GPUdrive: Reconsidering Storage Accesses for GPU Acceleration

Mustafa Shihab, Karl Taht and Myoungsoo Jung
Computer Architecture and Memory Systems Laboratory
Department of Electrical Engineering
The University of Texas at Dallas

Abstract

GPU-accelerated data-intensive applications demonstrate in excess of ten-fold speedups over CPU-only approaches. However, file-driven data movement between the CPU and the GPU can degrade performance and energy efficiencies by an order of magnitude as a result of traditional storage latency and ineffectual memory management. In this paper, we first analyze these two critical performance bottlenecks in GPU-accelerated data processing. We then study design considerations to reduce the overheads imposed by file-driven data movements in GPU computing. To address these issues, we prototype a low cost and low power all-flash array designed specifically for stream-based, I/O-rich workloads inherent in GPUs. As preliminary evaluation results, we demonstrate that our early-stage all-flash array solution can eliminate 60% $\sim$ 90% performance discrepancy between memory-level GPU data transfer rates and storage access bandwidth by removing unnecessary data copies, memory management, and user/kernel-mode switching in the current system software stack. In addition, our all-flash array prototype consumes less dynamic power than the baseline storage array by 49%, on average.

1 Introduction

The general purpose graphics processing unit (GPGPU or GPU – we use GPU hereafter) has risen to prominence as an accelerator with comparatively low power consumption. In a GPU, there exist hundreds of processing cores per chip, and many of those cores share execution controls rather than maintaining their own instruction registers. This single-instruction multiple-data (SIMD) architecture is capable of accelerating data-intensive applications that seek to perform identical operations on numerous pieces of data via thread-level and data-level parallelism. Furthermore, data-intensive applications can also offload computational kernels from the CPU to the GPU, therefore introducing massive device-level parallelism. As a consequence of this efficacy in managing application execution over many levels of parallelism, GPU-accelerated data-intensive applications demonstrate 2x $\sim$ 55x speedups [15, 11, 12, 1] and big data analytics such as MapReduce improve by 16x $\sim$ 72x compared to a CPU-only approach [2, 14].

While a powerful computation acceleration technology, these GPU-based accelerations are unfortunately limited in many cases by significant communication overheads associated with uploading and downloading data sets, kernel launching, and synchronization processes. Specifically, latency related to data transfer between the CPU and the GPU exceeds actual GPU's data processing time [3] by 200% $\sim$ 5000%. As a result, both industry and academia are turning their focus on reducing data transfer costs between CPU-memory and GPU-memory. As an example, NVIDIA's direct access (called GPUDirect) [9] and zero-copying data movement mechanisms [7] allow the GPU to directly access a PCIe device if they co-exist under a same root complex that connects the CPU and the memory controller hub to PCIe. Similarly, a new CPU-GPU synchronization technique [8] shortens the offload latency by employing a fine-granularity data transfer, early kernel launch, and a proactive data return mechanism. In addition, CUDA's pinned memory (also known as non-pageable memory) [10] and unified virtual addressing (UVA) [13] reduces CPU intervention to IOMMU in managing memory-level data transfers.

While these optimizations work to alleviate memory copy operations between the CPU and the GPU, there is a dearth of similar optimizations to enable efficient storage-level operations. Part of this is merely a raw difference in device-level latencies such as a storage I/O access, which is in practice orders of magnitudes slower than a memory access, making it one of the greatest challenges GPU-accelerated data-intensive applications need to address.

Further, the input data that GPU-kernels will process

should be available in pageable/non-pageable host-side memory before the non-preemptive direct memory access (DMA) begins to transfer them to GPU [6]. We observe that, in cases where the GPU applications need to fetch the target data sets from the underlying storage drive, the data transfer rates between the CPU and the GPU degrade by about 95%, which might not be acceptable in many GPU computing applications. We also observe that the corresponding storage accesses introduce additional power consumption (12 $\sim$ 19 watts) in GPU computing per computing node.

