Data Transfer Matters of Real-Time GPU Computing

Yusuke Fujii, Takuya Azumi, and Nobuhiko Nishio
College of Information Science and Engineering
Ritsumeikan University

Shinpei Kato and Masato Edahiro
Department of Information Science
Nagoya University

***Abstract*—**

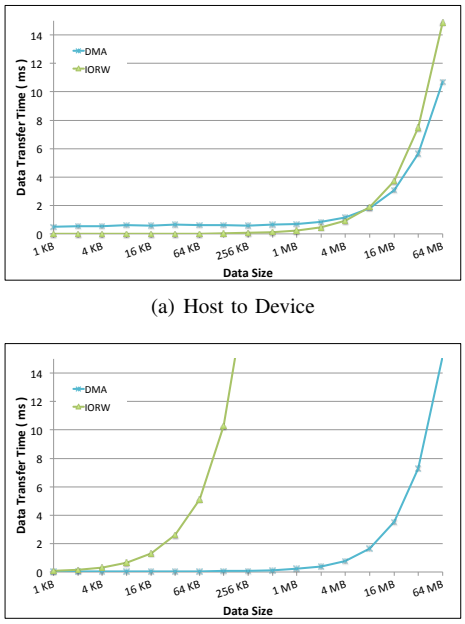
***Keywords*-GPUs; Data Transfer; Latency; Performance**

I. Introduction

Graphics processing units (GPUs) are becoming powerful many-core compute devices. For example, NVIDIA GPUs integrate more than $1,500$ processing cores on a single chip and the peak double-precision performance exceeds 1 TFLOPS while sustaining thermal design power (TDP) in the same order of magnitude as traditional multicore CPUs [16]. This rapid growth of GPUs is due to recent advances in the programming model, often referred to as general-purpose computing on GPUs (GPGPU). Data-parallel and compute-intensive applications receive signficant performance benefits using GPGPU. Currently a main application of GPGPU is supercomputing [19] but there are more and more emerging applications in different fields. Examples include plasma control [8], autonomous driving [13], software routing [6], encrypted networking [7], and storage management [2], [5], [12], [18]. This broad range of applications raises the need of further developing GPU technology to support scalablability of emerging data-parallel and compute-intensive applications.

One of the greatest challenges of GPU computing is the integration of real-time systems. So far the real-time systems community has addressed resource management issues of GPUs [1], [3], [4], [9]–[11]. The main contribution of these work is the scheduling of compute kernels and data transfers associated with the GPU. On the other hand, the basic performance and latency issues of GPUs are not well explored in the real-time systems literature. Given that compute kernels are offloaded to the GPU, their performance and latency are more dominated by compiler and hardware technology. However, an optimization of data transfers must be complemented by system software due to constrants of PCIe devices [12]. Data transfers are also interefered by compelling workload on the CPU, while compute kernels are protected within the GPU. These data transfer issues must be well understood and addressed in order to build predictable soft real-time, if not hard real-time, systems augmented with GPUs.

Figure 1 illustrates the average data transfer times of hardware-based direct memory access (DMA) and memory-mapped I/O read and write access performed on an NVIDIA GeForce GTX 480 graphics card using an open-source Linux driver [12]. Apparently the performance characteristics are not identical between the host-to-device and the device-to-host directions. A very elementary issue of this performance difference has been discussed [12], but there is no clear conclusion on what methods can optimize the data transfer performance, what different methods are available, and what is their implication for real-time systems. Currently we pray that the black-box data transfer methods provided by vendors' proprietary software are well optimized, because the hardware details of GPUs are not disclosed to the public. However, real-time systems must build on predictable disciplines. A better understanding of the data transfer performance for the GPU is an essetial piece of work to facilitate research on GPU-accelerated heterogeneous real-time systems.



(a) Host to Device

(b) Device to Host

Figure 1. Performance of DMA and I/O read/write for the NVIDIA GPU.

