# Zero-copy I/O processing for low-latency GPU computing

Shinpei Kato
Department of Information Engineering
Nagoya University

Jason Aumiller and Scott Brandt
Department of Computer Science
University of California, Santa Cruz

## ABSTRACT

Cyber-physical systems (CPS) aim to monitor and control complex real-world phenomena where the computational cost and real-time constraints could be a major challenge. Many-core hardware accelerators such as graphics processing units (GPUs) promise to enhancing computation, leveraging the data parallelism often found in real-world scenarios of CPS, but performance is limited by the overhead of the data transfer between the host and the device memory. For example, plasma control in the HBT-EP Tokamak device at Columbia University [11, 18] must execute the control algorithm in a few microseconds, but may take tens of microseconds to copy the data set between the host and the device memory. This paper presents a zero-copy I/O processing scheme that maps the I/O address space of the system to the virtual address space of the compute device, allowing sensors and actuators to transfer data to and from the compute device directly. Experiments using the plasma control system show a 33% reduction in computational cost, and microbenchmarks with more generic matrix operations show a 34% reduction, while in both cases, effective data throughput remains at least as good as the current best performers.

## Categories and Subject Descriptors

C.3 [**Special-purpose and Application-based Systems**]: Real-time and embedded systems; C.5 [**Computer System Implementation**]: Miscellaneous; D.3.4 [**Programming Languages**]: Processors—*Run-time environments*

## 1. INTRODUCTION

Cyber-physical systems (CPS) represent next generation networked and embedded systems, tightly coupled with computation and physical elements to control real-world phenomena. Their control algorithms, therefore, are becoming more and more complex, which distinguishes CPS from traditional safety-critical embedded systems in terms of the computational cost. In other words, "real-fast" (or really fast) is often as important as "real-time" (or predictable) in

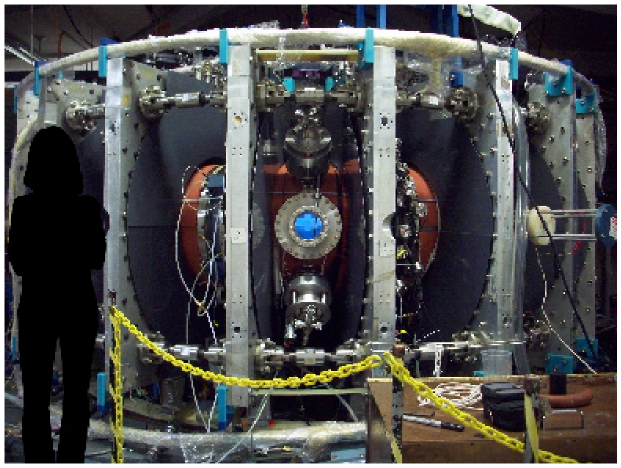


**Figure 1: Columbia's HBT-EP "Tokamak".**

CPS due to interaction speeds of the real world. This combined real-fast and real-time requirement of CPS imposes a core challenge on computer systems technology.

Plasma control for fusion is an application of energy CPS where complex algorithms must run at a very high rate. Figure 1 shows the HBT-EP high-beta Tokamak device at Columbia University [11, 18]. It must process 96 inputs and 64 outputs of data in a few microseconds to magnetically control the 3-D magnetohydrodynamic instabilities. An idea was to parallelize the algorithm using the graphics processing unit (GPU) and CUDA [16], which is a well recognized set of parallel computing technology. However, GPU programming is currently not tailored to integrate sensor and actuator devices. This is due to the fact that the current GPU programming stack is completely independent of I/O device drivers. Since it can take tens of microseconds to transfer hundreds of bytes data between the CPU and the GPU, the state of the art is not adequate to apply the GPU for plasma control in real-time. This is a significant problem for not only plasma control but also any CPS applications that are augmented with compute devices.

In order to effectively utilize GPUs for low-latency large-data CPS applications, platform systems are desired to provide a data communication scheme that can bypass the host computer, connecting GPUs and I/O devices directly. To the best of our knowledge, however, there is currently no standardized support for such a direct data transfer mechanism other than specialized commercial products for the

InfiniBand network [13]. The CUDA programming framework provides memory addressing technology that allows the GPU to directly access data allocated on the host for the purpose of mitigating the data transfer overhead, but this scheme is ill-suited for low-latency GPU computing due to the high cost of GPU's data access to the host memory. Given that GPUs are increasingly deployed in CPS applications [3, 10, 12, 14], and real-time GPU resource management techniques are starting to be developed [1, 2, 5, 6, 7, 8, 9], it is time to look into a tighter integration of I/O processing and GPU computing.

**Contribution:** This paper presents a new *zero-copy* I/O processing scheme for GPU computing. This scheme allows I/O devices to directly transfer data to and from the GPU by mapping of memory and I/O address space. We investigate the problem of existing schemes for low-latency GPU computing, and demonstrate performance advantages of our zero-copy scheme with a case study using Columbia's Tokamak device. Experimental results show the clear benefit of our zero-copy scheme on the achievable latency. We also provide microbenchmarks to highlight more generic properties of the I/O processing schemes. By clarifying these capabilities, we aim to not only improve performance but also broaden the scope of CPS that can benefit from state-of-the-art parallel computing technology.

**Organization:** The rest of this paper is organized as follows. Section 2 describes the system model and assumptions behind this paper. Section 3 presents our zero-copy I/O processing scheme, and differentiates it from the existing schemes. Section 4 describes details of system implementation. In Section 5, a case study of plasma control is provided to demonstrate the real-world impact of our contribution. Microbenchmarks are also used to evaluate more generic properties of the I/O processing schemes in Section 6. Section 7 introduces related work, and this paper concludes in Section 8.

