Zero-Copy I/O for Low-Latency GPU Computing

Shinpei Kato
Dept. Information Engineering
Nagoya University

Nikolaus Rath
Dept. Applied Physics and Applied Mathematics
Columbia University

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

ABSTRACT

Cyber-physical systems (CPS) aim to control complex real-world phenomenon. Due to real-time constraints, however, the computational cost of control algorithms is becoming a major issue of CPS. Parallel computing of the control algorithms using state-of-the-art compute devices, such as graphics processing units (GPUs), is a promissing approach to reduce this computational cost, given that real-world phenomenon by nature compose data parallelism, yet another problem is introduced by the overhead of data transfer between the host processor and the compute device. As a matter of fact, plasma control requires an order of a few microseconds for the control rate, while today's systems may take an order of ten microseconds to copy the corresponding problem size of data between the host processor and the compute device, which is unacceptable lantecy. In this paper, we propose a zero-copy I/O processing scheme that enables sensor and actuator devices to directly transfer data to and from the compute device. The basic idea behind this scheme is to map I/O address space, accessible to sensor and actuator devices, to virtual memory space of the compute device. The results of experiments using the real-world plasma control system demonstrates that the computational cost of the plasma control algorithm is reduced by 33% under our new scheme. We further provide the results of microbenchmarking to show that more generic matrix computations are completed in 34% less time than current methods, using our new scheme, while effective data throughput remains at least as good as the current best performers.

Keywords

GPGPU, Zero-Copy I/O, Fusion and Plasma, CPS

1. INTRODUCTION

Cyber-physical systems (CPS) are next generations of networked and embedded systems, tightly coupled with computation and physical elements to control real-world phenomenon. Control algorithms of CPS, therefore, are becoming more and more complex, which makes CPS distinguished from traditional safety-critical systems. In CPS applications, "real-fast" is often as important as "real-time", while safety-critical systems are likely to have only the real-time constraint. Such a double-edge real-time and real-fast requirement of CPS, however, has imposed a core challenge on systems technology. In this paper, we tackle this problem with a specific example of plasma control.

Plasma control for fusion is an applications of energy CPS, where complex algorithms must be computed at a very high rate. Figure 1 shows the HBT-EP Tokamak at Columbia University [11, 17] that magnetically controls the 3-D perturbed equilibrium state of the plasma [1]. It is required to process 96 inputs and 64 outputs of 16-bit data at a control rate of a few microseconds. An initial attempt of the Columbia team employed fast CPUs or FPGAs, but even the simplified algorithm failed to run within 20μs. An alternative approach was to parallelize the algorithm using the graphics processing unit (GPU) and CUDA [16] – the most successful massively parallel computing technology. However, the current system for GPU computing is not designed to integrate sensor and actuator devices with the GPU. This is attributed to the fact that the current software stack of GPU computing is independent of I/O device drivers. Since it might take tens of microseconds to transfer hundreds-of-bytes data between the CPU and the GPU, it is not affordable for the current software stack to apply the GPU for plasma control in real-time. This is a signficant problem not only for plasma control but also for any applications of CPS that are augmented with compute devices.

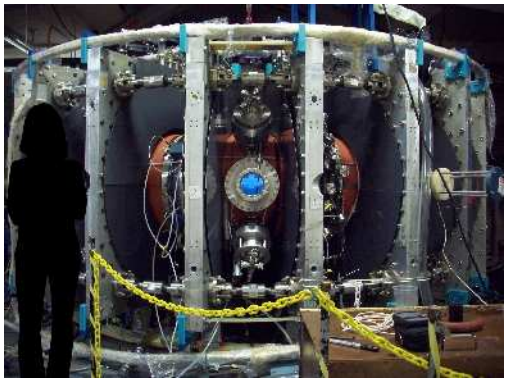

Figure 1: The HBT-EP "Tokamak" at Columbia University.

In order to utilize the GPU for applications of CPS, the system is required to support a method of bypassing data transfer between the CPU and the GPU, instead connecting the GPU and I/O devices directly. To the best of our knowledge, however, there is currently no generic systems support for such a direct data transfer machanism except for specialized commercial products for the InfiniBand network [13]. As a way of eliminating the data transfer cost between the CPU and the GPU, the current programming framework often supports host memory allocation, which enables the GPU to access data on the host memory; however, it is unclear if this scheme is best suited for low-latency GPU computing, because the data access to the host memory from the GPU is expensive. Given that GPUs are increasingly deployed in the domain of CPS [4, 10, 12, 14], and basic real-time resource management techniques for the GPU started to be disclosed [2, 3, 5, 6, 7, 8, 9], it is time to look into a tight integration of I/O processing and GPU computing.

