0/16/$31.00 ©2016 IEEE

- Performance analysis of the latest two available hardware choices of PCIe 2.0 and PCIe 3.0 for each bound class of application and each type of memory allocation technique.

The rest of this paper is organized as follows. In Section II, we describe the related work. In Section III, we explain our hardware configurations and testing benchmark versions. In Section IV and V, we explore types of concurrent memory mapping and computation and communication concurrency for micro benchmark versions. Section VI validates our observations derived from micro benchmark versions by using *vector add* application. In Section VII, we explore the upgrading decision of PCIe bus version. Finally, in Section VII we summarize our results as guidelines.

II. Related Work

Many applications are being accelerated on GPUs. Using GPUs for general purpose scientific computing has allowed researchers to solve larger and finer-grained datasets [9], [10]. However, many researchers have concluded that it is difficult to get performance improvement [1], [11]. There are very few studies of design space exploration for multiple applications [6], [12], [13]. These studies do not give any guidelines applicable to broad class of applications like all compute bound applications or communication bound applications. Lack of general guidelines to achieve speed-up limits the use of accelerators such as GPUs. There are many recent studies that focus on design space exploration for performance improvement of a particular application [2]–[5], [14]–[17].

The current state of the art also reflects that most GPU studies have concentrated on the optimization of the kernel, including optimization related to occupancy, hashing of kernel, data access patterns, coalescing, SIMD and computation concurrency [12], [13]. A few studies also concentrate on performance with respect to data size and dimension [6]. More recent research studies optimization of shared resources like NoC and shared memory [7], [18]. Currently, there are very few studies which explore the design space of data transfer between host and device [19], [20].

Given the rapid evolution of improved techniques of data transfer such as unified memory allocation, and the rapid development of new hardware like PCIe 3.0, there is a need for design space exploration concentrating on data transfer and applicable to a wide class of applications. To the best of our knowledge, our research is the first to explore the available design space of optimal data transfer, considering recent developments such as unified memory and PCIe 3.0, and applicable to classes of applications depending on the bound nature of an application.

III. Hardware Configurations and Testing Benchmark Versions

A. Hardware Topology

We consider a platform consisting of a CPU - GPU layout as shown in Figure 1. The accelerator (GPU) is connected to the host (CPU) using Peripheral Component Interconnect Express (PCIe). We consider two types of accelerator namely the NVIDIA Tesla K20 GPU and the NVIDIA Tesla K40 GPU. Table I gives the detailed configurations for our platforms.

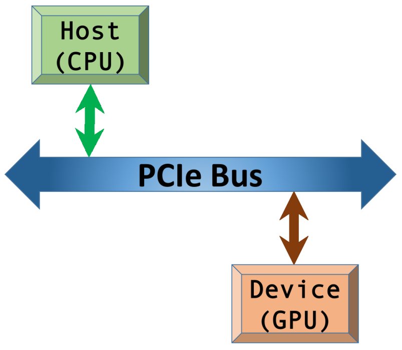

Fig. 1: Topology of hardware layout

TABLE I: Hardware (P.I - Platform I & P.II - Platform II)

	P.I	P.II
CPU type	Intel Xeon E5-2650	Intel Xeon E5-2650
CPU speed	2.00 GHz	2.00 GHz
CPU #cores/node	40	48
CPU mem/node	128 GByte	128 GByte
OS	Linux EDT 2012	Linux EDT 2012
GPU type	Tesla K20	Tesla K40
GPU #cores/node	2496	2880
PCIe Version	2.0	3.0
PCIe #lanes	x16	x16
Prog. language	CUDA-C	CUDA-C

B. Micro Benchmark Versions

We modify the micro benchmark presented in [21] to make different versions in order to examine the available design choices. Functionally these micro benchmark versions perform a host to device transfer of the null array. Detailed operations of the kernel with no streams is shown in Algorithm 1. The kernel performs addition of 1 (as a trigonometric function) to each element of the array in parallel as the main task (Algorithm 1, lines 4 to 8), followed by a *for* loop of dummy calculations introduced to control computational intensity of the kernel (Algorithm 1, lines 10 to 15). We use number of iterations of the dummy loop to vary the bound nature of this micro benchmark between compute intensive, communication intensive and balanced. Finally the modified array is copied back to the host where we check correctness of each implemented version. We report time of host to device memory transfer, kernel execution and device to host memory transfer in all our results.

