Data Transfer Matters for 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 Engineering
Nagoya University

***Abstract*—Graphics processing units (GPUs) embrace many-core compute devices where massively parallel compute threads are offloaded from CPUs. This heterogeneous nature of GPU computing raises non-trivial data transfer issues especially for low-latency real-time systems, but the fundamental characteristics of the data transfer are not well studied in the literature. In this paper, we investigate and characterize currently-achievable data transfer methods with state-of-the-art GPU technology. We also provide open-source implementations of these methods to compare their advantage and disadvantage in performance using real-world systems. Our experimental results show that the hardware-based direct memory access (DMA) and the I/O read-and-write methods are the most effective, while on-chip microcontrollers inside the GPU are useful in case of reducing the data transfer latency for concurrent multiple data streams. Our findings also include that CPU priorities can protect the performance of GPU data transfers in the context of real-time systems.**

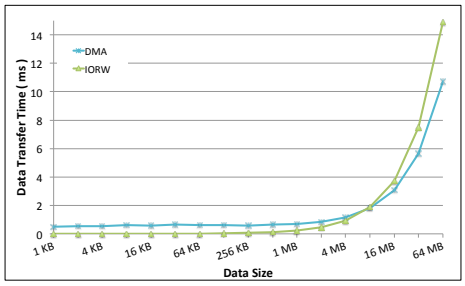
***Keywords*-GPUs; Data Transfer; Latency; Performance**

I. Introduction

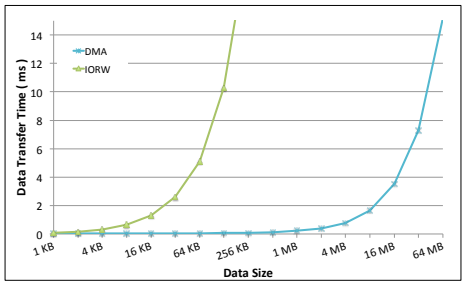
Graphics processing units (GPUs) are becoming powerful many-core compute devices. For example, NVIDIA GPUs integrate more than $1,500$ processing cores on a single chip and the peak double-precision performance exceeds 1 TFLOPS while sustaining thermal design power (TDP) in the same order of magnitude as traditional multicore CPUs [**?**]. 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 significant performance benefits from GPGPU. Currently a main application of GPGPU is supercomputing [**?**] but there are more and more emerging applications in different fields. Examples include plasma control [**?**], autonomous driving [**?**], software routing [**?**], encrypted networking [**?**], and storage management [**?**], [**?**], [**?**], [**?**]. This broad range of applications raises the need of further developing GPU technology to enhance scalability of emerging data-parallel and compute-intensive applications.

Grand challenges of GPU computing include an integration of real-time systems. So far the real-time systems community has addressed resource management issues for GPUs [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], but the main contribution of these work is limited to the scheduling of compute kernels and data transfers. The basic performance and latency issues for GPUs are not well studied in the 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 the constraint of PCIe devices [**?**]. Data transfers may also be interfered by competing workload on the CPU, while offloaded compute kernels are isolated on the GPU. These data transfer issues must be well understood and addressed to build predictable soft real-time systems, if not hard real-time systems, being integrated with cutting-edge GPU technology.



(a) Host to Device



(b) Device to Host

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

Data transfer issues are particularly important for low-latency GPU computing. Kato *et. al.* demonstrated that the data transfer is a dominant property of GPU-accelerated plasma control systems [**?**]. This is a specific application where the data must be transferred between sensor/actuator devices and the GPU at a high-rate, but is a good example presenting impact of data transfers for GPU computing. Since emerging applications augmented with GPUs may demand a similar performance requirement, a better under-

standing of the GPU data transfer mechanism is desired.

Figure **??** shows the average data transfer times of hardware-based direct memory access (DMA) and memory-mapped I/O read-and-write access, which are performed on an NVIDIA GeForce GTX 480 graphics card using the open-source Linux driver [**?**]. Apparently the performance characteristics of the data transfer are not identical between the host-to-device and device-to-host directions. In previous work, a very elementary issue of this performance difference has been discussed [**?**], 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 functions of proprietary software provided by GPU vendors are well optimized, because the hardware details of GPUs are not disclosed to the public. However real-time systems must build on a predictable basis. A better understanding of the data transfer is an essential piece of work to facilitate research on real-time GPU computing.