Contribution: This paper provides an open investigation of the data transfer methods for the GPU and their empirical comparison. We unveil several data transfer methods for the GPU in addition to well-known DMA and I/O read

and write approaches. The advantage and disadvantage of these methods are also discussed. The contribution of this paper clarifies how to design and implement optimized data transfer methods especially in the context of real-time constraints. We believe that this is a useful contribution to advance real-time technology with GPUs and emerging heterogenous compute devices.

II. Assumption and Terminology

We assume the Compute Unified Device Architecture (CUDA) for GPU programming [17]. A unit of code that is individually launched on the GPU is called a *kernel*. The kernel is composed of multiple *threads* that execute the code in parallel. A unit of threads that are co-scheduled by hardware is called a *block*, while a collection of blocks for the corresponding kernel is called a *grid*. The maximum number of threads that can be contained by an individual block is defined by the GPU architecture.

CUDA uses a set of an application programming interface (API) functions to control the GPU. A CUDA program often follows (i) allocate space to the device memory, (ii) copy input data to the allocated device memory space, (iii) launch the program on the GPU, (iv) copy output data back to the host memory, and (v) free the allocated device memory space. The scope of this paper is related to (ii) and (iv). In particular, we focus on the `cuMemCopyHtoD()` and the `cuMemCopyDtoH()` functions of the CUDA Driver API, which correspond to (ii) and (iv) respectively. Since an open-source implementation of these functions is available with Gdev [12], we modify them to accomodate the data transfer methods investigated in this paper.

Our computing platform contains a single set of the CPU and the GPU. Although we restrict our attention to CUDA and the GPU, the notion of the investigated data transfer methods is well applicable to other heterogeneous compute devices. GPUs are currently well-recognized forms of the heterogeneous compute devices, but emerging alternatives include the Intel Many Integrated Core (MIC) and the AMD Fusion technology. The programming models of these different platforms are almost identical in that the CPU controls the compute devices. Our future work includes an integrated investigation of these different platforms.

III. Data Transfer Methods

In this section, we investigate data transfer methods for the GPU. A standard data transfer method uses hardware DMA engines integrated on the GPU, while direct data read and write accesses to the GPU device memory are allowed through PCIe BARs. We can also use microcontrollers integrated on the GPU to send and receive data across the host and the device memory. Unfortunately, only a limited piece of these schemes has been studied in the literature. Our investigation and open implementations of these schemes provide a better understanding of data transfer mechanisms

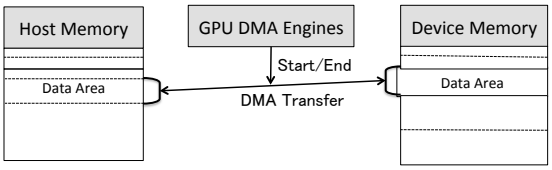


Figure 2. Standard DMA.

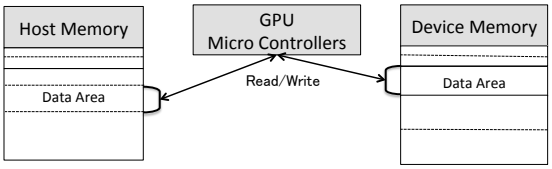


Figure 3. Microcontroller-based data transfer.

for the GPU. Note that we restrict our attention to the NVIDIA GPU architecture, but the concepts of data transfer methods introduced in this paper are mostly applicable to PCIe-connected compute devices.

A. Standard DMA

The most typical method for GPU data transfers is to use standard DMA engines integrated on the GPU. There are two types of such DMA engines for synchronous and asynchronous data transfer operations respectively. We focus on the synchronous DMA engines, which always operate in a sequential fashion with compute engines.

Figure 2 shows a concept of this standard DMA method. To perform this DMA, we write *GPU commands* to an on-board DMA engine. Upon a request of GPU commands, the DMA engine transfers a specified data set between the host and the device memory. Once a DMA transfer starts, it is non-preemptive. This method is often the most effective to transfer a large size of data. Details of the GPU DMA mechanism can be found in previous work [11], [12].