To address these challenges, we prototype *GPUdrive*, an inexpensive, low power, high-performance storage system, specifically designed to address the data-transfer performance disparity between the storage and the GPU. As a first pass and baseline example of GPUdrive, we construct an all-flash array by employing multiple low-end SSDs, which exposes the aggregate SSD performance through a conventional thin interface. To accelerate GPUdrive performance, we optimize existing system software stacks for both the storage and the GPU devices, which helps to eliminate unnecessary data copies, memory management, and user/kernel-mode switching overheads.

Our preliminary evaluation results show that, the early-stage of our GPUdrive completely eliminates the performance disparity for request sizes greater than 16 MB, and its data-transfer rates accounts for 60% $\sim$ 90% of the GPU data-transfer rates with much less power consumption when GPU applications access the underlying storage mediums.

2 Overview

In this section, we briefly detail typical GPU architecture, data paths, and the software stack on the host which facilitates all the data movements.

2.1 Architecture

GPU and CPU. Figure 1 illustrates a high-level view of the conventional computer organization (left) and a connected GPU architecture (right). A GPU consists of multiple execution units (EU), each of which has small data-parallel compute cores. These EUs are connected to multiple memory modules, hereafter referred to as *GPU-memory*, through the on-device caches and integrated memory controllers (MC). In parallel, the host-side CPU can have multiple memory modules, called *CPU-memory*, which are connected to its own on-chip *memory controller hub* (MCH). The communications and data movements between the GPU and the CPU are co-managed by the host-side memory controller in MCH and the GPU-side on-device microprocessor. Due to the highly parallel and throughput-oriented nature of

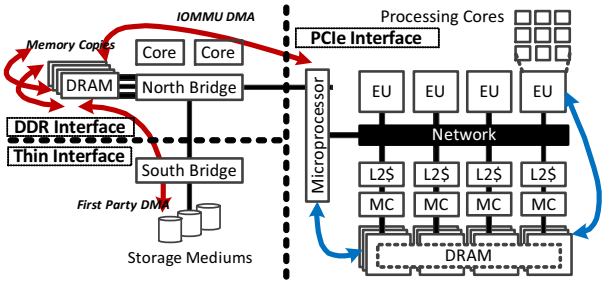


Figure 1: A high-level view of a conventional computer and a GPGPU architecture.

the GPU, such a heterogeneous GPU-CPU architecture offers excellent performance, often an order of magnitude better than CPU-only computing for large-scale and data-intensive workloads [8]. However, the overheads associated with communication and data movement prevent the GPU from being able to take full advantage of its inherent massive parallelism in such workloads. Put succinctly, moving data between the storage and the GPU is the bottleneck for data-intensive applications that might otherwise benefit from the properties of the GPU.

The Data Paths on GPU and Storage. Since traditional spinning disks are a thousand times slower than memory devices, they are connected to the off-chip *I/O controller hub*, IOH (distant from the CPU), through thin storage interfaces. When a GPU-kernel requires data from the underlying storage drive, it therefore must traverse various interface boundaries including: GPU-memory, CPU-memory, thin interfaces, and the storage device itself. These numerous hops through layers ill-tuned to handle the raw bandwidth and low latency GPUs expect makes storage data transfers cumbersome and slow.

2.2 System Software

I/O communication methods and data paths between GPU and storage are completely desperate because of their different functionalities and responsibilities (computation oriented and data-management oriented). As a consequence, in conventional operating systems (OS), there exist two discrete and different I/O and GPU runtime libraries, which co-exist on the same host machine and are both utilized in GPU applications. Figure 2 illustrates these software stacks for both traditional storage and the GPU. All storage accesses and file services are managed by modules on the *storage software stack*, while all GPU-related activities including memory allocations and data transfers are handled by modules on the *GPU software stack*.