Contribution: This paper explores data transfer methods for GPU computing and provides their detailed empirical comparison. We unveil several data transfer methods other than the well-known DMA and I/O read-and-write access. The advantage and disadvantage of these methods are quantitatively clarified. The contribution of this paper provides an insight of how to design and implement data transfer methods especially in the context of real-time GPU computing.

Organization: The rest of this paper is organized as follows. Section **??** presents the assumption and terminology behind this paper. Section **??** provides an open investigation of data transfer methods for GPU computing. Section **??** compares the performances of the investigated data transfer methods in the context of real-time systems. Related work are discussed in Section **??**. We provide our concluding remarks in Section **??**.

II. Assumption and Terminology

We assume the Compute Unified Device Architecture (CUDA) for GPU programming [**?**]. 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

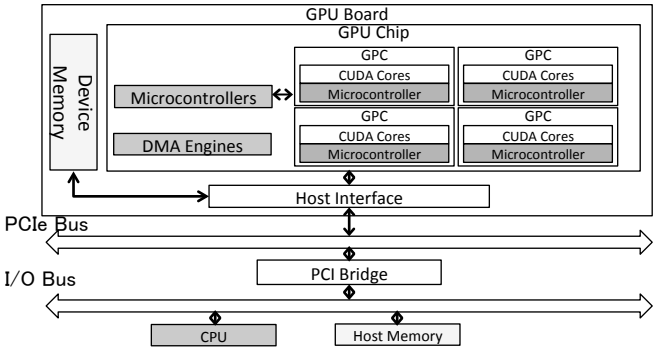


Figure 2. Block diagram of the target system.

memory space. The scope of this paper is related to (ii) and (iv). In particular, we use 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 [**?**], we modify them to accommodate the data transfer methods investigated in this paper.

In order to focus on the performance of data transfers between the host and the device memory, we allocate a data buffer to the pinned host memory rather than the typical heap allocated by `malloc()`. This pinned host memory space is mapped to the PCIe address and is never swapped out. It is also accessible to the GPU directly.

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.

Figure **??** shows a summarized block diagram of the target system. The host computer consists of the CPU and the host memory communicating on the system I/O bus. They are connected to the PCIe bus to which the GPU board is also connected. This means that the GPU is visible to the CPU as a PCIe device. The GPU is a complex compute device integrating a lot of hardware functional units on a chip. This paper is only focused on the CUDA-related units. There are the device memory and the GPU chip connected through a high bandwidth memory bus. The GPU chip contains graphics processing clusters (GPCs), each of which integrates hundreds of processing cores, *a.k.a*, CUDA cores. The number of GPCs and CUDA cores is architecture-specific. For example, GPUs based on the NVIDIA GeForce Fermi architecture [**?**] used in this paper suppor at most 4

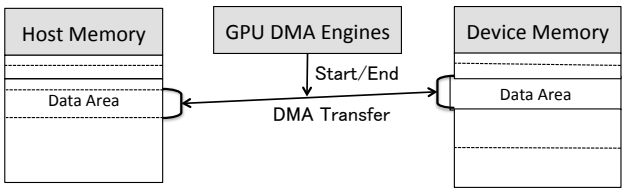


Figure 3. Standard DMA.

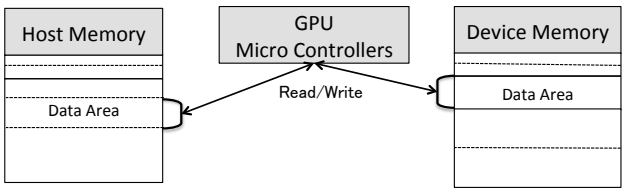


Figure 4. Microcontroller-based data transfer.

GPCs and 512 CUDA cores. Each GPC is configured by an on-chip microcontroller. This microcontroller is wimpy but is capable of executing firmware code with its own instruction set. There is also a special *hub* microcontroller, which broadcasts the operations on all the GPC-dedicated microcontrollers. In addition to hardware DMA engines, this paper investigates how these detailed hardware components operate and interact with each other to support data transfers in GPU computing.