B. Microcontroller-based Data Transfer

The GPU provides on-board microcontrollers to control GPU functional units (compute, DMA, power, temperature, encode, decode, etc.). Albeit tiny hardware, these microcontrollers are available for GPU resource management beyond just controlling the functional units. Each microcontroller supports special instructions to transfer data in the data sections to and from the host or the device memory. The data transfer is offloaded to the microcontroller, *i.e.*, DMA, but is controlled by the microcontroller itself. Leveraging this mechanism, we can provide data communications between the host and the device memory.

Figure 3 shows a concept of this microcontroller-based data transfer method. Since there is no data path to directly copy data between the host and the device memory using a

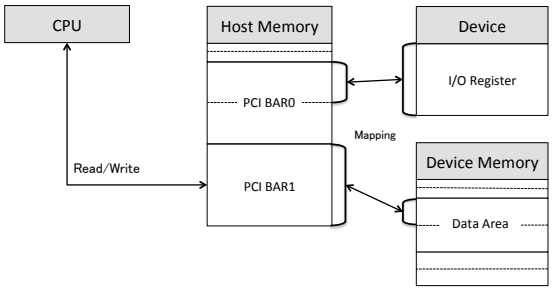


Figure 4. Memory-mapped read and write.

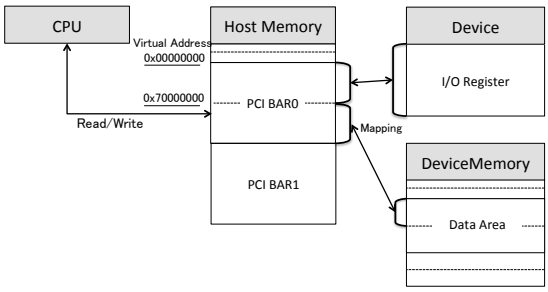


Figure 5. Memory-window read and write.

microcontroller, each data transfer needs to take two hops: (i) the host memory and a microcontroller and (ii) the device memory and a microcontroller. This is non-trivial overhead but the handling of this DMA is very light-weight as compared to the standard DMA method.

The microcontroller is executing firmware loaded by the device driver. We modify this firmware code to employ an interface for the data communications. The firmware adopts an event-driven design: it invokes only when the device driver sends some command from the CPU. The user program hence first needs to communicate with the device driver to issue such a command. Our implementation uses `ioctl` system calls to achieve this user and device driver communication.

C. Memory-mapped Read and Write

The aforementioned two methods are based on DMA functions. DMA is usually high-throughput but it inevitably incurs overhead in the setup. A small size of data transfers may encounter severe latency problems due to this overhead. One of good examples can be found in the plasma control system [8]. If low-latency is required, direct read and write accesses are more appropriate than hardware-based DMA. Since the GPU as a PCIe-connected device provides memory-mapped regions upon the PCI address space, the CPU can directly access the device memory without using bounce buffers on the host memory.

Figure 4 shows a concept of this memory-mapped read and write method. NVIDIA GPUs as well as most other PCIe devices expose base address registers (BARs) to the system, through which the CPU can access specific areas of the device memory. There are several BARs depending on the target device. NVIDIA GPUs typically provide the following BARs:

BAR0 Memory-mapped I/O (MMIO) registers.
BAR1 Device memory aperture (windows).
BAR2 I/O port or complementary space of BAR1.
BAR3 Same as BAR2.
BAR5 I/O port.
BAR6 PCI ROM.

Often the BAR0 is used to access the control registers of the GPU while the BAR1 makes the device memory visible to the CPU. This direct read and write method is pretty simple. We create virtual address space for the BAR1 region and set its leading address to a specific control register. Thus the BAR1 region can be directly accessed by the GPU using the unified memory addressing (UMA) mode, where all memory objects allocated to the host and the device memory can be referenced by the same address space. Once the BAR1 region is mapped, all we have to do is to manage the page table of the GPU to allocate memory objects from this BAR1 region and call the I/O remapping function supported by the OS kernel to remap the corresponding BAR1 region to the user-space buffer.