In order to further explore the three types of concurrent memory copies, we develop different versions of this micro benchmark, i.e. using pinned memory, pageable memory and unified memory.

Computation and communication can be overlapped by using streams of data. The basic concept of using streams is to divide the data into small blocks and process each block in a pipelined fashion. We use a unique offset for each stream to represent the unique index of input array *a* where each stream starts operation.

For a kernel using streams, CUDA operations are dis-

Algorithm 1: No Streams Kernel

```
Input  Array a, int max_iter
Output  Modified Array a
//Initialize unique thread id as i
int i = threadIdx.x + blockIdx.x*blockDim.x;
float x = (float)i;
float s = sinf(x);
float c = cosf(x);
a[i] = a[i] + sqrtf(s*s+c*c);
//fake compute inject
for(int p=0; p <max_iter; p++)
{
float zs = sinf(x);
float zc = cosf(x);
float z = sqrtf(zs*zs+zc*zc);
}
```

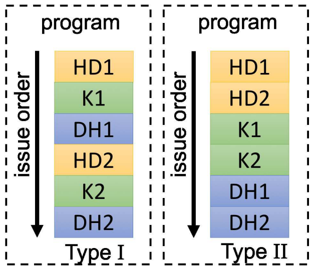

Fig. 2: Types of stream scheduling scheme (HD1 - host to device transfer for first stream, K1 - kernel compute for first stream and DH1 - device to host transfer for first stream)

patched to hardware in the sequence they were issued. The architecture considered contains one compute engine queue and two copy engine queues - one for host to device and one for device to host. Stream scheduling manages the order of issue of the streams to the compute engine queue, host to device data transfer queue and device to host data transfer queue. We explore two types of scheduling scheme in our experiments. Figure 2 shows an example of these two types of stream scheduling schemes, assuming two streams. Scheduling scheme type I iterates over the total streams to transfer data from host to device, compute the kernel and transfer data from device to host for each stream. Scheduling scheme type II iterates over the total streams to transfer data from host to device, then it iterates over the total streams to compute the kernel, and finally it again iterates over the total streams to transfer data from device to host. There are many other possible scheduling schemes but we limit ourselves to exploring these two different types.

In summary, we have twenty-seven versions of the micro benchmark which are developed to be compute bound, communication bound and balanced in combination with three types of concurrent memory copy techniques (i.e., pinned memory, pageable memory and unified memory) and three types of kernel (i.e., kernel with no streams and a kernel with streams issued in two different scheduling schemes).

IV. Concurrent Memory Mapping

In order to use an accelerator such as a GPU, we first need to transfer the required data to the device (GPU) from the host (CPU). There are three ways in which we can map memory to facilitate this transfer: 1) pinned memory, 2) pageable memory, and 3) unified memory.

Pinned memory is memory allocated using the *cudaMallocHost* function, which prevents the memory from being swapped out and provides improved transfer speeds. Pinned memory enables DMA (Direct Memory Access) on the GPU to request transfers to and from the host memory without the involvement of the CPU. In other words, pinned memory is stored in the physical memory (RAM) so the GPU can fetch data without the help of the host. This can only be used for applications where the data to be transferred fits within the host memory space. Pinned memory is much more expensive to allocate and deallocate but provides higher transfer throughput for large memory transfers compared with pageable memory.

Pageable memory is memory allocated using the *malloc* function. To allow programmers to use a larger virtual address space than is actually available in RAM, CPUs implement a virtual memory system called pageable memory. It is also called non-locked memory because in its implementation, a physical memory page can be swapped out to disk. When the host needs that page, it is loaded back from the disk. The drawback with pageable memory transfers is that the memory transactions are slower, i.e., the bandwidth of the PCI-e bus that connects CPU and GPU is not fully exploited.