III. Data Transfer Methods

In this section, we investigate data transfer methods for GPU computing. The most intuitive data transfer method uses standard hardware DMA engines on the GPU, while direct read and write accesses to the device memory of the GPU are allowed through PCIe BARs. We may 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 for the GPU. Note that we restrict our attention to the NVIDIA GPU architecture, but the concept of data transfer methods introduced in this paper is mostly applicable to any 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 **??** 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. To wait for the completion of DMA, the system can either poll a specific GPU register or generate a GPU interrupt. This method is often the most effective to transfer a large size of data. The details of this hardware-based DMA mechanism can be found in previous work [**?**], [**?**].

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 **??** 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 microcontroller, each data transfer is forced to take two hops: (i) the host memory and microcontroller and (ii) the device memory and 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 executes firmware loaded by the device driver. We modify this firmware code to employ an interface for the data communications. This firmware adopts an event-driven design. The firmware task invokes only when the device driver sends a corresponding command from the CPU through the PCIe bus. The user program first needs to communicate with the device driver to issue this command to the microcontroller. Our implementation uses `ioctl` system calls to achieve this user and device driver communication. The firmware command can be issued by poking a specific MMIO register. We have also developed an open-source C compiler for this GPU microcontroller. It may be downloaded from http://github.com/cs005/guc.

A constraint of this microcontroller approach is that the size of each data transfer is limited by 256 bytes. If the data transfer size exceeds 256 bytes, we have to split a transaction into multiple chunks. Although this could be additional overhead to the data transfer time, we can also use multiple microcontrollers to send these separate chunks in parallel. This parallel transaction can improve the makespan of the total data transfer time.

C. Memory-mapped Read and Write

The aforementioned two methods are based on DMA functions. DMA is usually high-throughput but it inevitably

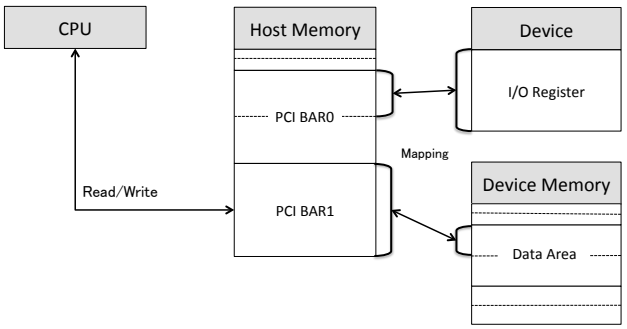


Figure 5. Memory-mapped read and write.

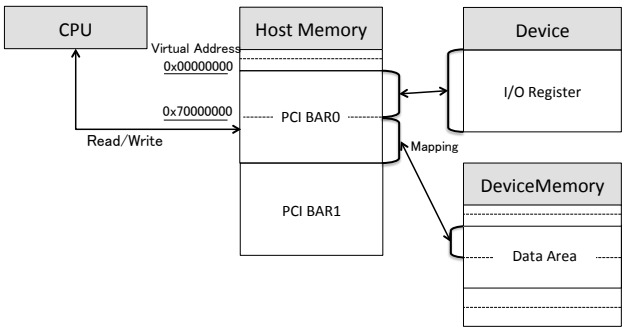


Figure 6. Memory-window read and write.

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 [**?**]. If low-latency is required, this direct I/O read and write method is more appropriate than the hardware-based DMA method. 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 **??** 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, separated 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 **??** 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 [**?**], it is outside the scope of this paper and we focus on how to access the device memory in this paper. Note that we still use the host memory as buffer space; we do not consider a method that makes the GPU read from and write to this host memory directly through UMA.

IV. Empirical Comparison

We now provide a detailed empirical comparison for the advantage and disadvantage of the data transfer methods presented in Section **??**. 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 [**?**] 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 [**?**] and are compiled using the NVIDIA CUDA Compiler (NVCC) v4.2 [**?**]. 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.

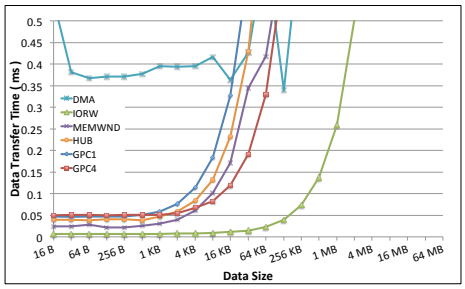
Data streams between the host and the device memory are provided by a single GPU context. Performance interference among multiple GPU contexts is outside the scope of this paper. We evaluate the data transfer performances of both real-time and normal tasks. For the scheduling policies of the Linux kernel, we use `SCHED_FIFO` for real-time tasks while `SCHED_OTHER` for normal tasks, where the real-time tasks are always prioritized over the normal tasks. 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. Overall the scheduling performance issues are outside the scope of this paper.