Storage Software Stack. Walking through how these stacks interact for storage commands, when a GPU application calls an I/O runtime library through a POSIX interface, the runtime library stores all the user-level contexts and jumps to the underlying *virtual file system*

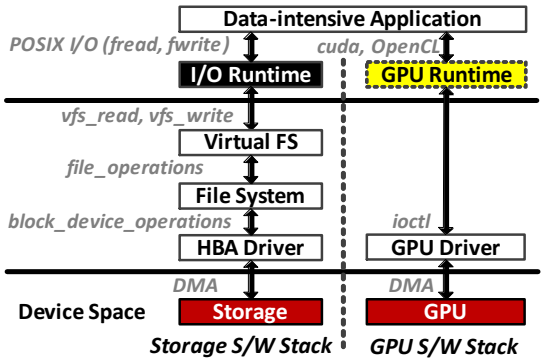


Figure 2: System software stack for storage and GPU device.

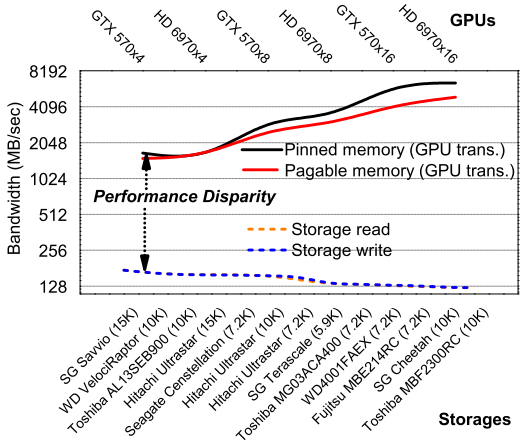


Figure 3: Performance disparity observed between real enterprise storage and GPU devices. Note that data-transfer-rates degrade by 2000% $\sim$ 8000% when the GPU applications access the storage devices.

(VFS), which is a kernel module in charge of managing all standard UNIX file system calls. VFS can select an appropriate native file system such as EXT4 and initiate file I/O requests. This is done by calling an I/O system function pointer extracted from a *file-operation* function container, a kernel-level OS data structure. The native file system then checks the actual physical location associated with the file requests and composes block-level I/O service transactions by calling another function pointer that can be retrieved from a *block-device-operation* data structure. Finally, the host block adopter (HBA) driver issues I/O requests to the underlying storage drives. Once I/O services are completed, the data is returned to the GPU application via the aforementioned modules but in reverse order.

GPU Software Stack. The GPU runtime library, on the other hand, is mainly responsible for executing GPU-kernels and copying data between CPU-memory and GPU-memory. Unlike the storage software stack, this GPU runtime library creates GPU device commands at the user-level and directly submits them with the target data to the kernel-side GPU driver via an *ioctl*, which is another system call enabling device-specific I/O operation. Depending on the GPU commands, the kernel-side GPU device driver can map a kernel-memory space (CPU-memory) to GPU-memory space and/or translate addresses such as CPU-memory's virtual addresses to physical addresses of the GPU-memory space. These GPU-specific data transfer activities include *base address register* (BAR) and *graphics address remapping table* (GART) management. Once the address translations/mappings are completed, the on-device microprocessor of the GPU facilitates data movement between CPU-memory and GPU-memory.

It should be noted that, in current GPU computing, redundant and unnecessary memory copy activities exist due to multiple hops on file-driven data movements. These overheads, imposed by these system software stacks, cannot be addressed by employing low-level communication technique or optimization such as GPUDirect and zero-copy since these two discrete and different system stack are in ignorance of each other.

