Data Transfer Matters for GPU Computing

Conference Paper · December 2013

DOI: 10.1109/ICPADS.2013.47

CITATIONS
39

READS
584

5 authors, including:

Takuya Azumi
Saitama University
68 PUBLICATIONS **354** CITATIONS

SEE PROFILE

Nobuhiko Nishio
Ritsumeikan University
78 PUBLICATIONS **390** CITATIONS

SEE PROFILE

Data Transfer Matters for GPU Computing

Yusuke Fujii*, Takuya Azumi†, Nobuhiko Nishio†, Shinpei Kato‡ and Masato Edahiro‡
*Graduate School of Information Science and Engineering, Ritsumeikan University
†College of Information Science and Engineering, Ritsumeikan University
‡School of Information Science, 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 problems especially against latency-critical real-time systems. However even the basic characteristics of data transfers associated with GPU computing are not well studied in the literature. In this paper, we investigate and characterize currently-achievable data transfer methods of cutting-edge GPU technology. We implement these methods using open-source software to compare their performance and latency for real-world systems. Our experimental results show that the hardware-assisted direct memory access (DMA) and the I/O read-and-write access methods are usually the most effective, while on-chip microcontrollers inside the GPU are useful in terms of reducing the data transfer latency for concurrent multiple data streams. We also disclose that CPU priorities can protect the performance of GPU data transfers.**

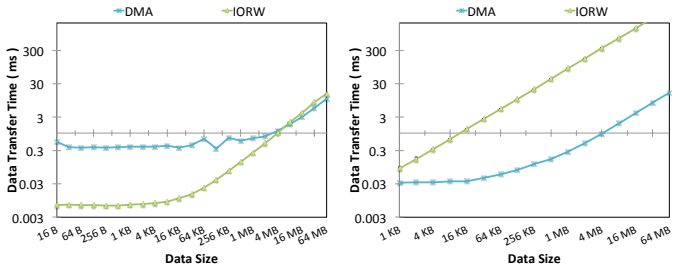
***Keywords*-GPGPU; Data Transfer; Latency; Performance; OS**

I. Introduction

Graphics processing units (GPUs) are becoming more and more commonplace as many-core compute devices. For example, NVIDIA GPUs integrate thousands of 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 multi-core CPUs [1]. 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 [2] but there are more and more emerging applications in different fields. Examples include plasma control [3], autonomous driving [4], software routing [5], encrypted networking [6], and storage management [7]–[10]. This broad range of applications raises the need of further developing GPU technology to enhance scalability of emerging data-parallel and compute-intensive applications.

GPU programming inevitably incurs data transfers between the host and the device memory. This resulting latency could be a performance stopper of I/O bound GPGPU applications. In fact, 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 [9]. 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 low-latency GPU computing.



(a) Host to Device (b) Device to Host

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

The data transfer is particularly an important issue for low-latency GPU computing. Kato *et. al.* demonstrated that the data transfer is a dominant property of GPU-accelerated plasma control systems [3]. 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 the impact of data transfers on GPU computing. Since emerging applications augmented with GPUs may demand a similar performance requirement, a better understanding of the GPU data transfer mechanism is desired.

Figure 1 depicts the average data transfer times of hardware-based direct memory access (DMA) and memory-mapped I/O read-and-write access, which are obtained on an NVIDIA GeForce GTX 480 graphics card using the open-source Linux driver [9]. Apparently the performance characteristics of the data transfer are not identical for the host-to-device and device-to-host directions. In previous work, a very elementary issue of this performance difference has been discussed [9], but there is no clear conclusion on what methods can optimize the data transfer performance, what different methods are available.

Currently we pray that the black-box data transfer mechanism of proprietary software, provided by GPU vendors, is well optimized to meet the performance that programmers expect, because hardware details of GPUs are not disclosed to the public. In order to achieve low-latency GPU computing, we must understand what latency and performance interference exist when using the GPU.

To some extent, GPUs are suitable for real-time computing once workload is offloaded, but host-device data transfers

may be affected by some competing workload on the host computer. Hence a better understanding of data transfers associated with GPU computing is an essential piece of work to support latency-critical real-time systems. Unfortunately prior work [3], [9] did not provide performance characterization in the context of latency and concurrent workload; they focused on a basic comparison of DMA and direct I/O access.