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

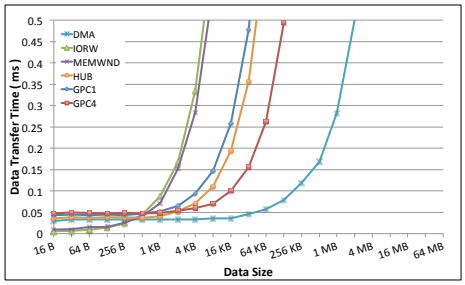
- **DMA** denotes the standard DMA method presented in Section **??**.
- **IORW** denotes the memory-mapped read and write method presented in Section **??**.
- **MEMWND** denotes the memory-window read and write method presented in Section **??**.
- **HUB** denotes the microcontroller-based data transfer method presented in Section **??**, 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 **??**, particularly using a single GPC microcontroller.
- **GPC4** denotes the microcontroller-based data transfer method presented in Section **??**, particularly using four different GPC microcontrollers in parallel. Note that the NVIDIA Fermi architecture [**?**] provides four GPCs and their microcontrollers can perform individually. Therefore we can split the data transfer into four pieces and make the four microcontrollers work in parallel.

A. Basic Performance

Figure **??** shows the average performance of each data transfer method when a real-time task runs 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

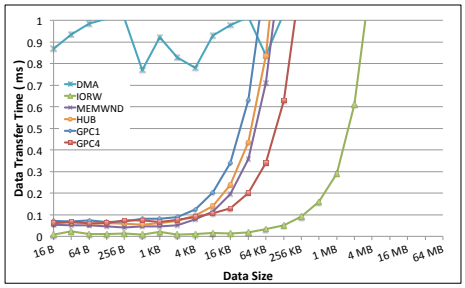


(a) Host to Device

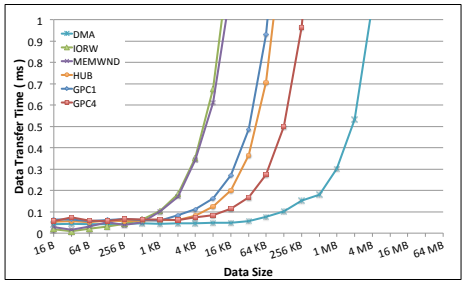


(b) Device to Host

Figure 7. Average performance of each data transfer method with a real-time task.



(a) Host to Device



(b) Device to Host

Figure 8. Worst-case performance of each data transfer method with a real-time task.

DMA exhibits a 10x difference between the two directions of communications. We believe that this is due to hardware capabilities that are not documented to the public. We also find that the performances of different methods are diverse but either DMA or IORW can derive the best performance for any data size. In case of the host-to-device direction, for small data from 16B to 8MB, IORW is preferred, while DMA outperforms IORW larger data than 8MB. The other methods are almost always inferior to either of DMA or IORW. The most significant finding is that IORW becomes very slow for the device-to-host direction. It is faster than DMA only until 512B. 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 compromise write access performance. Another interesting observation is that using multiple GPC microcontrollers to parallelize the data transfer is less effective than a single GPC or HUB controller when the data size is small.

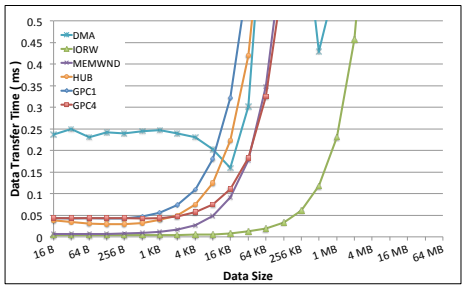
Figure **??** shows the worst-case performance in the same setup as the above experiment. It is important to note that we acquire almost the same results as those shown in Figure **??**, though there is some degradation in the performance of DMA for the host-to-device direction. These comparisons lead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

We omit the results of normal tasks in this setup, because they are almost equal to those of real-time tasks shown above. However, real-time and normal tasks behave in a very different manner in the presence of competing workload. This will be discussed in the next subsection.