D. Memory-window Read and Write

The BAR0 region is often called memory-mapped I/O (MMIO) space. This is the main control space of the GPU, through which all hardware engines are controlled. Its space is sparsely populated with areas representing individual hardware engines, which in turn are sparsely populated with control registers. The list of hardware engines is architecture-dependent. The MMIO space contains a special subarea for indirect device and host memory accesses, seperated from the control registers. This plays a role of windows that make the device memory visible to the CPU in a different way than the BAR1 region.

Figure 5 shows a concept of this memory-window read and write method. To set the memory window, we obtain the leading physical address of the corresponding memory object and set it to a specific control register. By doing so, a limited range of the memory object becomes visible to the CPU through a specific BAR0 region. In case of NVIDIA GPUs, the size of this range is 4MB and the specific BAR0 region begins at 0x70000000 in the MMIO address. Once the window is set, we can read and write this BAR0 region to access data on the device memory.

E. Pinned Host Memory

Yet another approach to GPU data transfers is to use the pinned host memory. As aforementioned, NVIDIA GPUs support UMA. This is due to the graphics address remapping table (GART) employed by the GPU, allowing the system to specify physical host memory addresses directly in the

GPU page table as far as they are associated with pinned PCI-mapped pages. Although this approach is also effective to an extent for low-latency GPU computing [8], it is not within the scope of this paper and we focus on how to access the device memory in this paper.

IV. Empirical Comparison

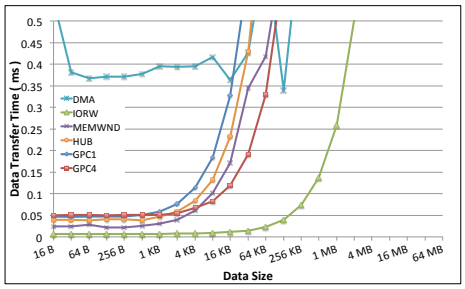
We now provide a detailed empirical comparison for the advantage and disadvantage of the data transfer methods presented in Section III. Our experimental setup is composed of an Intel Core i7 2600 processor and an NVIDIA GeForce GTX 480 graphics card. We use the Linux kernel v2.6.42 and Gdev [12] as the underlying OS and GPGPU runtime/driver software respectively. This set of open-source platforms allows our implementations of the investigated data transfer methods.

The test programs are written in CUDA [17] and are compiled using the NVIDIA CUDA Compiler (NVCC) v4.2 [15]. Note that Gdev is compatible with this binary compiler toolkit. We exclude compute kernels and focus on data transfer functions in this empirical comparison. While the test programs uniformly use the same CUDA API functions, we provide different internal implementations according to the target data transfer methods.

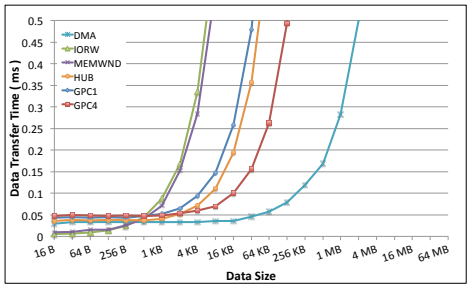
Data streams between the host and the device memory are provided by real-time tasks in Linux. These real-time tasks are prioritized over other background tasks running in Linux. The real-time capability relies on the default performance of the real-time scheduling class supported by the Linux kernel. We believe that this setup is sufficient for our experiments given that we execute at most one real-time task in the system while multiple data streams may be produced by this task. Scheduling performance issues are not within the scope of this paper.