## 2. SYSTEM MODEL

This paper assumes that the system is composed of compute devices, input sensors, and output actuators, in addition to typical components of a host computer. We restrict our attention to GPUs as auxiliary compute devices, which best embrace the concept of many-core computing in the state of the art. There must be at least three *device drivers* employed by the host computer to manage the GPU, sensors, and actuators, respectively. We assume that they are connected to the Peripheral Component Interconnect Express (PCIe) bus of the host computer. The contribution of this paper is not limited to the PCIe bus; it is applicable to any interconnect that is mappable to I/O address space. There are two kinds of memory associated with the address space. One is the host memory, also referred to *main memory*. The other is the device memory, which is encapsulated in the GPU. Both memory types are (and must be) mappable to the PCIe bus.

**Algorithm Implementation:** The control algorithm is parallelized and offloaded to the GPU. We use CUDA to implement the algorithm, but any programming language for the GPU is available under our model, because the application programming interface (API) considered in this paper does not depend on a specific programming language. All input data come from the sensor modules, while all output data go to the actuator modules. The buffers for the data can be allocated to any place visible to I/O address space. According to the control system design, the data may or may not be further copied to other buffers. In either case, they are expressed as data *arrays* in the control algorithm.

**PCIe BAR:** The current form of the GPU is a PCI device. Such a PCI-based compute device is typically designed to 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. For example, NVIDIA GPUs 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 given their different sizes of memory space. Specifically the BAR1 region can be directly accessed by the GPU using the unified memory addressing (UMA) mode that allows all memory objects allocated to the host and the device memory to be referenced by the same address space. This BAR1 region can also be accessed by the CPU using the I/O remapping function supported by the operating system kernel as well as I/O devices by obtaining the physical I/O address of the corresponding BAR1 region. This paper makes use of the latter technique to achieve direct data communications between the GPU and I/O devices.

**Limitation:** There are several other assumptions that simplify our system model. The system contains only the single real-time process (task) that executes the control algorithm, except for trivial background jobs to run the system. We ignore the problem of shared resources among multiple tasks, which have been addressed elsewhere [1, 5]. We also focus on a single instance of the GPU to implement the control algorithm. This is not a conceptual limitation of this paper; the algorithm implementation could just as easily use multiple GPUs.

## 3. I/O PROCESSING SCHEMES

This section presents a new zero-copy I/O processing scheme for GPU computing, which differs from existing schemes in that both GPU execution and data transfer times are minimized to achieve the goal of low-latency GPU computing. In the rest of this section, we first investigate two existing schemes, *H+D* and *Hpin*, that are already supported by CUDA. We then present a new scheme called *Dmap* and introduce a hybrid variant of *Dmap* and an existing one, suited for a specific case.

### 3.1 Host and Device Memory (*H+D*)

This is the most common scheme of GPU computing in the literature. In this scheme, there is space overhead in that the input and output data exist in both the device and the host memory at the same time and must be explicitly copied between them. Furthermore, there is a time penalty incurred

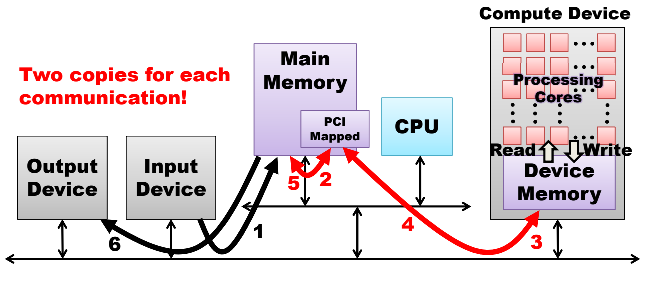


**Figure 2: The traditional *H+D* scheme.**

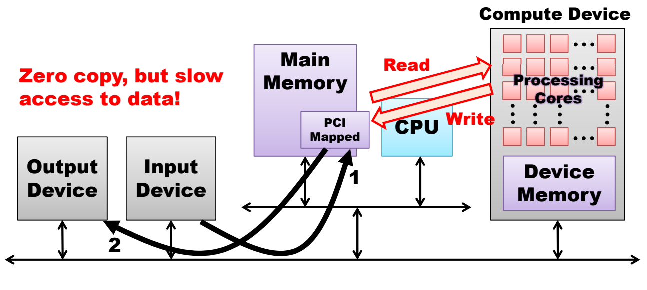


**Figure 3: The traditional *Hpin* scheme.**

to copy data at a cost nearly proportional to the data size. When applying this scheme, we allocate the same size of buffers to the host and the device memory individually and copy data between them explicitly.

Figure 2 illustrates an overview of how this scheme works:

1. The device driver of the input device configures the input device to send data to the allocated space of the host memory.
2. The device driver of the GPU copies the data to the PCI-mapped space of the host memory, which is accessible to the device memory.
3. The device driver of the GPU further copies the data to the allocated space of the device memory. Now, the GPU can access the data.
4. When GPU computation is completed, the device driver of the GPU copies the output data back to the PCI-mapped space of the host memory.
5. The device driver of the GPU further copies the output data back to the allocated space of the host memory.
6. Finally, the device driver of the output device configures the output device to receive the data from the allocated space of the host memory.

As described above, the *H+D* scheme incurs overhead to copy the same data twice for each direction of data transfer between the host and the device memory. This overhead might be a crucial penalty for low-latency GPU computing.

## 3.2 Host Pinned Memory (*Hpin*)

As an alternative to allocating the buffers to both the host and the device memory, we can allocate the buffers to page-locked PCI-mapped space of the host memory, also known as *pinned* host memory. Since recent GPU architectures support unified addressing, this memory space can be referenced by the GPU. A major advantage of this scheme is

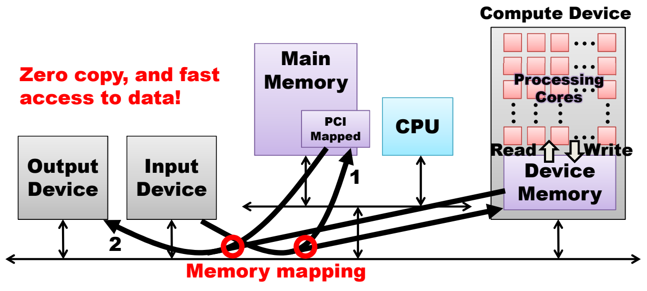


**Figure 4: The new zero-copy *Dmap* scheme.**

that the input and the output devices can also directly access this memory space, which means that there is no need for intermediate buffers and data copies to have the GPU access the data.