3 Storage Accesses in GPU Computing

3.1 Data Transfer with Large Problem Set

Initial Observation. Figure 3 shows the significance of the storage access and file I/O overheads by comparing real data transfer rates on two different GPU devices (with varying PCIe lane configurations) and fifteen enterprise-scale high-end storage drives. The top and bottom x-axes of the figure indicate data transfer rates for GPUs employing x4 $\sim$ x16 PCIe lanes and high-end enterprise storage drives working on 7K $\sim$ 15K RPM, respectively. For this test, each GPU kernel loads 8MB input matrix. While the memory-transfer test launches only single GPU kernel (for both pinned and unpinned memory), file-driven data movement transfer test executes such GPU kernel by ten times and measure average data transfer rates.

One can observe from this figure that, the data transfer rates between the CPU and the GPU degrade between 94% and 98% due to poor performance of the storage accesses and file operations in cases where the GPU applications need to fetch the target data sets from the underlying storage drive. Specifically, data movement performance degrade from 1.5GB/sec $\sim$ 7.5GB/sec to 80MB/sec $\sim$ 140 MB/sec. This undesirable performance degradation is mainly caused by the *performance disparity* between memory-level transfers and data movements, which should be addressed for GPU to process data with large problem sets.

High-Bandwidth Flash Array Construction. We built the GPUdrive prototype by employing multiple com-

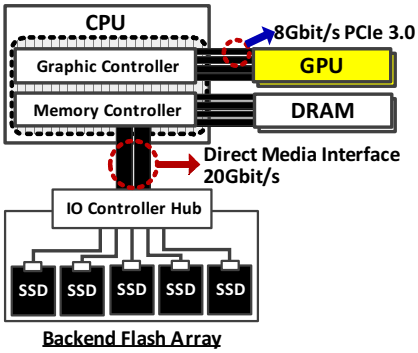


Figure 4: GPUdrive prototype.

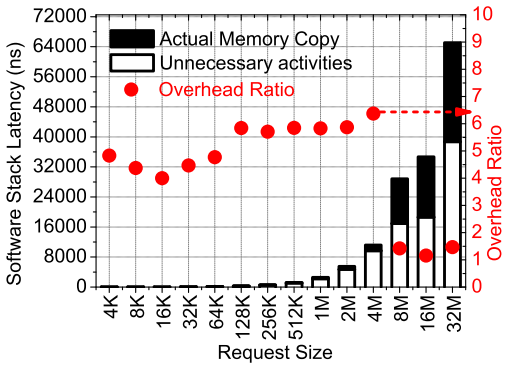


Figure 5: Empirical evaluations for performance overheads of the current system stacks .

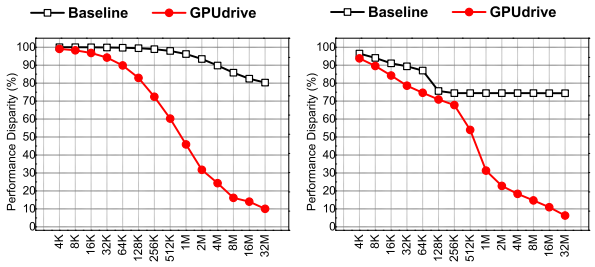
modity SSDs via the low-power SATA 6Gbps interface rather than high-end PCIe based SSDs. This baseline approach in turn enables us to reach a lower cost-per-GB (in the ballpark of what an HDD-based storage array offers) and also consume 49% less power than a conventional enterprise-scale storage array. Looking into how our prototype is architecturally achieved, Figure 4 illustrates GPUdrive prototype organization in distributed GPU system; multiple commodity SSDs are all connected to the I/O controller hub and each of them occupies an individual SATA 3.0 physical channel. Each SSD bidirectionally communicates with the memory controller hub (e.g., the Integrated Memory Controller, or IMC) over the *Direct Media Interface* (DMI), which has similar interface characteristics to PCIe such as multiple lanes, point-to-point links, and full-duplex transmission. GPUdrive exposes the aggregate performance of all these SSD components in its flash array through DMI 2.0, whose data-transfer rate is 20Gbit/s for each direction. This array construction has three advantages over the alternative, a PCIe-based SSD; i) total cost-per-GB and power requirements exhibited by the flash array are much lower, ii) array performance is better than high-end PCIe SSDs, and iii) DMI does not share any PCIe physical ports and bandwidth that distributed GPU devices use. Even though we cannot offer detailed comparison of our prototype and the PCIe-based SSD approach in this paper (due to the space limit), we observe that our early-stage GPUdrive, on the average, consumes 63% lower static power and has 21 times better cost-per-GB benefits while it provides slightly better performance than the PCIe-based SSD approach.