Henchforth we use the following labels to denote the investigated data transfer methods respectively:

- **DMA** denotes the standard DMA method presented in Section III-A.
- **IORW** denotes the memory-mapped read and write method presented in Section III-C.
- **MEMWND** denotes the memory-window read and write method presented in Section III-D.
- **HUB** denotes the microcontroller-based data transfer method presented in Section III-B, particularly using a *hub* microcontroller designed to broadcast among the actual microcontrollers of graphics processing clusters (GPCs), *i.e.*, CUDA core clusters.
- **GPC** denotes the microcontroller-based data transfer method presented in Section III-B, particularly using a single GPC microcontroller.
- **GPC4** denotes the microcontroller-based data transfer method presented in Section III-B, particularly using four different GPC microcontrollers in parallel. Note that the NVIDIA Fermi architecture [14] provides four GPCs and their microcontrollers can perform invididually. Therefore we can split the data transfer into four pieces and make the four microcontrollers work in parallel.

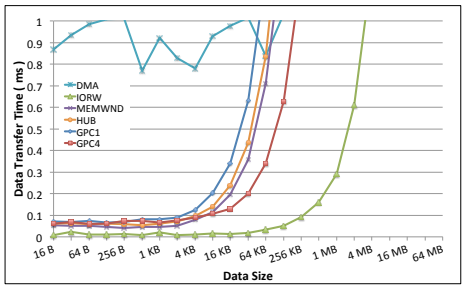


(a) Host to Device

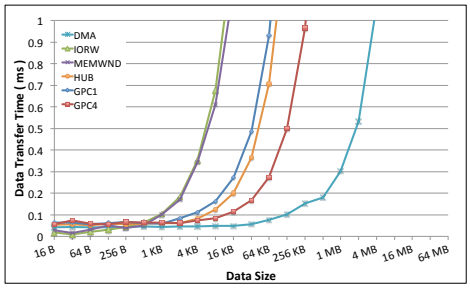


(b) Device to Host

Figure 6. Average performance of single streams.



(a) Host to Device



(b) Device to Host

Figure 7. Worst-case performance of single streams.

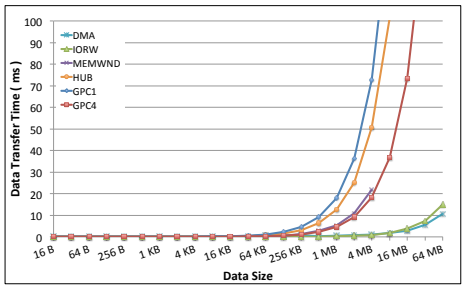
A. Basic Performance

Figure 6 shows the average performance of each data transfer method when a single data stream performs alone. Even with this most straightforward setup, there are several interesting observations. Performance characteristics of the host-to-device and the device-to-host communications are not identical at all. In particular, the performance of standard DMA exhibits a 10x difference between the two directions of communications. This is attributed to hardware capabilities that are not well documented to the public. We also find that performances of the methods are diverse but a combination of DMA and IORW can derive the best performance in this setup. The other methods are almost always inferior to either of DMA or IORW. The most signficant finding is that IORW becomes very slow for the device-to-host direction. We consider that this is due to a design specification of the GPU. It is designed so that the GPU can read data fast from the host computer but conpromise write access performance. Another interesting observation is that using multiple GPC microcontrollers to parallelize the data trasfer is less effective than a single GPC or HUB controller when the data size is small.

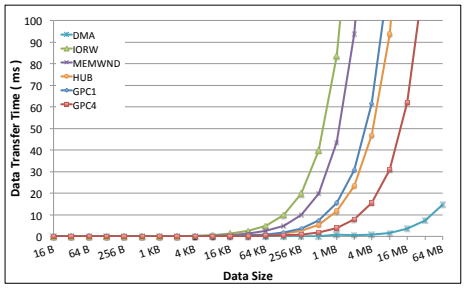
Figure 7 shows the worst-case performance in the same setup as the above experiment. It is important to note that we acquire almost the same results as those shown in Figure 6, though there is some degradation in the performance of DMA for the host-to-device direction. These comparisons lead to some conclusion that we may optimize the data transfer performance by switching between DMA and IORW at an appropriate boundary.

B. Interferred Performance

Figure 8 shows the average performance of each data transfer method when the CPU encounters extremely high workload.