Contribution: In this paper, we clarify the performance characteristics of currently-achievable data transfer methods for GPU computing while unveiling several new data transfer methods other than the well-known DMA and I/O read-and-write access. We reveal the advantage and disadvantage of these methods in a quantitative way leading a conclusion that the typical DMA and I/O read-and-write methods are the most effective in latency even in the presence of compelling workload, whereas concurrent data streams from multiple different contexts can benefit from the capability of on-chip microcontrollers integrated in the GPU. To the best of our knowledge, this is the first evidence of data transfer matters for GPU computing beyond an intuitive expectation, which allows system designers to choose appropriate data transfer methods depending on the requirement of their latency-sensitive GPU applications. Without our findings, none can reason about the usage of GPUs minimizing the data transfer latency and performance interference. These findings are also applicable for many PCIe compute devices rather than a specific GPU. We believe that the contributions of this paper are useful for low-latency GPU computing.

Organization: The rest of this paper is organized as follows. Section II presents the assumption and terminology behind this paper. Section III provides an open investigation of data transfer methods for GPU computing. Section IV compares the performances of the investigated data transfer methods. Related work are discussed in Section V. We provide our concluding remarks in Section VI.

II. Assumption and Terminology

We assume that the Compute Unified Device Architecture (CUDA) is used for GPU programming [11]. 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.

CUDA uses a set of an application programming interface (API) functions to manage the GPU. A typical CUDA program takes the following steps: (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). Particularly we use the `cuMemCopyHtoD()` and the `cuMemCopyDtoH()` functions provided by the CUDA Driver API, which correspond to (ii) and (iv) respectively. Since an open-source implementation of these functions is available [9], we modify them to accommodate various data transfer methods investigated in this paper. While they are synchronous data transfer functions, CUDA also provides asynchronous data transfer functions. In this paper, we restrict our attention to the synchronous data transfer functions for simplicity of description, but partly similar performance characteristics can also be applied for the asynchronous ones. This is because both techniques are using the same data transfer method. The only difference is synchronization timing.

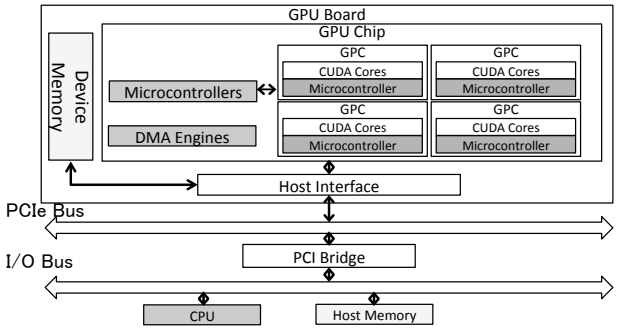


Fig. 2. Block diagram of the target system.

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 2 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 [12] used in this paper support at most 4 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

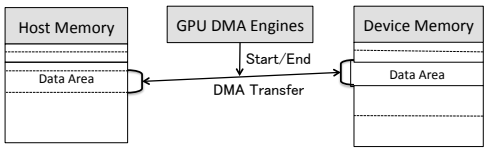


Fig. 3. Standard DMA.

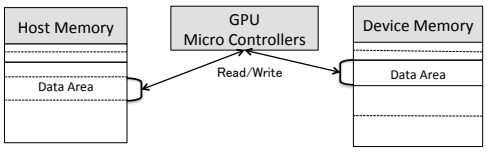


Fig. 4. Microcontroller-based data transfer.

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 base address registers (BARs). NVIDIA GPUs as well as most other PCIe devices expose 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 six to seven BARs. Often the BAR0 is used to access the control registers of the GPU while the BAR1 makes the device memory visible to the CPU.

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 applicable to any PCIe-connected compute devices.

A. Standard DMA (*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 3 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 [9], [13].

B. Microcontroller-based Data Transfer (*HUB,GPC,GPC4*)

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 4 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. 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.

There are three; **HUB**, **GPC** and **GPC4**. **HUB** is used a *hub* microcontroller designed to broad cast among the actual microcontrollers of graphics processing clusters (GPCs), *i.e.*, CUDA core clusters. **GPC** is used a single *GPC* microcontroller. **GPC4** is used four different *GPC* microcontrollers in parallel. We can split the data transfer into four pieces and make the four microcontrollers work in parallel.

C. Memory-mapped Read and Write (*IORW*)

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 [3].