3.2 Disconnected System Stack

Initial Observation. Challenges in current system stacks fundamentally stem from the reality that, the storage and the GPU devices are completely disconnected from each other, and are therefore maintained and managed by different software stacks. Consequently, many redundant memory allocations/releases occur in both user-space and kernel-space on the storage and GPU system stacks, as shown in Figure 5. To quantify the impact of this inefficiency, we compare the latency of actual data movements while extracting out execution times unrelated to GPU uploads/downloads. One can see that, the execution times spent performing unnecessary data copies exceeds the latency related to actual data movement by 16% $\sim$ 537%. It should be noted that, since kernel modules are prohibited from directly accessing user memory space[1], the memory management and data copy overheads between kernel-space and user-space imposed by every I/O access are unavoidable when uploading the file-associated data to GPU. Further, the kernel-mode and user-mode switching overheads associated with the data copies further exacerbates these latencies for file-associated data movements.

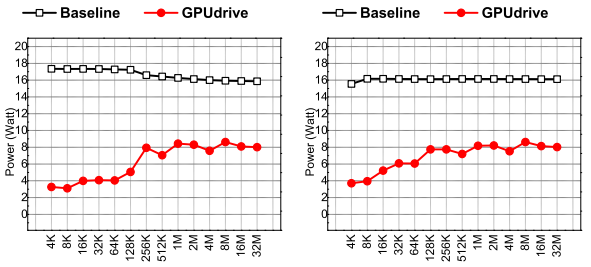
System Stack Optimization. This inefficiency can be addressed if we can remove some the I/O runtime library calls from the user-level GPU applications. Though the host-side data-intensive applications work upon two different runtime libraries, *the target data fetched by the storage stack is only needed by GPU-kernels, not by the host-side data-intensive applications themselves*. Consequently, memory allocations/releases and data copies between user-space and kernel-space on the same CPU-memory are not necessarily required if there is a mechanism to directly forward such data from the storage stack to the GPU stack. Motivated by this observation, our GPUdrive prototype directly forwards the target file-associated data from the storage software stack to the GPU software stack. Specifically, our early-stage solution reads/writes target file contents to the underlying native file systems such as EXT4 using its own kernel buffer and directly upload the target data from it to another kernel buffer mapped to GPU device memory (and vice versa). This system stack, as redesigned by our GPUdrive prototype, almost completely removes the need for unnecessary buffer allocations and redundant memory copies back and forth imposed by user-mode

[1]This is because there is no guarantee that the current kernel is running in the process that the I/O request was initiated



(a) bench-rdrd (b) bench-sqrd

Figure 6: Upload performance disparity analysis.



(a) bench-rdrd (b) bench-sqrd

Figure 7: Upload dynamic power consumption.

and kernel-mode switching and the two different runtime libraries communications. Therefore, our GPUdrive prototype can significantly reduce overheads in managing file-associated data and expose the aggregate performance of the local flash array to the GPU.

4 Performance Improvement

In this section, we will show *preliminary evaluation results*, that provide how our early-stage GPUdrive addresses file-driven data movements with less power consumption.