Contribution: In this paper, we present a zero-copy I/O processing scheme for GPU computing. This scheme enables I/O devices to directly transfer a limited size of data to and from the GPU, by coordinating their device drivers. We also investigate a possibility of the exisiting schemes to support low-latency GPU computing, and identify an advantage of our new scheme. To do so, we provide a case study using the Columbia University's Tokamak plasma control system that demonstrates an effect of our new scheme on a sampling period of plasma control. Furthermore, we provide microbenchmarking results to evaluate more generic properties of the I/O processing schemes By clarifying these capabilities, we aim to not only improve the overall performance but also broaden the scope of CPS that can benefit from the state-of-the-art GPU 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 proposes our zero-copy I/O processing scheme, and differentiates it from the exisiting schemes. Section 3.5 presents details of system implementation. In Section 4, 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 5. Section 6 introduces related work, and this paper concludes in Section **??**.

2. SYSTEM MODEL

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

The control algorithm is parallelized and offloaded to the GPU. We specifically 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 these data can be allocated to any place visible to I/O address space. According to the control system design, these data may or may not be further copied to other buffers. In either case, they are expressed as data *arrays* in the control algorithm.

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. Therefore, we ignore the problem of shared resources among multiple tasks. We also focus on a single instance of the GPU to implement the control algorithm; this is not a conceptual limitation of this paper, and the algorithm implementation could use multiple GPUs, as far as I/O address space is unified among them.

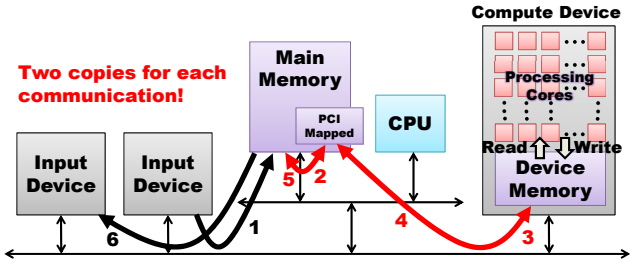

Figure 2: The traditional *H+D* scheme.

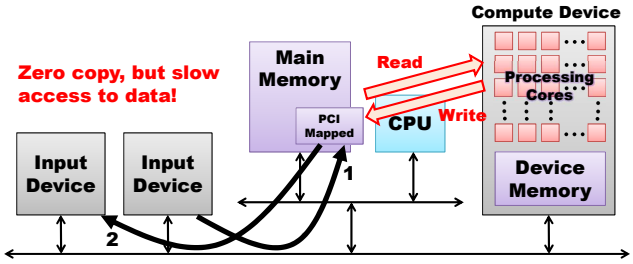

Figure 3: The zero-copy *Hpin* scheme.

3. I/O PROCESSING SCHEMES

This section presents a new zero-copy I/O processing scheme for GPU computing. This scheme differs from the existing schemes in that both the GPU execution and the data transfer times are minimized to meet a requirement of low-latency GPU computing. First of all, we introduce two existing schemes that are already supported by CUDA. We then present a new scheme, and also introduce a hybrid variant of our scheme and the existing one, which is suit 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 must exist in both the device and the host memory at once. Furthermore, there is a time penalty incurred to copy data, which is almost 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 deivce 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 desribed above, the *H+D* scheme incurs some 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 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 is a PCIe communication.

3.3 Device Memory Mapped to Host (*Dmap*)

We now present a new scheme called *Dmap* that overcomes the problems of the *H+D* and the *Hpin* schemes. The key to our scheme is having the PCIe base address register (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 appear on the corresponding allocated space of the device memory, seamlessly. This scheme, hence, does 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

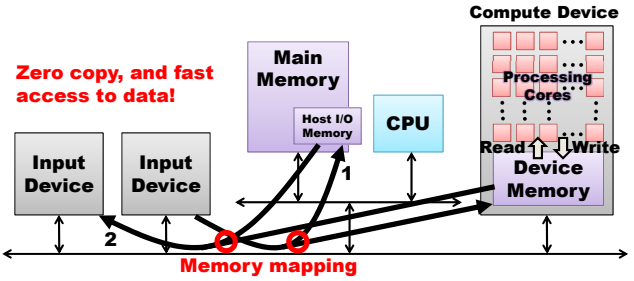

Figure 4: The zero-copy *Dmap* scheme.