Figure 3 illustrates an overview of how this scheme works. Unlike the *H+D* scheme, the data transfer flow is pretty simple. There are no additional data copies required, since PCI-mapped space is directly accessible to the input and the output devices. It is also pinned to always reside in the host memory, and therefore the GPU can read and write the data directly. However, this data access is expensive as it is a PCIe communication.

## 3.3 Device Memory Mapped to Host (*Dmap*)

We now present *Dmap*, which overcomes the problems of the *H+D* and the *Hpin* schemes. The key to the *Dmap* scheme is having the PCIe BAR point to the allocated space of the device memory, while mapping the allocated space of the host memory to the PCIe BAR as well. As a result, when the input device sends data to the PCI-mapped space of the host memory, the data seamlessly appear on the corresponding allocated space of the device memory. This scheme, hence, does not require the CPU to intermediate to copy the data between the host and the device memory, which removes the latency of data transfer observed in the *H+D* scheme. On the other hand, it also maintains the performance benefit of having the data on board, solving the problem of slow data accesses faced in the *Hpin* scheme.

Assuming that some space is already allocated to the device memory, the system is required to support the following functions to have I/O devices directly access this memory space:

- **Mapping Memory:** As technology reads today, PCIe BARs are the most reasonable windows that see through the device memory from the host and I/O space. Therefore, we first need to reserve the corresponding size of PCIe BAR space, and next map it to the allocated space of the device memory. Now, if the user requests to access it from the host program, the system needs to further remap it to the user buffers.
- **Unmapping Memory:** PCIe BARs are limited resources. Most GPUs currently provide at most 128MB for a single BAR. Therefore, unmapping the device memory from the BAR is an essential function for this scheme.
- **Getting Physical Address:** The mapped space is typically referenced as virtual memory space of the GPU or the CPU. However, I/O devices often target

physical address for data transfer. Hence, the system needs to maintain the physical address of the PCI-mapped space, and relay it to the device drivers of I/O devices.

In fact, PCIe BARs are not only the windows that can communicate with the device memory. Recent GPUs support special windows upon the memory-mapped I/O (MMIO) space. To simplify the discussion, this paper focuses on PCIe BARs, but the same concept of zero-copy I/O processing can be also applied to any mapping method.

### 3.4 Device Memory Mapped Hybrid (*DmapH*)

This is a hybrid of the *Dmap* and the *H+D* schemes, which is in particular suited to communicate between the host and the device memory. Applications of CPS using I/O devices, thereby, may not benefit from this scheme; if the host memory is used to store some data, this scheme is still effective.

This scheme is the same as *Dmap* as far as memory allocation and mapping. For transferring data from the host to the device memory, we have the host processor reference the device memory directly through the mapped region, like the *Dmap* scheme. However, we perform an explicit copy from the device to the host memory for an opposite direction of data transfer. The motivation to do so is that writing to the host memory is more expensive than reading, due to functionality of the host memory management unit (MMU). We evaluate the impact of this scheme in Section 6.

## 4. SYSTEM IMPLEMENTATION

GPU programs often execute in *virtual address spaces*. In CUDA programming, for instance, a device pointer acquired by the memory allocation API functions, such as `cuMemAlloc()` and `cuMemAllocHost()`, represents a virtual address. The relevant data-copy API functions, such as `cuMemcpyHtoD()` and `cuMemcpyDtoH()`, also use this pointer to the virtual address rather than a physical address. As long as the programs remain within the GPU or the CPU, this is a suitable addressing model. However, CPS applications often require I/O devices for sensing and actuation. The current software stack and the API design of GPU programming force these applications to use the host memory as an intermediate buffer to bridge between the I/O device and the GPU. No prior system implementation has allowed data to move directly between them.

In this section, we present our system implementation of the *Dmap* and *DmapH* schemes using Gdev [8], an open-source facility of the device driver and the runtime library for NVIDIA GPUs. A key limitation is that the data access of I/O devices is limited to the I/O address space. However, because the GPU is typically connected to the system via the PCIe bus, we can map the virtual address space of the device memory to the reserved I/O address space of the PCI bus, as mentioned in Section 3.3. In this way, I/O devices can directly access data in the device memory. Specifically, their device drivers can configure their hardware direct memory access (DMA) engines to target the physical address associated with the PCI-mapped space of the device memory.

Our system implementation adds the three functions presented in Section 3.3 to Gdev. While Gdev is a Linux kernel module for first-class GPU resource management, it also provides an implementation of the CUDA Driver API [16] for user programming. The three functions are wrapped by the Gdev-original extended API functions, `cuMemMap()`, `cuMemUnmap()`, and `cuMemGetPhysAddr()`, which are compatible to the form of the CUDA Driver API. We now provide the details of these functions:

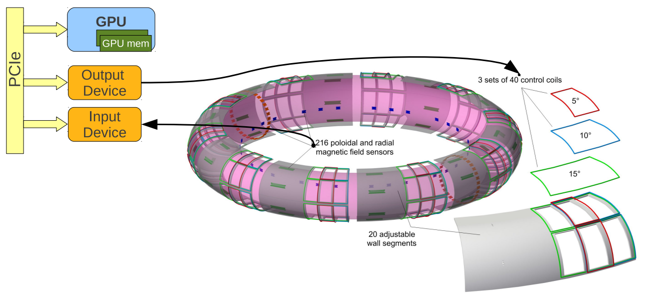
- **Mapping Memory:** We assign the second PCIe BAR region, *i.e.*, BAR1 (among the five of those supported by hardware as a window of the device memory), because this is the largest region, often equal to 128MB, for NVIDIA GPUs. Since these GPUs provide an MMIO register to point the PCIe BAR to any virtual address of the device memory, we create special virtual address space dedicated to the PCIe BAR when the system is loaded. When the user context requests mapping of some device memory object, we first find the physical pages allocated to this memory object. We next set up the page table of the GPU so that these physical pages are referenced by the virtual address space dedicated to the PCIe BAR. Now, these physical pages are referenced by two virtual address spaces: one dedicated to the PCIe BAR and the other associated with the user context. As a result, the data written to the PCIe BAR can be referenced by the user context and the mapping is done. If the user context furhter needs to map the same memory to the host virtual address, we can use a Linux kernel primitive such as `ioremap()`.
- **Unmapping Memory:** To unmap the allocated space of the device memory from the PCIe BAR, we simply delete the corresponding record of the page table. If host memory is also mapped, we also call a Linux kernel primitive such as `iounmap()`.
- **Getting Physical Address:** The operating system kernel usually provides a function to return the physical addresses of PCIe BARs. In the Linux kernel, we can use `pci_resource_start()` for this purpose.

## 5. CASE STUDY

In this section, we provide a case study of magnetic control of 3-D plasma instabilities using the HBT-EP Tokamak equipped with the GPU and the aforementioned I/O processing schemes. This control system requires low latency and high computing capabilities to achieve a sampling period of the order of ten microseconds, while processing 96 inputs and 64 outputs of 16-bit data with a complex algorithm. Implementing the algorithm on the CPU failed due to insufficient real-time performance. The case study presented herein, therefore, is significant in that we look into a possibility of GPU implementations for the plasma control system.

Figure 5 shows a system architecture used in this case study. The control input comes from a set of magnetic sensors through a D-TACQ ACQ196 digitizer, and the resulting control signal is sent to two D-TACQ AO32 analog output modules to excite control coils. These input and output modules are connected to the NVIDIA GTX 580 upon the PCIe bus. We evaluate three schemes against this system architecture from the viewpoint of GPU execution costs for the control algorithm and data I/O costs for the data transfer. Each scheme is applied as follows:

- In the *H+D* scheme, the device driver of the digitizer transfers the input data set to the buffers allocated



GPU
GPU mem
PCIe
Output Device
Input Device
3 sets of 40 control coils
5°
10°
15°
216 poloidal and radial magnetic field sensors
20 adjustable wall segments

**Figure 5: HBT-EP magnetic sensors and control coils connected with the GPU.**

on the host memory. The control program copies this data set to the device memory via PCI-mapped host memory space, and the parallelized control algorithm runs on the GPU with the data on the device memory. Once the output data set is produced by the GPU, the control program copies it back to the host memory, and it is finally pulled by the device driver of the analog output modules. This is the traditional form of GPU computing.

- The *Hpin* scheme pins the input and output buffers to PCI-mapped host memory space. Since the data set pushed and pulled by the device drivers of the I/O modules is directly accessible to the GPU, there is no need to perform data copies. However, this scheme must compromise the latency of data access imposed on the GPU when executing the control algorithm.
- Similarly to the *Hpin* scheme, the *Dmap* scheme presented in this paper uses pinned PCI-mapped host memory space to allocate the input and out buffers, and further maps it to the device memory through PCI BAR space. Thus, there is no need of data copies while the data access of the control algorithm is limited within the device memory.

This paper does not provide the details of the control algorithm which is outside the scope of this paper. The outline of the control system implementation is that the host program launches the device program on the GPU once at the beginning when the system is loaded. The device program polls until the input data set arrives. This is due to a requirement of low-latency computing. The input and output modules are configured to write the input data to and read the output data from the specified PCI regions through DMA, respectively. These PCI regions are directly mapped to the device memory space allocated by the control system, using the *Dmap* scheme presented in this paper. In consequence, once the input and output modules are configured, and the device program is launched at the beginning, the algorithm can keep executing on the GPU, without accessing the CPU and the host memory at all.

We now show that the *Dmap* scheme reduces both GPU execution costs for the control algorithm and data I/O costs for the data transfer. Figure 6 shows the result of experiments conducted under the three different schemes, respectively. Note that the values of the GPU execution and the data I/O costs are *estimations*. In the experiment, we could measure the sampling periods of the plasma control system achieved the *Dmap* and the *Hpin* schemes while there was

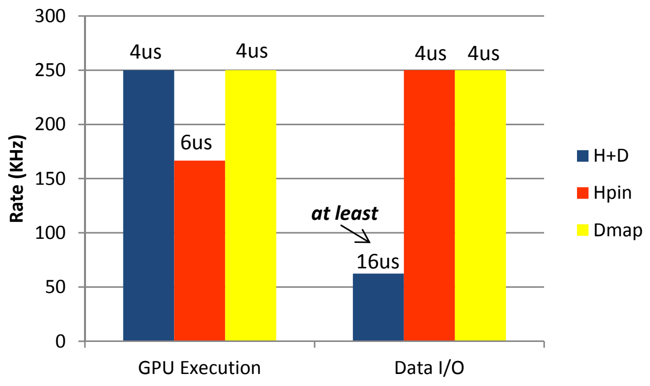


**Figure 6: *Estimated* GPU execution and data I/O costs of the plasma control system.**

no point of implementing the *H+D* scheme in terms of the total latency. We could also measure the round-trip time of the data transfer between the sensor/actuator and the GPU. We estimate details of the GPU execution and the data I/O costs based on these measured results. In particular, our assumption is that the *Dmap* and the *H+D* schemes should have the same algorithm cycle, while the the *Dmap* and the *Hpin* schemes should have the same data latency, by nature. The *Dmap* scheme achieves the highest rate in both algorithm execution and data transfer. The remaining two schemes, on the other hand, compromise one or the other of them. The *H+D* and the *Dmap* schemes exhibit the same performance level for algorithm execution since they both use the device memory, while the data transfer latency of the *Hpin* and the *Dmap* schemes are equivalent since they both remove data copies. Comparing the *H+D* and the *Hpin* schemes, one can also see that the impact of overhead introduced by data copies, *i.e.*, 16$\mu$s, is greater than that introduced by the GPU accessing pinned host memory space, *i.e.*, 6$\mu$s, on the overall system performance. Curiously, there is additional latency of 4$\mu$s observed when running the control system. We suspect that this latency comes from some interactions among the host computer, the graphics card, and the I/O modules. Lessons learned from this evaluation are summarized as follows:

- Zero-copy I/O processing is very effective for this control system, reducing the latency of data transfer from 16$\mu$s to 4$\mu$s. The speed-up ratio is 4×.
- Furthermore, the *Dmap* scheme reduces the cycle time of algorithm execution from 6$\mu$s to 4$\mu$s. The speed-up ratio is 1.5×. Since the HBT-EP Tokamak accommodates up to 216 inputs, meaning that the cycle time of algorithm execution is more dominated by data accesses, the benefit of the *Dmap* scheme over the *Hpin* scheme would be more significant for a larger scale of plasma control.

The above measurement explains that the control system can operate at a latency of 16$\mu$s. The data transfer from and to the input and output modules takes 4$\mu$s each. The algorithm execution takes 4$\mu$s. Adding additional latency of 4$\mu$s, the total control rate must be able to achieve 16$\mu$s. Figure 7 depicts the screenshot of the oscilloscope where we measure the signals of the input and the output modules. The topmost and middle signals represent the input and

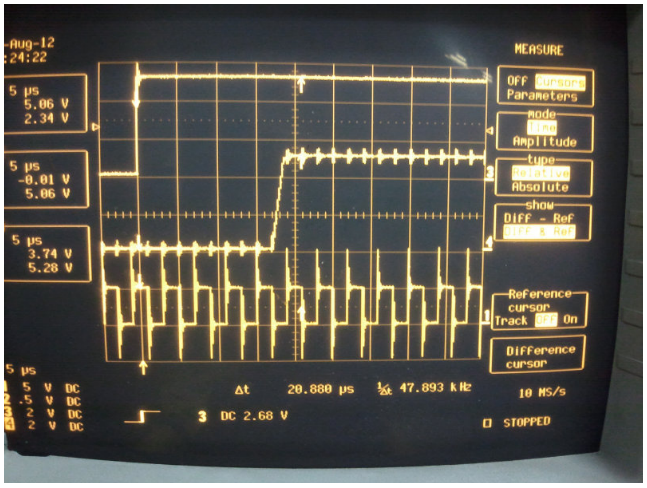


Figure 7: Screenshot of the oscilloscope.

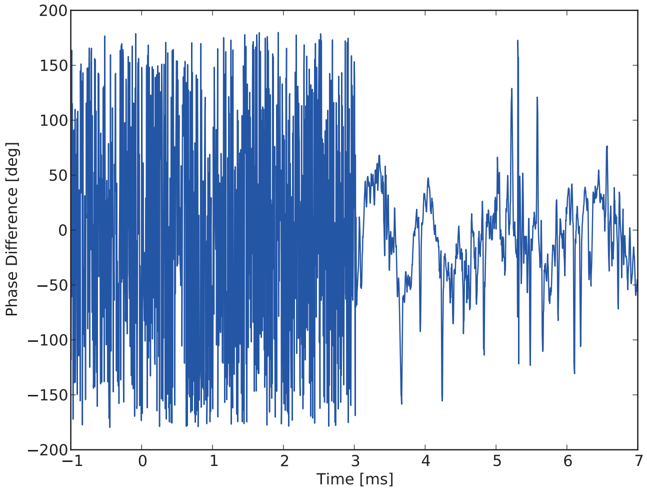


Figure 8: Phase difference observed with the base output signal.

the output, respectively, while the lower signal indicates the base clock. The grid spacing of the X axis is 5$\mu$s. The time interval from the first downward edge in the clock signal after the input signal goes up to the instant when the output signal starts uprising is almost equal to 16$\mu$s. This means that the total control processing time is 16$\mu$s.

We next demonstrate that the control system is running properly at a rate of 16$\mu$s. The control input comes from a set of magnetic sensors arranged in a ring, as illustrated in Figure 5, and the magnetic field that they measure is rotating, whose orientation is described by a *phase*. Ideally, the phase is equivalent to multiplication of time and frequency. To control this mode, the control system needs to produce a control signal that generates an equal and opposite field, which also needs to rotate. Obviously, the two fields should have a constant phase difference, because it is given by multiplication of the control processing time and the rotation frequency. However, in practice, the rotation frequency is not constant but is changing. As a result, the phase difference appears to oscillate, with the base output signal, which can be found as spikes in Figure 8. Now, we measure the phase difference with the output signal time-shifted by 16$\mu$s. In other words, the effective control system latency is reduced by 16$\mu$s. As shown in Figure 9, the spikes are now all removed. This indicates that the control system is perfectly in phase with the mode, and the effective control system latency now must be zero, *i.e.*, the actual latency is 16$\mu$s.

Finally, we discuss practical findings of plasma control regarding the HBT-EP "Tokamak" device. Figure 10 shows a comparison of the average perturbation amplitudes with different phasing. The control system incorporates four arrays of magnetic sensors and control coils, each of which controls one specific mode. They are placed at different poloidal angles around the toroidal ring. Due to their different locations, they measure slightly different amplitudes. From this experiment, we find that feedback at 280 degrees excites perturbation, while feedback at 100 degrees is the right range for suppression. As compared to no feedback scenario ("No FB" in the figure), for example, we find that we can suppress the strength of the rotating field by up to 30% for any mode observed in this experiment.