4.1 Experimental Setup

The GPU device we used is an NVIDIA GTX 480 that employs 480 CUDA cores and 1.2GB DDR3/GDDR5, which provides roughly 133 GB/sec data-transfer rates. This GPU device is connected to our host evaluation platform, which has an Intel Core i7 and 16GB DDR3, through PCI Express 2.0 x16 lanes. While the baseline storage array (RAID-0) employs multiple 7500 RPM enterprise-scale HDDs, we implement our early-stage of GPUdrive using multiple commodity SATA-based SSDs. To evaluate GPU and storage devices, we use benchmark applications by modifying sample code mainly from the NVIDIA CUDA SDK [10] and Intel IOmeter [4]. The *bench-rdrd* and *bench-sqrd* benchmarks generate for uploading file-associated data with random and sequential access patterns, respectively. In contrast, *bench-rdwr* and *bench-sqwr* benchmarks generate download scenarios with random and sequential access patterns, respectively. All tests generate fourteen different request sizes from 4KB to 32MB. Lastly, to measure PCIe data movement performance and storage performance, we utilize the CUDA SDK *bandwidthtest* [10].

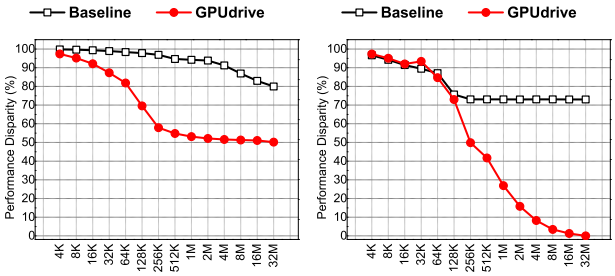
4.2 Upload Performance Analysis

Performance disparity reduction. Figure 6 shows the percentage of performance disparities between the CPU and the GPU during file-associated data movement. As shown, the performance of the baseline storage array only accounts for 4% and 1% of GPU data-transfer rates in cases where the I/O request size is relatively small (4K $\sim$ 256K). Though the baseline can improve read performance by leveraging system-level striping for large I/O sizes, uploading data-transfer rates under *bench-rdrd* and *bench-sqrd* only account for 20% and 25% of potential GPU bandwidths, respectively. In contrast, since GPUdrive takes advantage of system-level SSD parallelism to expose full aggregate performance, improvements when compared against the baseline-storage array range about four times. With large problem data sets, our GPUdrive prototype reduces the performance disparity between the CPU and the GPU on *bench-rdrd* and *bench-sqrd* by 90% and 92%, respectively. Considering the fact that most data-intensive applications perform large I/O requests (ranging from 1MB to 32MB), GPUdrive can almost completely address the performance discrepancy between memory-level transfer and storage accesses.

Dynamic power analysis. Figure 7 shows dynamic power consumed by the baseline storage array and the GPUdrive prototype on both *bench-rdrd* and *bench-sqrd*, respectively. One can see from this figure that, our GPUdrive prototype requires 77% $\sim$ 52% less dynamic power than the baseline storage array while reducing performance disparity between the memory-level transfer and the storage access by an average of 57%. Specifically, the early-stage of GPUdrive consumes low dynamic power ranging from 3.2 watts to 8.6 watts, which can in turn enable low power data processing with large problem sets.

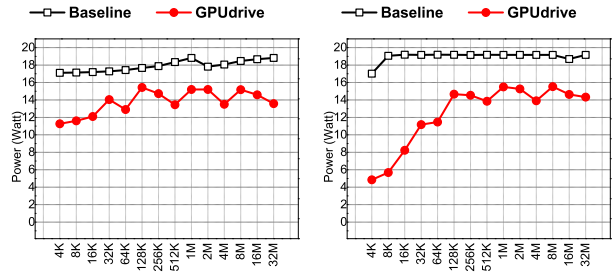
4.3 Download Performance Analysis

Performance disparity reduction. Figures 8a and 8b show the performance disparity between GPU and CPU for *bench-rdwr* and *bench-sqwr* workloads, respectively. While the data movement rate of the baseline storage array only accounts for 1% $\sim$ 30% of the actual data transfer rate of thr GPU, irrespective of which benchmark is employed, our GPU prototype successfully address the performance discrepancy between memory-level transfer and storage access. However, unlike the previous upload performance evaluations, the reduction rates on



(a) bench-rdwr (b) bench-sqwr

Figure 8: Download performance disparity analysis.