Unified memory is memory allocated using the *cudaMallocManaged* function. Unified memory creates a pool of managed memory that is shared between the CPU and GPU, bridging the CPU-GPU device. Managed memory is accessible to both the CPU and GPU using a single pointer. The key is that the system automatically migrates data allocated in Unified Memory between host and device, so that it looks like CPU memory to code running on the CPU and like GPU memory to code running on the GPU. The drawback with unified memory transfers is that the memory transactions involve memory management overhead and memory transfer overhead.

This raises an obvious question: *which type of memory is best to use?* We answer this question under three broad classes of application: compute bound, communication bound and balanced. Our observations are based on experiments conducted on P.II (see Table I). We vary data size to obtain 100 observations, where each observation is a mean derived from 1000 runs. In Figure 3, we show the results.

From Figure 3 (a) we observe that, when our application is compute bound, it is advantageous to use unified memory. This is because, when we have a lot of computation to be performed, the memory management overhead and the memory transfer overhead are effectively hidden between the computations to yield better overall performance. We also observe that there is not much performance difference between pinned and pageable memory.

From Figure 3 (b) we observe that, when our application is communication bound, unified memory gives the worst performance. This may be because, in the case where we are

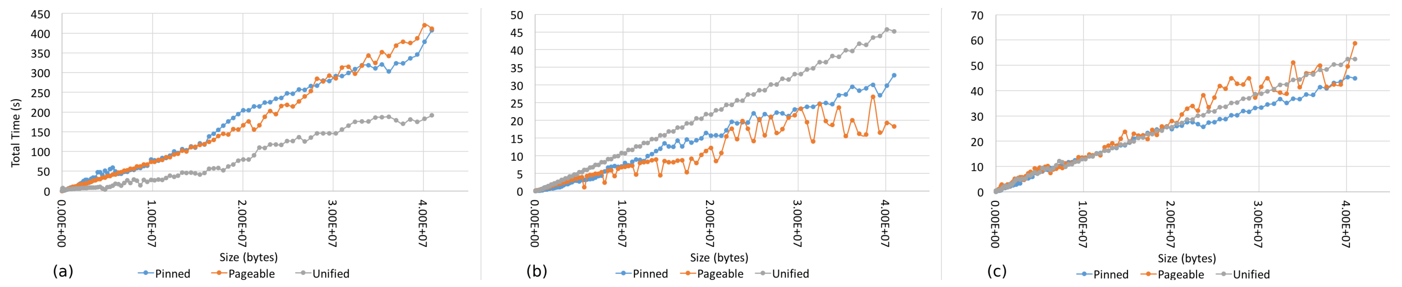

Fig. 3: Total time using memory allocation of pinned, pageable and unified type for, (a) compute bound, (b) communication bound, (c) balanced