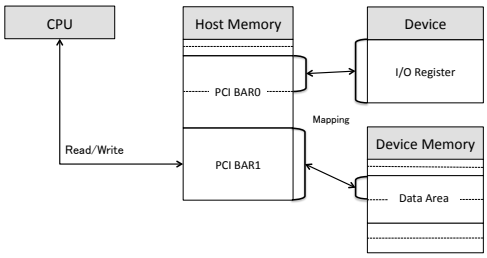


Fig. 5. Memory-mapped read and write.

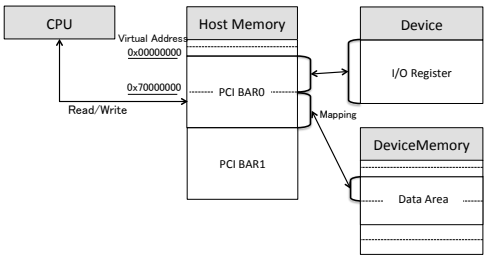


Fig. 6. Memory-window read and write.

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 5 shows a concept of this memory-mapped read and write method. 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 (**MEMWND**)*

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 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 6 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 Once the window is set, we can read and write this BAR0 region to access data on the device memory.

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 vanilla Linux kernel v2.6.42 and Gdev [9] 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.

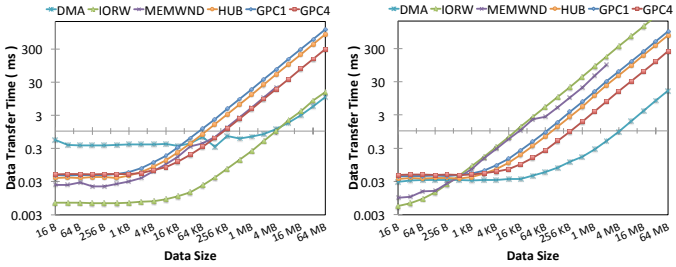
The test programs are written in CUDA [11] and are compiled using the NVIDIA CUDA Compiler (NVCC) v4.2 [14]. Note that Gdev is compatible with this binary compiler toolkit. We exclude compute kernels and focus on data transfer functions in this empirical comparison. While the test programs uniformly use the same CUDA API functions, we provide different internal implementations according to the target data transfer methods.

We prepared two test programs. One is Gdev memcpy benchmark[1] to measure *Basic Performance* and *Interfered Performance*. The other which is an extended version of the first one calls to a data transfer API two times, we used the second program to measure *Concurrent Performance*.

The scope of measurement is limited to the corresponding data size from 16B to 64MB, since data transfers with the data size greater than 64MB exibit similar performance characteristics for each method. Time stamps for the measurement are recorded within a test program itself, including interfered intervals during user-space memory copies. We present the average data transfer time obtained from 1000 times measurements for each data size and each method. Each data stream between the host and the device memory is provided by a single GPU context. The performance interference among multiple GPU contexts is considered here. We evaluate the data transfer performance for both real-time and normal tasks. In particular, we use the `SCHED_FIFO` scheduling policy for real-time tasks while normal tasks are scheduled by the default policy. 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 observe whether it is possible to reduce the competing noise by utilizing real-time tasks. 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.

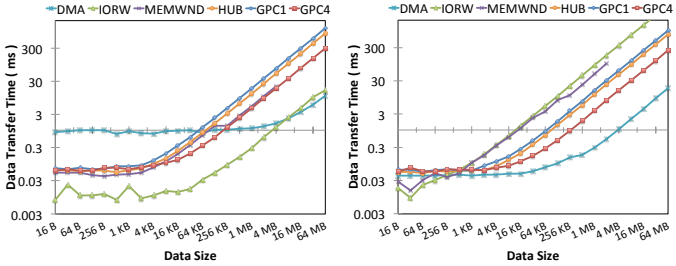
Henceforth we use the labels that explained in Section III, to denote the investigated data transfer method respectively.

[1] https://github.com/shinpei0208/gdev/tree/master/test/cuda/user/memcpy



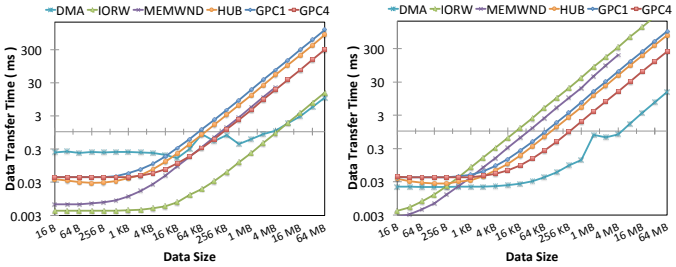
(a) Host to Device (b) Device to Host

Fig. 7. The average performance of the investigated data transfer methods using a real-time task.