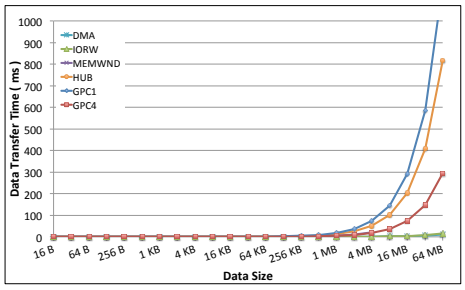


(a) Host to Device

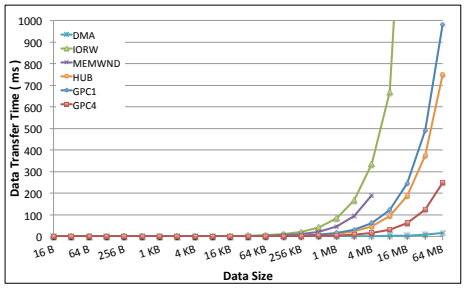


(b) Device to Host

Figure 8. Average performance of single streams under high CPU load.



(a) Host to Device



(b) Device to Host

Figure 9. Worst-case performance of single streams under high CPU load.

REFERENCES

[1] C. Basaran and K-D. Kang. Supporting Preemptive Task Executions and Memory Copies in GPGPUs. In *Proc. of the Euromicro Conference on Real-Time Systems*, pages 287–296, 2012.

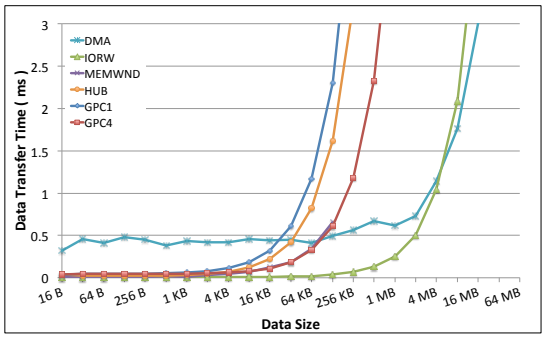
[2] P. Bhatotia, R. Rodrigues, and A. Verma. Shredder: GPU-accelerated incremental storage and computation. In *Proc. of the USENIX Conference on File and Storage Technologies*, 2012.

[3] G. Elliott and J. Anderson. Globally Scheduled Real-Time Multiprocessor Systems with GPUs. *Real-Time Systems*, 48(1):34–74, 2012.

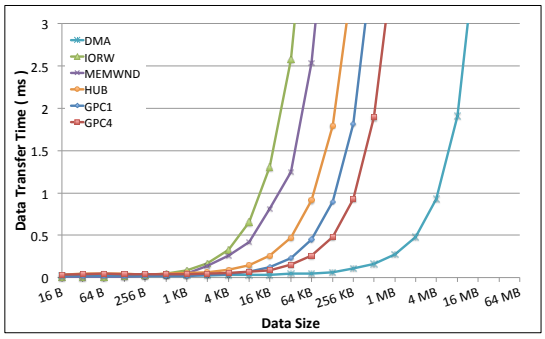
[4] G. Elliott and J. Anderson. Robust Real-Time Multiprocessor Interrupt Handling Motivated by GPUs. In *Proc. of the Euromicro Conference on Real-Time Systems*, pages 267–276, 2012.

[5] A. Gharaibeh, S. Al-Kiswany, S. Gopalakrishan, and M. Ripeanu. A GPU-accelerated storage system. In *Proc. of the ACM International Symposium on High Performance Distributed Computing*, pages 167–178, 2010.