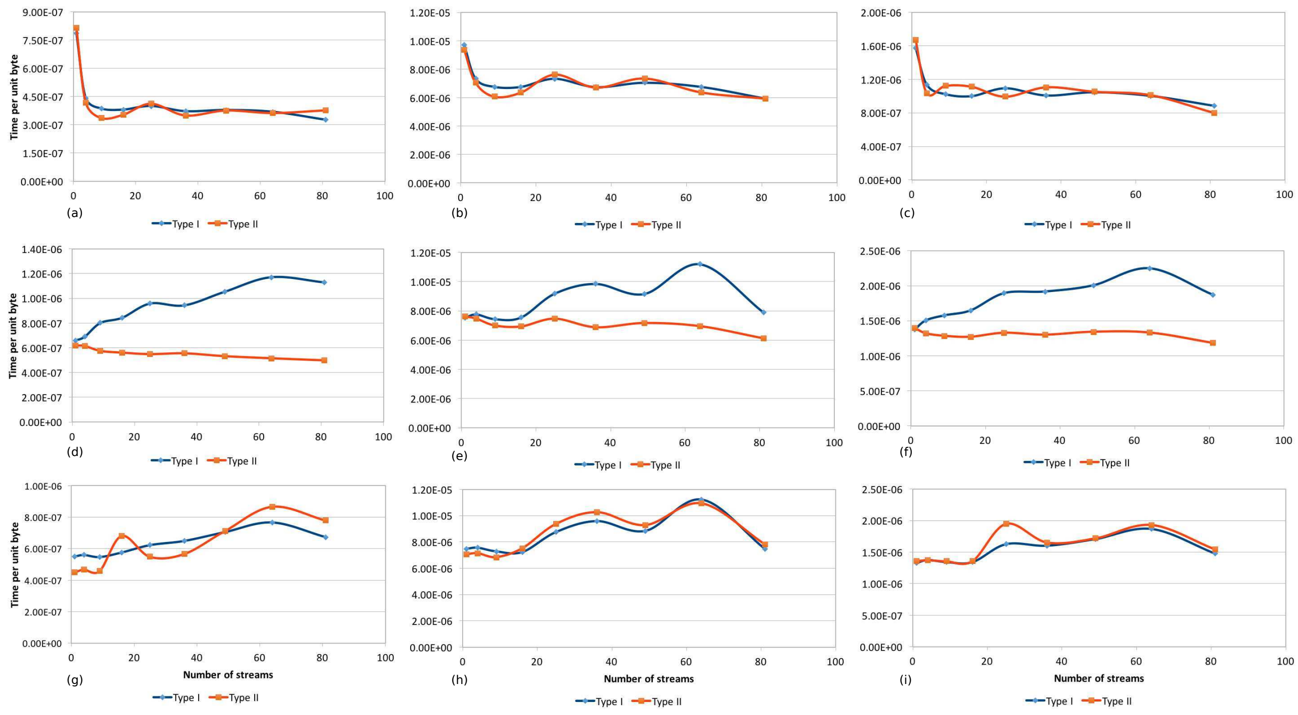

Fig. 4: Time taken for unit byte w.r.t. number of streams for (a) pinned memory allocation with compute bound (b) pinned memory allocation with communication bound (c) pinned memory allocation with balanced (d) pageable memory allocation with compute bound (e) pageable memory allocation with communication bound (f) pageable memory allocation with balanced (g) unified memory allocation with compute bound (h) unified memory allocation with communication bound (i) unified memory allocation with balanced

already communication bound, unified memory management adds to memory transfer overhead. Pinned and pageable memory perform almost the same.

Figure 3 (c) gives our observed results for the class of applications which are neither compute bound nor communication bound. In such a scenario we observe that using pinned memory could yield slightly better performance over pageable and unified memory, for larger size of data.

V. Computation and Communication Concurrency

In order to achieve better performance using a GPU, we would like to simultaneously perform parallel computation on the GPU and communication of the data between CPU and GPU. A parallel pipeline behavior can be obtained by using streams. A stream is a sequence of operations that execute in issue-order on the GPU. Thus, it is a programming model to increase concurrency. CUDA operations in different streams may run concurrently and CUDA operations from different streams may be interleaved. Basically, concurrency can be achieved by two types of overlap: (1) an overlap of computation of different streams and, (2) an overlap of computation of one stream with the communication of another stream. In order to derive performance improvement from the use of more streams, it is important that sufficient resources be made available. Using more streams does not guaranty performance improvement and may even cause a decrease in performance due to resource contention. We explore the performance when increasing the number of streams for compute bound, communication bound and balanced scenarios with pinned, pageable and unified types of memory allocation.

Since the stream is a sequence of operations that execute in issue-order on the GPU, overall performance also depends on the issue order of the instructions of each stream. This issue order management is called stream scheduling. Stream scheduling manages the order of issue of the streams to the compute engine queue, the host to device data transfer queue and the device to host data transfer queue. We explore two types of scheduling scheme in our experiments. Figure 2 shows an example of these two types of stream scheduling schemes, assuming two streams. Scheduling scheme type I iterates over the total streams, to transfer host to device, compute the kernel and transfer device to host for each stream. Scheduling scheme type II iterates over the total streams to transfer host to device, then it iterates over total streams to compute kernel, and finally it again iterates over total streams to transfer device to host. There are many other scheduling schemes, but we limit ourselves to exploring these two types.

Our observations are based on experiments conducted on Platform P.II (Table I). We vary the number of streams with constant data size of 1638400 bytes and observe the total time per unit byte. Each observation is a mean derived from 1000 runs to reduce the effect of run time variance.

Figure 4 (a), (b) and (c) show the experiments for pinned memory allocation. Irrespective of our scheduling scheme, we get performance improvement to approximately 10 streams and then with further increase in the number of streams, there is very little performance improvement.