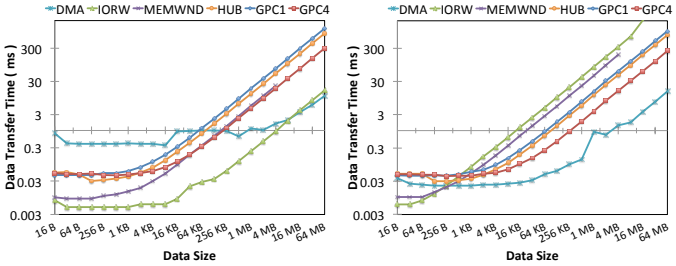
(a) Host to Device (b) Device to Host

Fig. 8. The worst-case performance of the investigated data transfer methods using a real-time task.



(a) Host to Device (b) Device to Host

Fig. 9. The average performance of the investigated data transfer methods using a real-time task under high CPU load.



(a) Host to Device (b) Device to Host

Fig. 10. The worst-case performance of the investigated data transfer methods using a real-time task under high CPU load.

A. Basic Performance

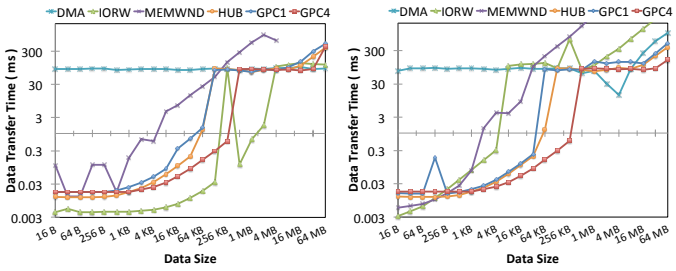
Figure 7 shows the average performance of the investigated data transfer methods when using a real-time task alone. Even with this most straightforward setup, there are several interesting observations. Performance characteristics of the host-to-device and the device-to-host communications are not identical at all. In particular the performance of standard DMA exhibits a 10x difference between the two directions of communications. 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. As shown in Figure 1 earlier, the small data ranging from 16B to 8MB prefers IORW for the host-to-device direction, while DMA outperforms IORW for larger data than 8MB. Figure 7 shows that 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. 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. This is attributed to the fact that the runtime management of multiple microcontrollers incurs additional overhead as compared to a single microcontroller, which does not pay for transferring small data sets.

Figure 8 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 7, 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 be able to optimize the data transfer performance by switching between DMA and IORW at an appropriate boundary. This boundary value however may depend on the system and device.

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.

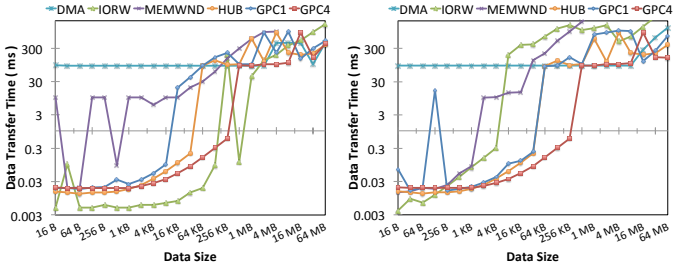
B. Interfered Performance

Figure 9 shows the average performance of the investigated data transfer methods 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. One may observe 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 launched by GPU commands, which could 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 10. We learn from these experiments that CPU priorities



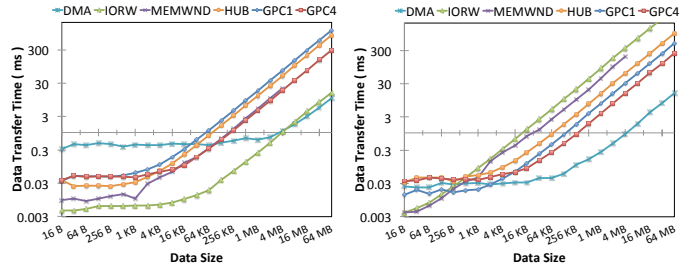
(a) Host to Device (b) Device to Host

Fig. 11. The average performance of the investigated data transfer methods using a normal task under high CPU load.