[6] S. Hand, K. Jang, K. Park, and S. Moon. PacketShader: a GPU-accelerated software router. In *Proc. of ACM SIG-*

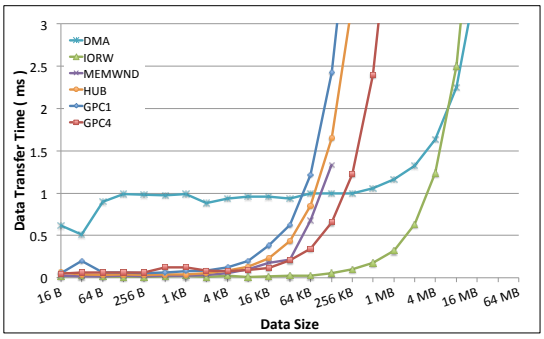


(a) Host to Device

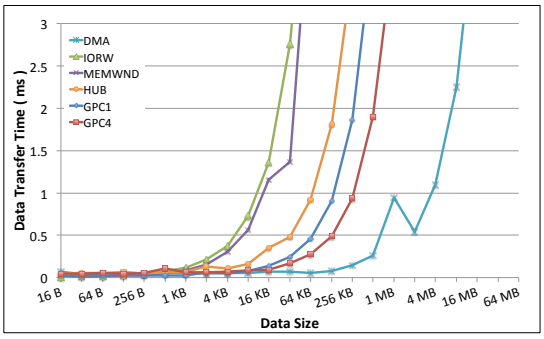


(b) Device to Host

Figure 10. Average data transfer times of real-time single streams under high memory access load.



(a) Host to Device



(b) Device to Host

Figure 11. Worst-case data transfer times of real-time single streams under high memory access load.

COMM, 2010.

[7] K. Jang, S. Han, S. Han, S. Moon, and K. Park. SSLShader: cheap SSL acceleration with commodity processors. In *Proc. of the USENIX Conference on Networked Systems Design and Implementation*, 2011.

[8] S. Kato, J. Aumiller, and S. Brandt. Zero-Copy I/O Processing for Low-Latency GPU Computing. In *Proc. of the IEEE/ACM International Conference on Cyber-Physical Systems*, 2013 (to appear).

[9] S. Kato, K. Lakshmanan, Y. Ishikawa, and R. Rajkumar. Resource Sharing in GPU-accelerated Windowing Systems. In *Proc. of the IEEE Real-Time and Embedded Technology and Aplications Symposium*, pages 191–200, 2011.

[10] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar. RGEM: A Responsive GPGPU Execution Model for Runtime Engines. In *Proc. of the IEEE Real-Time Systems Symposium*, pages 57–66, 2011.

[11] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. TimeGraph: GPU Scheduling for Real-Time Multi-Tasking Environments. In *Proc. of the USENIX Annual Technical Conference*, 2011.

[12] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt. Gdev: First-Class GPU Resource Management in the Operating System. In *Proc. of the USENIX Annual Technical Conference*, 2012.

[13] M. McNaughton, C. Urmson, J. Dolan, and J-W. Lee. Motion Planning for Autonomous Driving with a Conformal Spatiotemporal Lattice. In *Proc. of the IEE International Conference on Robotics and Automation*, pages 4889–4895, 2011.

[14] NVIDIA. NVIDIA's next generation CUDA computer architecture: Fermi. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, 2009.

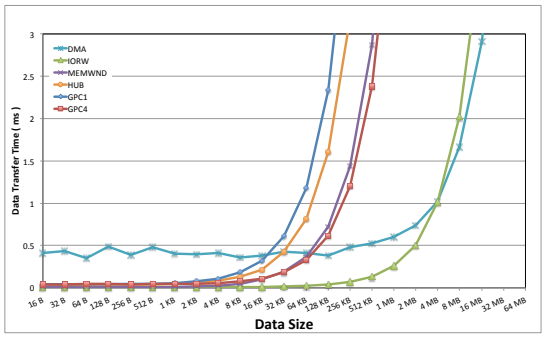
[15] NVIDIA. CUDA TOOLKIT 4.2. http://developer.nvidia.com/cuda/cuda-downloads, 2012.

[16] NVIDIA. NVIDIA's next generation CUDA computer architecture: Kepler GK110. http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf, 2012.