From Figure 4 (d), (e) and (f), we observe that for pageable memory allocation, using scheduling scheme type I decreases the performance. This may be because the required data to transfer from host to device by a stream might be paged out. For example see Figure 2 - Type I, when host to device transfer for first stream (HD1) is issued then the entire block of data is paged, which may also contain data for host to device transfer for second stream (HD2). But according to scheduling scheme type I, kernel of first stream (K1) and device to host transfer of first stream (DH1) are issued before HD2. So the data required by HD2 is paged out by the time it is issued. This leads to the latency of fetching the data again from disk for each stream. We also observe for scheduling scheme type II, the performance increases slowly with increasing number of streams. Thus while using pageable memory allocation it is advisable to wisely choose the stream scheduling scheme.

In the last three figures we analyze unified memory allocation. From Figure 4 (g), (h) and (i), we observe that performance decreases with increase in the number of streams. Thus the use of streams with unified memory is not advisable. The reason behind such a steep decrease in performance might be that memory management overhead and memory transfer overhead of unified memory becomes larger with increasing number of streams.

VI. Validation

The observations of Sections IV and V, which we derive by using micro benchmark versions, are validated using a *vector add* application. The vector add application initializes two vectors on the host and adds them on the device in an embarrassingly parallel fashion. It is a communication bound application, as there is not much computation.

We reprogram the vector add application using pinned, pageable and unified memory. Figure 5 shows the total time per unit byte with respect to different sizes of data for different types of memory allocation. The observations from Figure 5 validate the observations from Figure 3 (b). Both show that using unified memory for a communication bound application is not advisable.

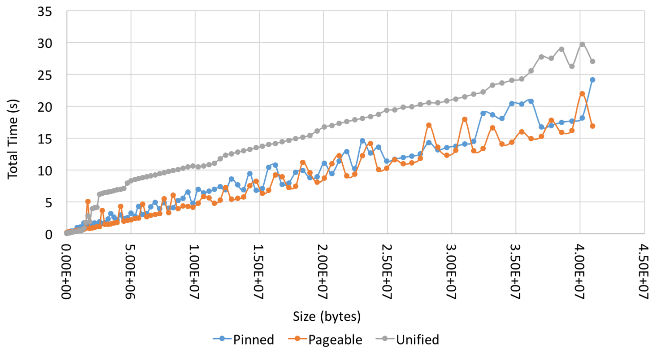

Fig. 5: Total time using memory allocation of pinned, pageable and unified type for vector add application

TABLE II: Summary of PCI Express Interface Parameters

	`PCIe 3.0`	`PCIe 2.0`
Base Clock Speed	8.0GHz	5.0GHz
Data Rate	1000MB/s	500MB/s
Total B/W (x16 link)	32GB/s	16GB/s
Data Transfer Rate	8.0GT/s	5.0GT/s

From Figure 6 (a), (b) and (c) and from Figure 4 (b), (e) and (h), we observe that both shows similar results. Note that using streams is recommended with pinned memory and not recommended with unified memory. For pageable memory, performance is sensitive to stream scheduling policy. Thus we validate that our observations are applicable to a broad class of applications.

VII. Upgrade a PCIe Bus or Not

In this section we experimentally analyze the potential data throughput and performance between PCIe 3.0 and PCIe 2.0. We have PCIe 3.0 in platform P.II and PCIe 2.0 in platform P.I. The theoretical interface parameters of both versions are given in Table II.

PCIe 3.0 features a number of interface architecture improvements compared to PCIe 2.0. This raises the expectation of performance improvement achievable by upgrading to the latest hardware. But, as the latest hardware comes with a cost, it is important to decide wisely whether to invest in an upgrade. We present our observations from results for the three broad classes of applications: compute bound, communication bound and balanced as well as for the three types of memory allocation: pinned, pageable and unified memory.

Our observations are based on experiments conducted on platforms P.I and P.II (Table I). We use a single stream of data. We vary data size to obtain 100 observations, where each observation is a mean derived from 1000 runs. We consider total communication time taken per unit byte as our metric of comparison. Table III summarizes our observations.