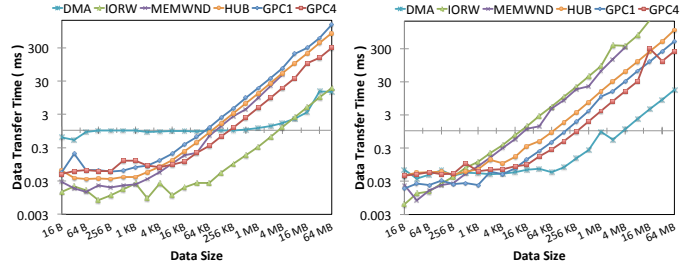
(a) Host to Device (b) Device to Host

Fig. 12. The worst-case performance of the investigated data transfer methods using a normal task under high CPU load.



(a) Host to Device (b) Device to Host

Fig. 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

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

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 11 and 12 show the average and the worst-case performance achieved by a normal task when 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. We guess spikes factor is affected by long polling wait time which is caused by high CPU load.

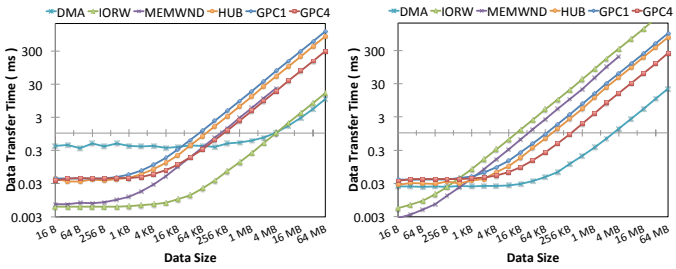
DMA shows a low performance for small data sets while it can be sustained for large data sets. This is attributed to the fact that once a DMA command is fired, the data transfer does not have to compete with CPU workload. The other methods are more or less controlled by the CPU, and thus are more affected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

We evaluate the performance of each data transfer method under high memory pressure, creating another task that eats up host memory space. Although we use pinned host memory space to allocate buffers while the memory pressure is supposed to compel the paged host memory space, it could still impose indirect interference on real-time tasks [15], [16]. Particularly for the I/O read-and-write method, the data must be read from and written to the host memory. Therefore the impact of memory pressure needs to be quantified and it is worth conducting this experiment. As demonstrated in Figure 13 and 14, 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 15 and 16 show the average and the worst-case performance of the investigated data transfer methods 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.

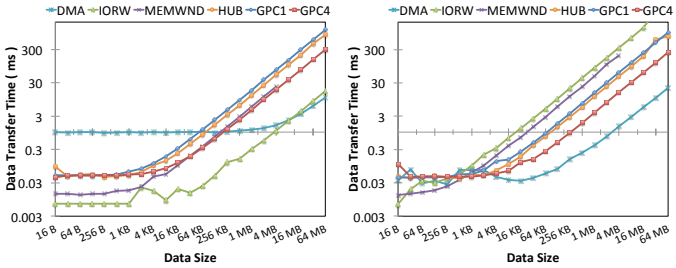
C. Concurrent Performance

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 17 and 18. 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



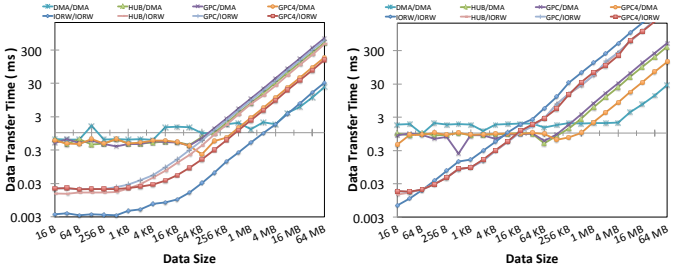
(a) Host to Device (b) Device to Host

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



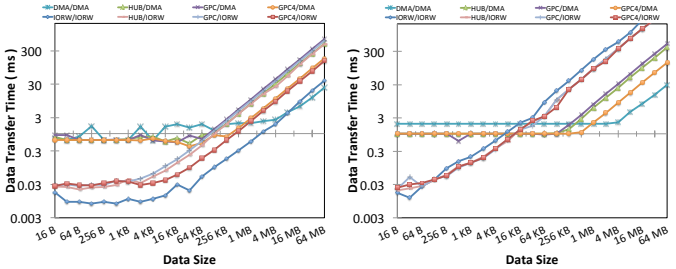
(a) Host to Device (b) Device to Host

Fig. 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

Fig. 17. The average performance of the combined data transfer methods for concurrent real-time tasks.