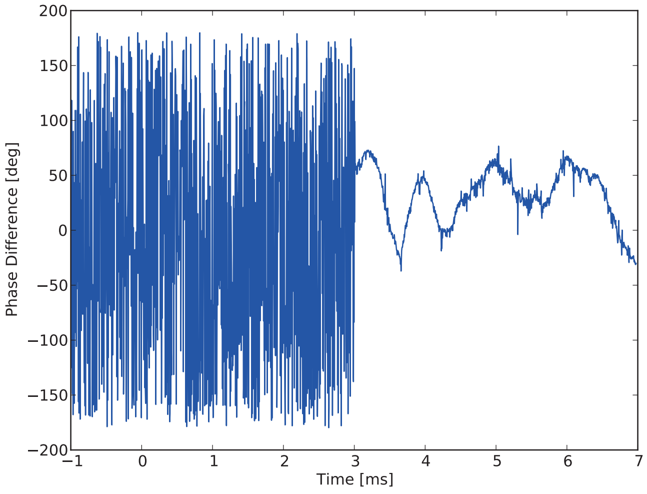


Figure 9: Phase difference observed with the output signal shifted by 16$\mu$s.

## 6. BENCHMARKING

We now evaluate generic properties of the presented I/O processing schemes. To do so, we remove constraints of I/O devices, and focus on naive microbenchmarking programs of matrix operations performing with the host and the device memory. In particular, timing analysis of addition and multiplication of varying sized matrices is conducted. These programs are considered as the most basic parallel computing programs also used in prior work [19]. Since our focus is on data access and I/O processing, but not on computation, we choose matrix addition as a microbenchmark, as it is a straightforward operation for the GPU. Matrix multiplication, on the other hand, is also included to briefly illustrate how an increase of computational complexity and data accesses affects time to completion. We also showcase effective host read and write throughput for each I/O processing scheme. This benchmarking clarifies the capabilities of the presented scheme, not specific to plasma control but applicable to generic low-latency GPU computing.

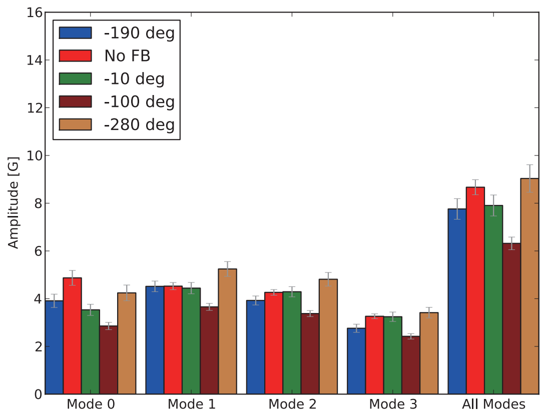


**Figure 10: Practical findings of plasma control.**

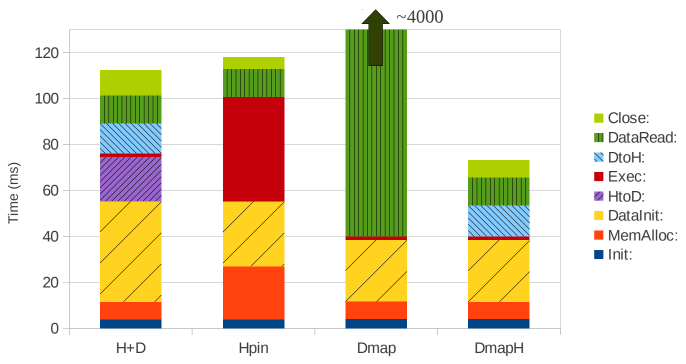


**Figure 11: Details of matrix addition.**

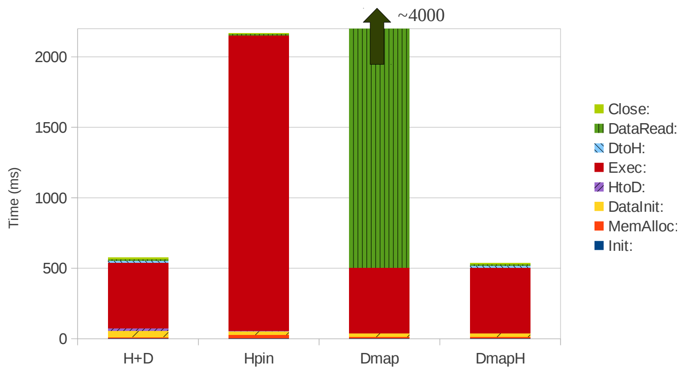


**Figure 12: Details of matrix multiplication.**

The microbenchmarking experiments are conducted using the NVIDIA Quadro 5000 GPU (448 cores and 2.5GB memory), which is more end-user oriented than the Tesla C2050 used in the case study. Note that the supported number of compute cores is the same between these two GPUs.

## 6.1 Matrix Operations

We first demonstrate performance details of matrix operations. To highlight the details, we use a large size of $2048 \times 2048$ matrix in this experiment. The properties of operations focused on are listed as follows:

- **Init:** GPU initialization time.
- **MemAlloc:** Memory allocation time, with respect to the host and/or the device memory.
- **DataInit:** Matrix initialization time.
- **HtoD:** Copy time from the host to the device memory (only *H+D*).
- **Exec:** Execution time of the kernel function.
- **DtoH:** Copy time from the device to the host memory (only *H+D* and *DmapH*).
- **DataRead:** Access time to read the result.
- **Close:** Context destroying time.

Figure 11 shows where the system spends its time in performing a $2048 \times 2048$ integer matrix addition using each of the four schemes. First, it is clear from Figure 11 that using our *DmapH* scheme the total time to completion is less than the others, almost 34% less than its nearest competitor, *i.e.*, the *H+D* scheme.

A more interesting observation is the comparison of how much time is spent for each sub-task. For most time categories, the *DmapH* scheme seems to enjoy the best of each of the other three. The memory allocation time for the *DmapH* scheme is nearly identical to that of the *H+D* and the *Dmap* schemes, and clearly less than the *Hpin* scheme. The same is true for the execution times. Similarly, for data initialization, the *DmapH* scheme is just as good as the *Hpin* and the *Dmap* scheme, which are superior to the *H+D* scheme.