(a) bench-rdwr (b) bench-sqwr

Figure 9: Download dynamic power consumption.

bench-rdwr (see Figure 8a) are limited in our GPUdrive prototype, which needs to be addressed in future work. We believe that, this is because the performance of each of commodity SSDs we employed exhibit poor performance on random access patterns, which is also a known problem in SSD research area [5]. In contrast, as shown in Figure 8b, our early-stage GPUdrive successfully removes the performance disparity in the case of large I/O requests (32MB) since it is easier to leverage system-level SSD parallelism for larger I/O sizes. Similarly, these performance improvements of our GPUdrive prototype continue under *bench-sqwr* because most incoming I/O requests can be striped in an interleaving fashion.

Dynamic power analysis. Unlike the previous GPU upload analysis, the power consumption on GPU downloads varies based on I/O sizes as well as access patterns of our GPU workloads. For *bench-rdwr*, the baseline storage array consumes around 18 watts of power, whereas our early-stage of GPUdrive consumes about 13 watts irrespective of the request sizes. On the other hands, for *bench-sqwr*, the GPUdrive prototype require less power than the baseline storage array by an average of 30%. We believe that, the main reason why the power consumption of our GPUdrive prototype on downloads is not as much promising as for uploads is that, in practice, most commodity SSDs are well optimized to hide performance disadvantage of underlying flash medium, which usually requires higher computations and involves more internal components.

5 Conclusion

In this paper, we present performance degradation of GPU-accelerated data processing caused by file-driven data movement. We remedy this shortcoming by designing a flash array and by optimizing the system software stacks associated to GPU computing. We then built an prototype of our design, GPUdrive, using multiple real commodity SSDs and optimized the system stacks. Our prototype successfully reduces the performance disparity between storage accesses and memory-level transfers with less dynamic power consumption than an conventional storage array by 49%.

References

[1] Ranieri Baraglia et al. Sorting using bitonic network with cuda, 2009.

[2] Wenbin Fang et al. Mars: Accelerating mapreduce with graphics processors. *TPDS*, 2011.

[3] C. Gregg and K. Hazelwood. Where is the data? why you cannot debate cpu vs. gpu performance without the answer. In *ISPASS*, 2011.

[4] Intel. *Iometer User's Guide*. 2003.

[5] Myoungsoo Jung and Mahmut Kandemir. Revisiting widely held ssd expectations and rethinking system-level implications. In *SIGMETRICS*, 2013.

[6] S. Kato et al. Rgem: A responsive gpgpu execution model for runtime engines. In *RTSS*, 2011.

[7] Shinpei Kato et al. Zero-copy i/o processing for low-latency gpu computing. In *ICCPS*, 2013.

[8] Daniel Lustig et al. Reducing gpu offload latency via fine-grained cpu-gpu synchronization. In *HPCA*, 2013.

[9] Mellanox. Nvidia gpudirect technology accelerating gpu-based systems. `http://www.mellanox.com/pdf/whitepapers/TB_GPU_Direct.pdf`.

[10] NVDIA. Nvidia cuda library documentation. `http://docs.nvidia.com/cuda/`.

[11] NVIDIA. Gpu-accelerated applications. http://www.nvidia.com/content/tesla/pdf/gpu-accelerated-applications-for-hpc.pdf.

[12] Nadathur Satish et al. Designing efficient sorting algorithms for manycore gpus, 2009.

[13] Tim C. Schroeder. Peer-to-peer and unified virtual addressing. 2013.

[14] Jeff A. Stuart and John D. Owens. Multi-gpu mapreduce on gpu clusters. In *IPDPS*, 2011.

[15] Ren Wu et al. Gpu-accelerated large scale analytics, 2009.