(a) Host to Device (b) Device to Host

Fig. 18. The worst-case performance of the combined data transfer methods for concurrent real-time tasks.

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 IV-A, 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 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.

D. Lessons Learned

We identified that the maximum data transfer performance for GPU computing can obtain from a combination of the hardware-assisted DMA and the I/O read-and-write access methods depending on the data size. The small data should be transferred using the I/O read-and-write access while the large data should benefit from the hardware-assisted DMA. It is important to find the data size boundary where the better performer changes. We made a significant optimization for the I/O read-and-write access implementation over previous work [9] whereas the hardware-assisted DMA has nothing to do with software optimization. It turned out that the I/O read-and-write access has an advantage until the data size reaches a few mega bytes, while the previous work reported that the hardware-assisted DMA performs better for a few kilo bytes of data using the same hardware device. This explains that a fine-grained tuning of the data transfer could dominate the performance of GPU computing, which has never been unveiled in previous work.

The novel findings also include that CPU workload could affect the performance of data transfers associated with GPU computing. This is because the I/O read-and-write access is performed by the CPU and the hardware-assisted DMA is also triggered by the commands sent from the CPU. To protect their performance from competing CPU workload, therefore, a real-time CPU scheduler plays a vital role. Although this result is intuitively apparent, this paper demonstrated it quantitatively using real-world systems.

Finally we provided a new approach to data transfers using on-chip microcontrollers integrated inside the GPU. While this approach does not provide a performance benefit for a single stream of the data transfer, we found that it is useful to support concurrent multiple streams overlapping their data transfers. The Helios project [17] showed that the integrated

microcontroller is useful to offload the packet processing job of the network interface card and mentioned that GPUs and compute devices could also benefit from microcontrollers, though there was no actual development. We revealed a way of using microcontrollers to hide the latency of the data transfer associated with GPU computing, and demonstrated its effectiveness. We believe that this is a novel idea toward the development of low-latency GPU computing technology.

V. Related Work

The zero-copy data transfer methods for low-latency GPU computing were developed for plasma control systems [3]. 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 [9]. 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 [18], [19]. 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 the performance characteristics of data transfers for GPU computing. We found that the hardware-assisted DMA and the I/O read-and-write access methods are the most effective to maximize the data transfer performance for a single stream, while the microcontroller-based method can overlap data transfers with the DMA and the IO read-and-write access methods reducing the total makespan of multiple data streams.

We also showed that the standard real-time CPU scheduler can shield the performance of data transfers from competing workload. They are novel findings and useful contributions to the development of low-latency GPU-accelerated systems.

The implementations of the investigated data transfer methods are provided in the Gdev project at http://github.com/cs005/gdev/.

In future work, we will investigate how to determine the choice of data transfer methods depending on the target system and workload. Since data transfers are abstracted by the API in terms of user programs, the runtime system must understand environments and choose appropriate data transfer methods to meet the performance and latency requirements of workload.

References

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

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

[3] 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).

[4] 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*, 2011, pp. 4889–4895.

[5] S. Hand, K. Jang, K. Park, and S. Moon, "PacketShader: a GPU-accelerated software router," in *Proc. of ACM SIGCOMM*, 2010.

[6] 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.

[7] 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.

[8] 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*, 2010, pp. 167–178.

[9] 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.

[10] 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.

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

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

[13] 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.

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

[15] S. Kato, Y. Ishikawa, and R. Rajkumar, "CPU Scheduling and Memory Management for Interactive Real-Time Applications," *Real-Time Systems*, 2011.

[16] T. Yang, T. Liu, E. Berger, S. Kaplan, and J.-B. Moss, "Redline: First Class Support for Interactivity in Commodity Operating Systems," in *Proc. of the USENIX Symposium on Operating Systems Design and Implementation*, 2008, pp. 73–86.

[17] E. Nightingale, O. Hodson, R. Mcllory, C. Hawblitzel, and G. Hunt, "Helios: Heterogeneous Multiprocessing with Satellite Kernels," in *Proc. of the ACM Symposium on Operating Systems Principles*, 2009.

[18] 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*, 2012, pp. 287–296.

[19] 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*, 2011, pp. 57–66.