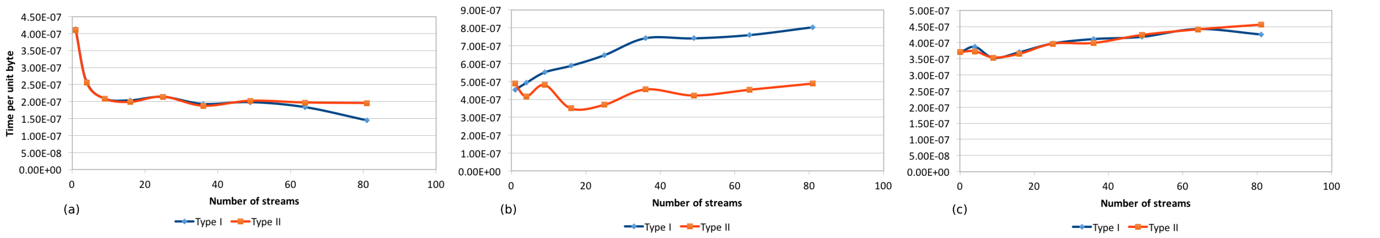

Fig. 6: Time per unit byte w.r.t. number of streams using memory allocation of pinned, pageable and unified type for vector add application

TABLE III: Comparison of PCIe bus version 2 (V.2) and version 3 (V.3) on metric of total time, where V.3 <V.2 means V.3 takes less time than V.2 and V.3 $\sim$ V.2 means no performance improvement with V.3

Bound	Comp.	Comm.	Equi.
Pinned	V.3 $\sim$ V.2	V.3 $\sim$ V.2	V.3 $\sim$ V.2
Pageable	V.3 <V.2	V.3 <V.2	V.3 <V.2
Unified	V.3 <V.2	V.3 <V.2	V.3 <V.2

From Table III, it is notable that for pinned memory allocation, PCIe.3.0 does not show any improvement over PCIe.2.0. We also observe that PCIe.3.0 performs approximately twice as fast compared to PCIe.2.0 for pageable memory allocation and unified memory allocation.

VIII. Conclusions and Future Work

We have presented novel design guidelines to optimize the data transfer from host (CPU) to device (GPU) using the PCIe bus. We developed different versions of a micro benchmark in which the same operations are performed but design choices made by each version is unique. The suite contains twenty-seven micro benchmark versions exploring compute bound, communication bound, balanced, pinned memory allocation, pageable memory allocation and unified memory allocation. Our experiments result in the following major guidelines:

- For computation bound applications, using unified memory allocation is recommended.
- For communication bound applications, using pinned memory allocation for small size of data and pageable memory for larger size of data is recommended.
- It is safe to use any type of memory allocation for balanced applications.
- For pinned memory allocation, use of streams is advisable, independent of the bound nature of applications and stream scheduling policy.
- For pageable memory allocation, performance is sensitive to stream scheduling policy.
- For unified memory allocation, use of streams is not advisable.
- An upgrade to PCIe 3.0 from PCIe 2.0 is advisable if user applications are taking advantage of unified memory or pageable memory

In the future, we plan to keep reforming our guidelines as newer software techniques and hardware versions are introduced. We also plan to keep updating our micro benchmark versions as technological generations changes.

IX. Acknowledgements

This work was partially supported by National Science Foundation Career Award CNS-1452751 and AFOSR grant FA9550-14-1-0160.

References

[1] R. Vuduc, A. Chandramowlishwaran, J. Choi, M. Guney, and A. Shringarpure, "On the limits of GPU acceleration," in *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*. USENIX Association, 2010, pp. 13–13.

[2] A. Jooya, N. Dimopoulos, and A. Baniasadi, "GPU design space exploration: NN-based models," in *Communications, Computers and Signal Processing (PACRIM)*. IEEE, 2015, pp. 159–162.

[3] J. Bhimani, M. Leeser, and N. Mi, "Accelerating K-Means clustering with parallel implementations and GPU computing," in *High Performance Extreme Computing Conference (HPEC), 2015 IEEE*. IEEE, 2015, pp. 1–6.

[4] L. Cheng and G. Butler, "Accelerating search of protein sequence databases using CUDA-enabled GPU," in *Database Systems for Advanced Applications*. Springer, 2015, pp. 279–298.