The most notable difference among the four schemes is the data read time for the *Dmap* scheme, which is orders-of-magnitude greater than the rest. A $2048 \times 2048$ integer matrix represents 16MB of data, and the *Dmap* scheme reads 4 bytes at a time across the PCIe bus. This was the motivation for our *DmapH* scheme. Instead of reading one matrix element at a time, the *DmapH* scheme first copies the data to the host before reading. This evinces that the *Dmap* scheme may suffer from a large size of data, while it was very effective for the case study presented in Section 5 that deals with 96 inputs and 64 outputs of 16-bit data, *i.e.*, 2KB of data.

The same anomaly is present in the execution time for the *Hpin* scheme – the GPU must read one element at a time from the host memory, while in the other three schemes the data reside on the GPU during computation. This makes the *Hpin* scheme increasingly inferior as the data size grows.

Figure 12 shows the same time analysis for matrix multiplication. The only difference appears in the execution times. This is not surprising as the multiplication experiments are exactly the same as the addition ones except for the kernel function on the GPU. In fact, if execution times are omitted, Figures 11 and 12 would look almost identical. This observation implies that the zero-copy I/O processing schemes are not really appreciated by strongly compute-intensive applications.

While the above experiments do present a real-world scenario, in a real-time system it is more likely that tasks are performed repeatedly, and therefore the GPU initialization and closing costs might occur only once – the same con-

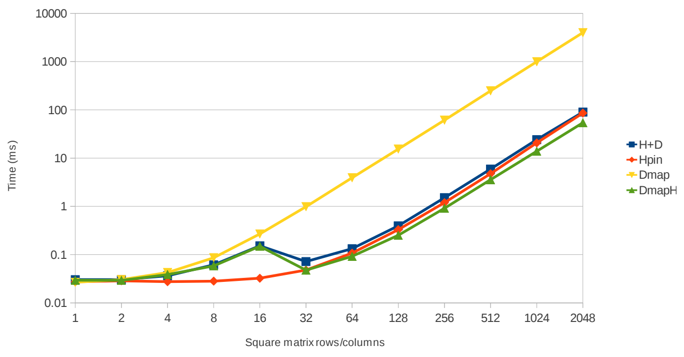


**Figure 13: Total matrix addition times.**

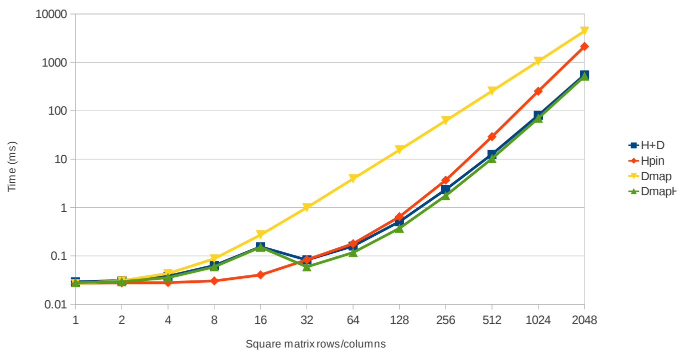


**Figure 14: Total matrix multiplication times.**

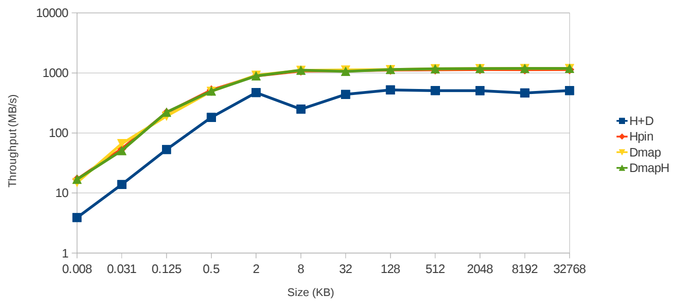


**Figure 15: Host write throughput.**

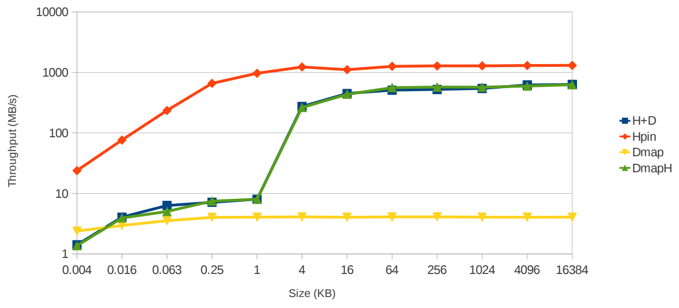


**Figure 16: Host read throughput.**

text loaded a priori would remain active while only the data might change. In addition, it could be the case that the memory allocated for the task could be reused, and only the contents modified. For these reasons, the remainder of our analysis focuses only on read, write, transfer, and execution.

Figure 13 shows the time to completion of matrix addition as a function of matrix size at the logarithmic scale. The *Hpin* scheme appears to be the best performer until a matrix size of $32 \times 32$, corresponding to a data size of 4KB for each matrix, at which point the *DmapH* becomes superior. This is also reflected in the matrix multiplication times, as shown in Figure 14. One thing to note is the growth rate of the *Hpin* in matrix multiplication; since each thread must perform multiplication and addition $n^2$ times, as compared with one addition in matrix addition, the number of reads that occur across the PCIe bus increases by a greater exponential factor. We expect that the time for the *Hpin* scheme would eventually surpass the *Dmap* scheme, as the trend in the graph indicates.