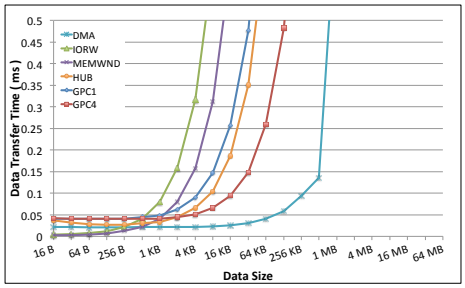
B. Interfered Performance

Figure **??** shows the average performance of each data transfer method when a real-time task encounters extremely high workload on the CPU. All the methods are successfully protected from performance interference due to the real-time scheduler. Note that DMA shows a better performance than the previous experiments despite the presence of competing workload. This is due to the Linux real-time scheduler feature. DMA is performed by GPU commands, which may impose a suspension on the caller task. In the Linux kernel, a real-time task is awakened in a more responsive manner when switched from a normal task than from an idle task. Therefore when the CPU is fully loaded by normal tasks, a real-time task is more responsive. The same is true for the worst-case performance as shown in Figure **??**. We learn from these experiments that CPU priorities can protect the performance of data transfer for the GPU. Note that Gdev uses a polling approach to wait for completions of data transfers. An interrupt approach is also worth being investigated.

For reference, Figure **??** and **??** show the average and the worst-case performance achieved by a normal task when

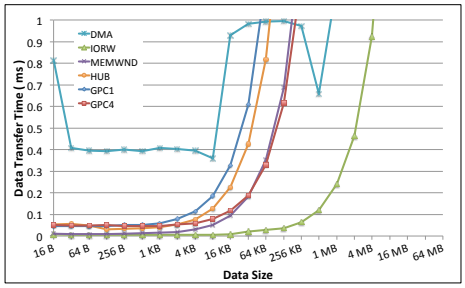


(a) Host to Device

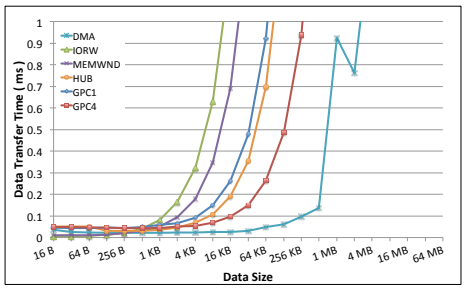


(b) Device to Host

Figure 9. Average performance of each data transfer method with a real-time task under high CPU load.



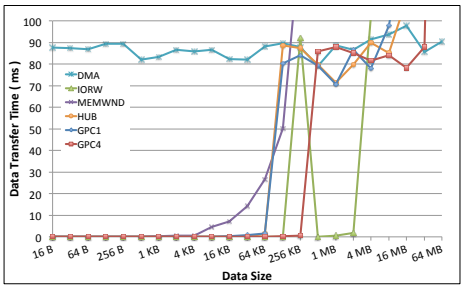
(a) Host to Device



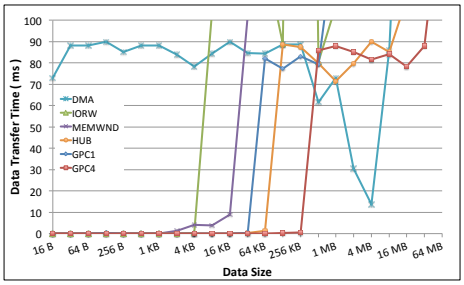
(b) Device to Host

Figure 10. Worst-case performance of each data transfer methods with a real-time task under high CPU load.

the CPU encounters extremely high workload same as the preceding experiments. Apparently the data transfer times increase by orders of magnitude as compared to those achieved by a real-time task. DMA shows a bad performance

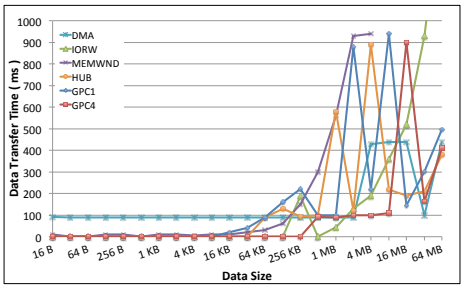


(a) Host to Device

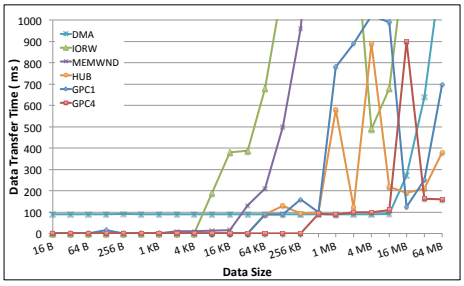


(b) Device to Host

Figure 11. Average performance of each data transfer method with a normal data stream under high CPU load.