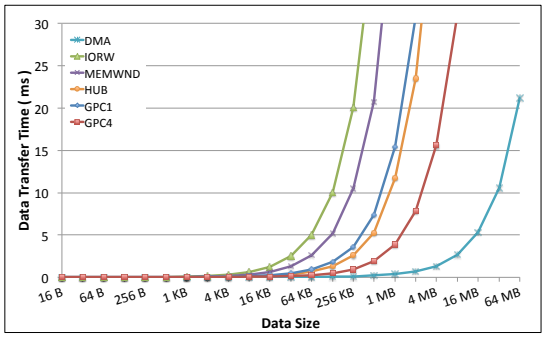
[17] NVIDIA. CUDA Documents. http://docs.nvidia.com/cuda/, 2013.

[18] W. Sun, R. Ricci, and M. Curry. GPUstore: harnessing GPU computing for storage systems in the OS kernel. In *Proc. of Annual International Systems and Storage Conference*, 2012.

[19] Top500 Supercomputing Sites. http://www.top500.org/.

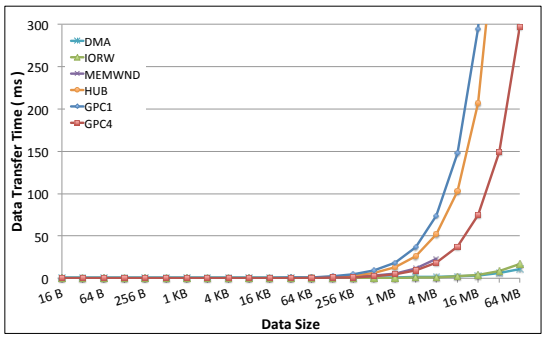


(a) Host to Device

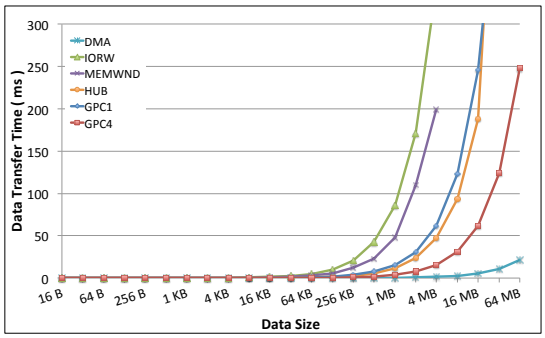


(b) Device to Host

Figure 12. Average data transfer times of real-time single streams in the presence of hackbench.

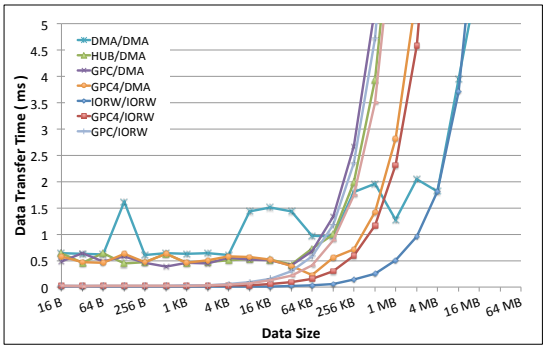


(a) Host to Device

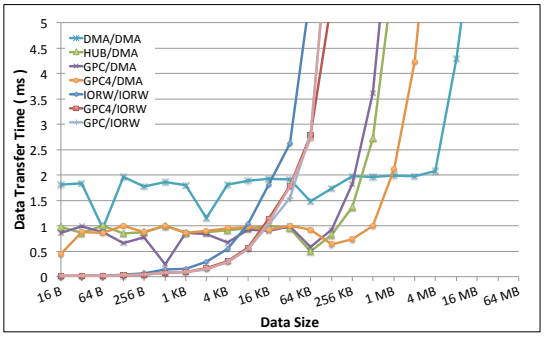


(b) Device to Host

Figure 13. Worst-case data transfer times of real-time single streams in the presence of hackbench.



(a) Host to Device



(b) Device to Host

Figure 14. Average data transfer times of concurrent two streams.