## 6.2 Data Throughput

Finally, we evaluate the data throughput of the presented I/O processing schemes, independent of computational units. In other words, this evaluation shows the pure performance of data communication between the host and the device memory. We use the term "effective throughput" to mean $(size/time)$ where *time* is measured from the beginning of data initialization to the point at which it is actually available to the GPU. For example, in the case of the *H+D* scheme, this corresponds to the total time for the host processor to write to each element in the data structure, which is in the host memory, plus the time to copy the data to the device memory.

Figure 15 shows that the write throughput of the *DmapH* is better than the *H+D* scheme by about a factor of 2 and at least as good as the others, which all coincide almost exactly in the graph. Meanwhile, Figure 16 paints a slightly different picture for read throughput. It should not be surprising that the *Hpin* scheme outperforms the rest as it represents the host iterating over a data structure that is already in the host memory. The *H+D* and the *DmapH* schemes, on the other hand, must first copy the data from the device to the host memory, and they achieve roughly equal performance. Note that they coincide in the graph. Finally, the *Dmap* scheme is the weakest performer due to a large number of small reads that occur across the PCIe bus.

## 7. RELATED WORK

Recent GPU architectures [15, 17] support unified virtual addressing (UVA), which creates a single address space for the host memory and multiple instances of the device memory. In particular, NVIDIA GPUs and CUDA provides the `cuMemcpyPeer()` function to copy data directly between GPUs over the PCIe bus without the involvement of the CPU or the host memory. Yet, it is restricted for use between NVIDIA GPUs.

GPUDirect [13] and its variant are vendor-specific proprietary products to directly connect the GPU and the InfiniBand network card. They may use a similar approach to zero-copy I/O processing presented in this paper, but are not applicable to arbitrary I/O devices. Their architectures are also not open to the public.

CGCM [4] is an automatic system for optimizing CPU-GPU communication. It uses compiler modifications in conjunction with a runtime library to manipulate the timing of events in a way that effectively reduces data transfer overhead. PTask [19] provides programming abstractions to use the GPU, which are supported by the OS. One aspect of PTask is the use of a data-flow model to minimize data communication between the host processor and the GPU. RGEM [6] aims to bound blocking times by dividing data into chunks, since high-priority tasks may be blocked due to data transfer in real-time systems. This effectively creates preemption points to allow finer grained scheduling of GPU tasks to fully exploit the ability to concurrently copy data and execute code. All these prior work primarily focus on the existing data communication basis, while we present new zero-copy I/O processing schemes.

## 8. CONCLUSION

In this paper, we have presented a new approach to GPU acceleration of low-latency CPS applications. This approach uses mapping of the device memory and the PCIe BAR region, and configures I/O devices to transfer data to and from the corresponding PCIe address space instead of the host main memory buffer space. The plasma control system, developed as an example of CPS applications, demonstrated that the presented zero-copy I/O processing scheme achieved a very high rate of 16$\mu$s for full plasma control processing. The additional microbenchmarking evaluation also clarified an advantage of zero-copy I/O processing among those currently implementable for GPU computing. We believe that the contribution of this paper would facilitate a grander vision of CPS using heterogeneous compute devices as well as GPUs.

In future work, we extend the zero-copy I/O processing scheme for multiple contexts. This extension is essential in a sense that we can control multiple plants with a single GPU. A key challenge is to ensure exclusive direct access to the same memory space; device drivers and runtime libraries are not able intermediate during DMA transfers. We also plan to apply the same zero-copy scheme to arbitrary I/O devices such as Ethernet and Firewire.

## 9. ACKNOWLEGDEMENT

We thank the HBT-EP Team at Columbia University for providing figures 7 to 10. We also thank the PathScale developers for sharing their technology with this research project.

## 10. REFERENCES

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

[2] 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.

[3] M. Hirabayashi, S. Kato, M. Edahiro, and Y. Sugiyama. Toward GPU-accelerated traffic simulation and its real-time challenge. In *Proc. of the International Workshop on Real-Time and Distributed Computing in Emerging Applications*, 2012.

[4] T. Jablin, P. Prabhu, J. Jablin, N. Johnson, S. Beard, and D. August. Automatic CPU-GPU communication management and optimization. In *Proc. of the ACM Conference on Programming Language Design and Implementation*, 2011.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] C. Liu, J. Li, W. Huang, J. Rubio, E. Speight, and X. Lin. Power-Efficient Time-Sensitive Mapping in Heterogeneous Systems. In *Proc. of the International Conference on Parallel Architectures and Compilation Techniques*, 2012.

[10] R. Mangharam and A. Saba. Anytime Algorithms for GPU Architectures. In *Proc. of the IEEE Real-Time Systems Symposium*, pages 47–56, 2011.

[11] D. Maurer, J. Bialek, P. Byrne, B. D. Bono, J. Levesque, and e. a. B.Q. Li. The high beta tokamak-extended pulse magnetohydrodynamic mode control research program. *Plasma Physics and Controlled Fusion*, 53, 2011.

[12] 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.

[13] Mellanox. NVIDIA GPUDirect Technology– Accelerating GPU-based Systems, 2010.

[14] P. Michel, J. Chestnutt, S. Kagami, K. Nishiwaki, J. Kuffner, and T. Kanade. GPU-accelerated Real-Time 3D Tracking for Humanoid Locomotion and Stair Climbing. In *Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 463–469, 2007.

[15] NVIDIA. NVIDIA's next generation CUDA computer architecture: Fermi, 2009.

[16] NVIDIA. CUDA C Programming Guide Version 4.2, 2012.

[17] NVIDIA. NVIDIA GeForce GTX 680: The fastest, most efficient GPU ever built, 2012.

[18] N. Rath, J. Bialek, P. Byrne, B. DeBono, J. Levesque, B. Li, M. Mauel, D. Maurer, G. Navratil, and D. Shiraki. High-speed, multi-input, multi-output control using GPU processing in the HBT-EP tokamak. *Fusion Engineering and Design*, 2012.

[19] C. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. PTask: Operating system abstractions to manage GPUs as compute devices. In *Proc. of the ACM Symposium on Operating Systems Principles*, 2011.