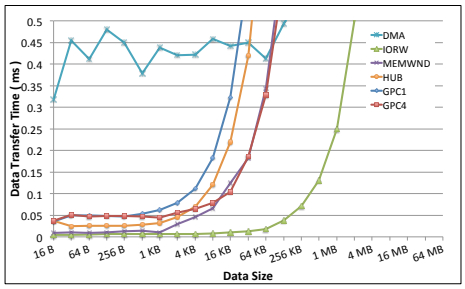


(a) Host to Device

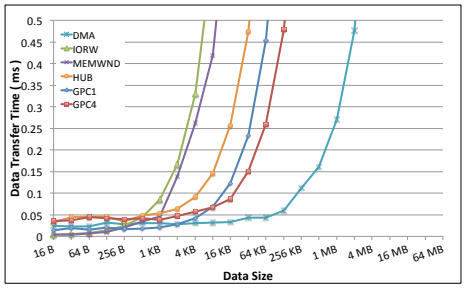


(b) Device to Host

Figure 12. Worst-case performance of each data transfer method with a normal data stream under high CPU load.

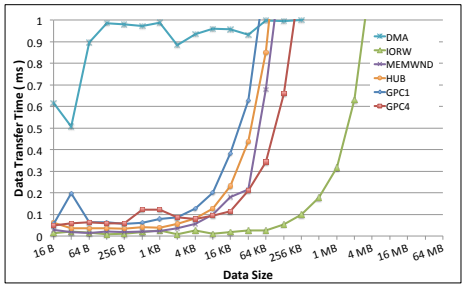


(a) Host to Device

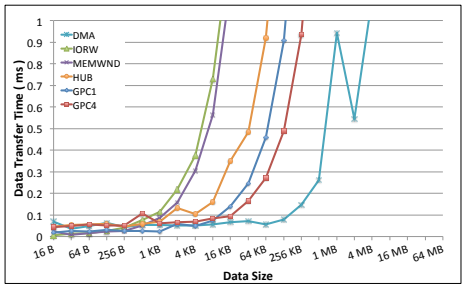


(b) Device to Host

Figure 13. Average performance of each data transfer method with a real-time task under high memory pressure.



(a) Host to Device



(b) Device to Host

Figure 14. Worst-case performance of each data transfer method with a real-time task under high memory pressure.

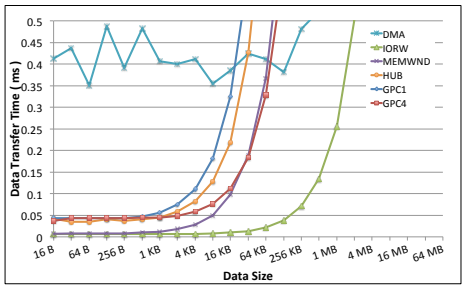
for small data while it can sustain that performance for large data. This is attributed to the fact that once a DMA command is fired, the data transfer does not have to compete with CPU workload. The other methods are more or less controlled by the CPU, and thus are more affected by CPU workload. This finding provides a design choice for the system implementation such that the hardware-based DMA method is preferred in fair-scheduled systems.

From now on, we restrict our attention to a real-time task. We next evaluate the performance of each data transfer method under high memory pressure, creating another task that eats up host memory space. In some sense, this is not a meaningful experiment because we use the pinned host memory space to allocate buffers while the memory pressure is supposed to compel the paged host memory space. Having said that, the memory pressure could still impose indirect interference on real-time tasks [**?**], [**?**]. It is worth conducting this experiment. As demonstrated in Figure **??** and **??**, the impact of memory pressure on the data transfer performance is negligible. This means that all the data transfer methods investigated in this paper require not much paged host memory space. Otherwise they must be interfered by memory workload.