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 provide at most 128MB for a single BAR, as of 2012. 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, particularly, 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 methods.

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 the *Dmap* scheme as far as memory allocation and mapping. For transfering 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 5.

3.5 System Implementation

GPU programs often execute in *virtual address space*. 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, applications of CPS often require I/O devices to be involved in 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 across the I/O device and the GPU. There is no available system implementation that enables 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 challenge is to cope with the limitation that the data access of I/O devices is limited to I/O address space. Considering that the GPU is typically connected to the system via the PCIe bus, we take an approach that maps 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. By this means, I/O devices can directly access data present 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 presented three functions correspondingly wrapped by the Gdev-original

extended API functions, `cuMemMap()`, `cuMemUnmap()`, and `cuMemGetPhysAddr()`, which are compatible to the form of CUDA Driver API. We now provide the details of these functions below:

- **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 specified 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 seek for physical pages allocated to this memory object. We next set up the page table of the GPU to have these physical pages be referenced by the virtual address space dedicated to the PCIe BAR. Now, these physical pages are referenced by two sorts of virtual address space. One is that dedicated to the PCIe BAR, and the other is associated with the user context. As a result, the data written to the PCIe BAR can be referenced by the user context, thus mapping is done. If the user context furhter requests to map it to the host memory, we 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. In case that the host memory is also mapped, we call a Linux kernel primitive, such as `iounmap()`.

- **Getting Physical Address:** Operating systems usually provide a function to return the physical addresses of PCIe BARs. In the Linux kernel, we can use `pci_resource_start()` for this purpose.

4. CASE STUDY

In this section, we provide a case study of magnetic control of perturbed plasma equilibria using the HBT-EP "Tokamak" equipped with the GPU and the presented 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. As mentioned in Section 1, the CPU implementation of this algorithm has never met the requirement of computing power. The case study presented herein, therefore, is significant in that we look into a possibility of GPU implementations for the plasma control system.

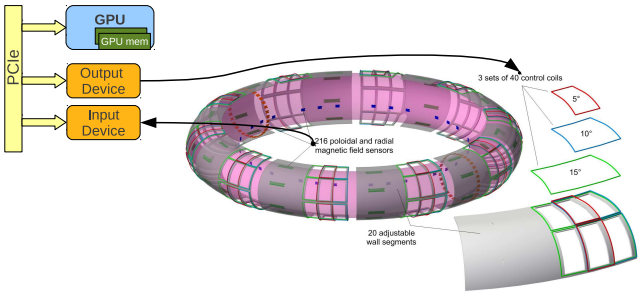

Figure 5: HBT-EP magnetic sensors and control coils connected with the GPU.

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 GPU upon the PCIe bus. We evaluate three different schemes against this system architecture from the viewpoint of the cycle time of algorithm execution and the latency of 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 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 proposed 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, since it is not within the scope of this paper. Interested readers are encouraged to reference [1]. The

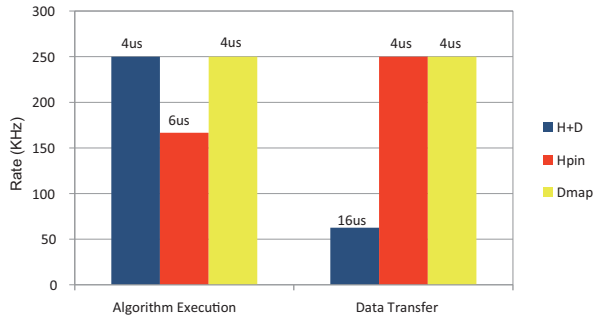

Figure 6: Cycle time and latency of the plasma control system.

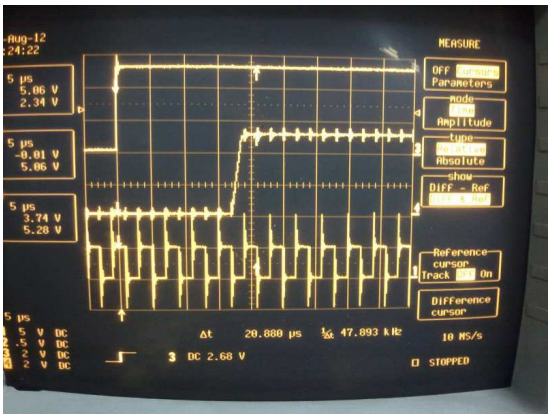

Figure 7: Screenshot of the oscilloscope.