[5] T. Xu, P. Cockshott, and S. Oehler, "Acceleration of stereo-matching on multi-core CPU and GPU," in *High Performance Computing and Communications*. IEEE, 2014, pp. 108–115.

[6] S. Al-Kiswany, A. Gharaibeh, E. Santos-Neto, G. Yuan, and M. Ripeanu, "StoreGPU: exploiting graphics processing units to accelerate distributed storage systems," in *Proceedings of the 17th international symposium on High performance distributed computing*. ACM, 2008, pp. 165–174.

[7] J. Fang, Z.-Y. Leng, S.-T. Liu, Z.-C. Yao, and X.-F. Sui, "Exploring heterogeneous NoC design space in heterogeneous GPU-CPU architectures," *Journal of Computer Science and Technology*, vol. 30, no. 1, pp. 74–83, 2015.

[8] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[9] N. K. Govindaraju, S. Larsen, J. Gray, and D. Manocha, "A memory model for scientific algorithms on graphics processors," in *IEEE conference on Supercomputing', ACM, New York, NY, USA*, 2006, p. 89.

[10] S. Popov, J. Günther, H.-P. Seidel, and P. Slusallek, "Stackless KD-tree traversal for high performance GPU ray tracing," in *Computer Graphics Forum*, vol. 26, no. 3. Wiley Online Library, 2007, pp. 415–424.

[11] A. Moerschell and J. D. Owens, "Distributed texture memory in a multi-GPU environment," in *Computer Graphics Forum*, vol. 27, no. 1. Wiley Online Library, 2008, pp. 130–151.

[12] D. Schaa and D. Kaeli, "Exploring the multiple-GPU design space," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 1–12.

[13] W. Jia, K. A. Shaw, and M. Martonosi, "Stargazer: Automated regression-based GPU design space exploration," in *Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on*. IEEE, 2012, pp. 2–13.

[14] A. Athanasopoulos, A. Dimou, V. Mezaris, and I. Kompatsiaris, "GPU acceleration for support vector machines," in *WIAMIS 2011: 12th International Workshop on Image Analysis for Multimedia Interactive Services, Delft, The Netherlands, April 13-15, 2011*. TU Delft; EWI; MM; PRB, 2011.

[15] T. Basten, E. Van Benthum, M. Geilen, M. Hendriks, F. Houben, G. Igna, F. Reckers, S. De Smet, L. Somers, E. Teeselink *et al.*, "Model-driven design-space exploration for embedded systems: The octopus toolset," in *Leveraging Applications of Formal Methods, Verification, and Validation*. Springer, 2010, pp. 90–105.

[16] P. Bakkum and K. Skadron, "Accelerating SQL database operations on a GPU with CUDA," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. ACM, 2010, pp. 94–103.

[17] P. Michel, J. Chestnutt, S. Kagami, K. Nishiwaki, J. Kuffner, and T. Kanade, "GPU-accelerated real-time 3D tracking for humanoid locomotion and stair climbing," in *Intelligent Robots and Systems, 2007*. IEEE, 2007, pp. 463–469.

[18] V. Allada, T. Benjegerdes, and B. Bode, "Performance analysis of memory transfers and GEMM subroutines on NVIDIA tesla GPU cluster," in *Cluster Computing and Workshops*. IEEE, 2009, pp. 1–9.

[19] G. Agrawal and S. Chakradhar, "Automatic and efficient data host-device communication for many-core coprocessors," in *Languages and Compilers for Parallel Computing: 28th International Workshop, LCPC 2015, Raleigh, NC, USA, September 9-11, 2015, Revised Selected Papers*, vol. 9519. Springer, 2016, p. 173.

[20] Y. Fujii, T. Azumi, N. Nishio, S. Kato, and M. Edahiro, "Data transfer matters for GPU computing," in *Parallel and Distributed Systems (ICPADS)*. IEEE, 2013, pp. 275–282.

[21] M. Harris, "How to overlap data transfers in CUDA C/C++," Dec. 2012. [Online]. Available: https://devblogs.nvidia.com/parallelforall/how-overlap-data-transfers-cuda-cc/