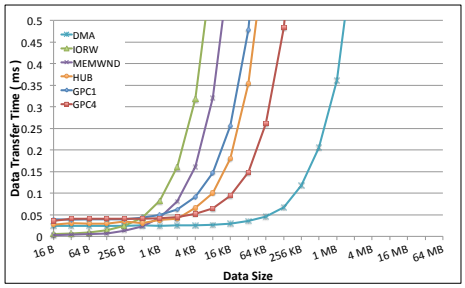
Figure **??** and **??** show the average and the worst-case performance of each data transfer method respectively, when a misbehaving hackbench process coexists. hackbench is a tool that generates many processes executing I/O system calls with pipes. The results are all similar to the previous ones. Since the Linux real-time scheduler is now well enhanced to protect a real-time task from such misbehaving workload, these results are obvious and trivial in some sense, but we can lead to a conclusion from a series of the above experiments that *the data transfer methods for the GPU can be protected by the traditional real-time scheduler capability*. This is a useful finding to facilitate an integration of real-time systems and GPU computing.

C. Concurrent Performance

So far we have studied the capabilities of the investigated data transfer methods and their causal relation to a real-time task. We now evaluate the performance of concurrent *two* data streams using different combinations of the data transfer methods as shown in Figure **??** and **??**. This is a very interesting result. For the host-to-device direction, the best performance is obtained when both the two tasks use IORW. In this case, the two data streams are not overlapped but are processed in sequential due to the use of the same IORW path. Nonetheless it outperforms the other combinations because the performance of IORW is way higher than the other methods as we have observed in a series of the previous experiments. However, the device-to-host direction shows a different performance. Since IORW becomes slow when the CPU reads the device memory as mentioned in Section **??**, IORW/IORW is not the best performer any longer. Instead using the microcontroller(s) provides the best performance until 2MB. This is attributed to the fact that the microcontroller-based data transfer method can be overlapped with any other data transfer methods. From 16B to 16KB, a combination of the microcontroller and IORW is the fastest, while that of the microcontroller and DMA is the fastest from 16KB to 2MB. Note that from 16B to 16KB the performance is aligned with the slow IORW curve. Therefore the choice of HUB, GPC, and GPC4 does not really matter to the performance. However, from 16KB to 2MB the performance is improved by using four microcontrollers in parallel (*i.e.*, GPC4), since DMA is

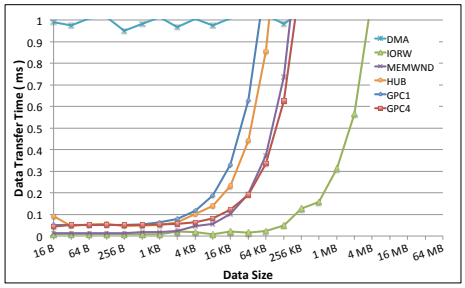


(a) Host to Device

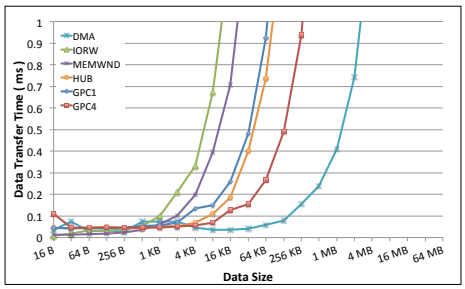


(b) Device to Host

Figure 15. Average performance of each data transfer method with a real-time task in the presence of hackbench.

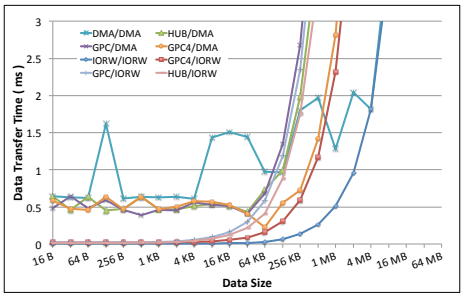


(a) Host to Device

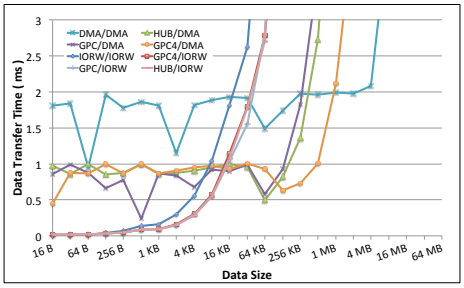


(b) Device to Host

Figure 16. Worst-case performance of each data transfer method with a real-time task in the presence of hackbench.

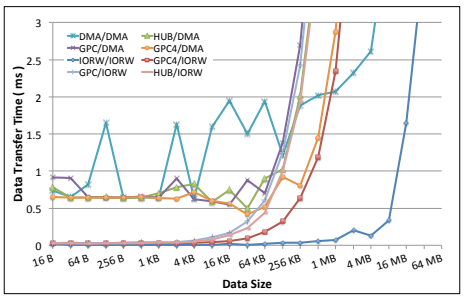


(a) Host to Device

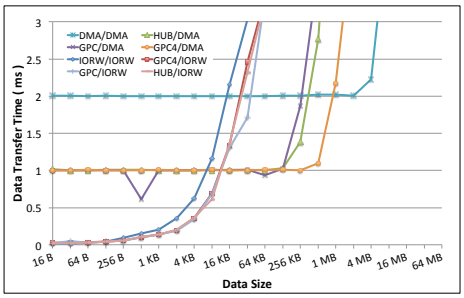


(b) Device to Host

Figure 17. Average performance of combinations of the data transfer methods for concurrent real-time tasks.



(a) Host to Device



(b) Device to Host

Figure 18. Worst-case performance of combinations of the data transfer methods for concurrent real-time tasks.

faster than the microcontroller and thereby the performance is aligned with the microcontroller curve. We learn from this experiment that the microcontroller is useful to overlap concurrent data streams with DMA or IORW, and using multiple microcontrollers in parallel can further improve the performance of concurrent data transfers.

V. Related Work

The zero-copy data transfer methods for low-latency GPU computing were developed for plasma control systems [**?**]. The authors argued that the hardware-based DMA transfer method does not meet the latency requirement of plasma control systems. They presented several zero-copy data transfer methods, some of which is similar to the memory-mapped I/O read and write method investigated in this paper. However, this previous work considered only a small size of data. This specific assumption allowed the I/O read and write method to perform always better than the hardware-based DMA method. We demonstrated that these two methods outperform each other depending on the target data size. In this regard, we provided more general observations of data transfer methods for GPU computing.

The performance boundary of the hardware-based DMA and the I/O read and write methods was briefly discussed in the Gdev project [**?**]. They showed that the hardware-based DMA method should be used only for large data. We provided the same claim in this paper. However, we dig into the causal relation of these two methods more in depth and also expanded our attention to the microcontroller-based method. Our findings complement the results of the Gdev project.

The scheduling of GPU data transfers was presented to improve the responsiveness of GPU computing [**?**], [**?**]. These work focused on making preemption points for burst non-preemptive DMA transfers with the GPU, but the underlying system relied on the proprietary closed-source software. On the other hand, we provided open-source implementations to disclose the fundamental of GPU data transfer methods. We found that the hardware-based DMA transfer method is not necessarily the best choice depending on the data size and workload. Since the I/O read and write method is fully preemptive and the microcontroller-based method is partly preemptive, the contribution of this paper provides a new insight to these preemptive data transfer approaches.

VI. Conclusion

In this paper, we have presented an investigation and empirical comparison of the data transfer methods for GPU computing. We found that the hardware-based DMA and the I/O read and write methods are the most effective to maximize the data transfer performance for a single stream even in the presence of competing CPU workload. On the other hand, the microcontroller-based method is useful to overlap the data transfer with the DMA or the IO read and write methods, reducing the total makespan of multiple data streams. We also demonstrated that the traditional real-time

CPU scheduler can shield the data transfer of a real-time task from performance interference. We believe that all these findings are useful contributions to develop an integration of real-time systems and GPU computing.

Our open-source implementations of the data transfer methods provided in this paper may be downloaded from http://github.com/shinpei0208/gdev/.

In future work, we will investigate how to optimize the choice of data transfer methods depending on the target system and workload. Since user programs use the same API function for the data transfer, the underlying system software must understand environments and choose appropriate data transfer methods to maximize the performance and/or minimize the latency. Runtime management of soft real-time capabilities of GPU computing is also a core challenge for future work.