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 is polling 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 proposed 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 demonstrate that the proposed *Dmap* scheme reduces both the cycle time of algorithm execution and the latency of data transfer in this control system. Figure 6 shows the result of experiments conducted under the three different schemes, respectively. Apparently, the *Dmap* scheme achieves the highest rate in both algorithm execution and data transfer. The remaining two schemes, on the other hand, compromise either of them. It is fairly reasonable to observe that the *H+D* and the *Dmap* schemes exhibit the same performance level for algorithm execution since they both use the device memory, while the latency of data transfer is equivalent between the *Hpin* and the *Dmap* schemes since they both remove data copies. Comparing the *H+D* and the *Hpin* schemes, one can also observe that the impact of overhead introduced by data copies, *i.e.*, 16μs, is greater than that introduced by the GPU accessing pinned host memory space, *i.e.*, 6μs, on the overall system performance. Curiously, there is additional latency of 4μs 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 16μs to 4μs. The speed-up ratio is 4×.
- Furthermore, the *Dmap* scheme proposed in this paper reduces the cycle time of algorithm execution from 6μs to 4μs. 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 sampling period of 16μs. The data transfer from and to the input and output modules takes 4μs each. The algorithm execution takes 4μs. Adding additional latency of 4μs, the total control rate must be able to achieve 16μs. 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 the output, respectively, while the lower signal indicates the base clock. The grid spacing of the X axis is 5μs. 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 16μs. This means that the total control processing time is 16μs.

We next demonstrate that the control system is running properly at a rate of 16μs. 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

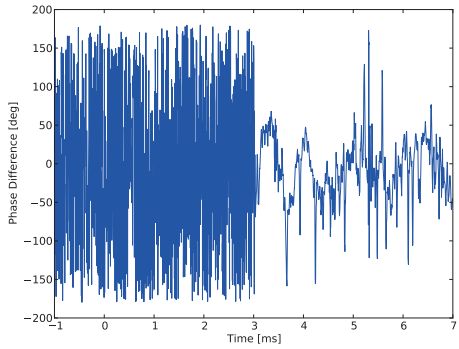

Figure 8: Phase difference observed with the base output signal.

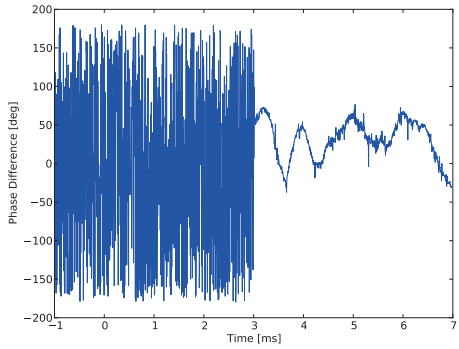

Figure 9: Phase difference observed with the output signal shifted by 16μs.

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 16μs. In other words, the effective control system latency is reduced by 16μs. 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 16μs.

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 phasings. 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 supress the strength of the rotating field by up to 30% for any mode observed in this experiment.

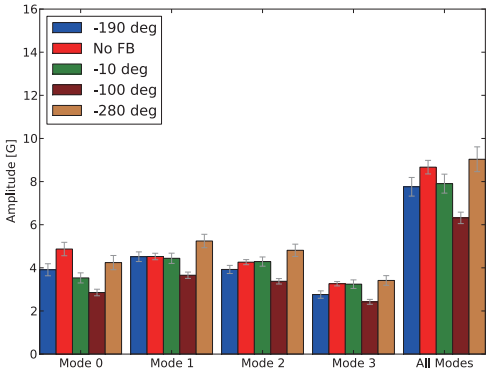

Figure 10: Practical findings of plasma control.

5. BENCHMARKING

To evaluate our system we compared it with the vendor provided methods mentioned in section **??**. Timing analysis of addition and multiplication of varying sized matrices was performed (two common benchmarks also used in related work [18]). Since our focus is on data access and communication (not computation) we chose matrix addition as a benchmark as it is a straightforward operation for a GPU to perform. Matrix multiplication is also included to briefly illustrate how increasing computational complexity and data accesses affect time to completion. Effective host read and write throughput for each method was also analyzed.

5.1 Matrix Operations

Figure 11 shows where the system spends its time in performing a 2048x2048 integer matrix addition using each of the four methods. The time categories are as follows:

- **Init:** GPU initialization time.
- **MemAlloc:** Memory allocation time (host and/or GPU).
- **DataInit:** Time to initialize the matrices.
- **HtoD:** Copy time from host to device (H+D).
- **Exec:** Execution time of the kernel function.
- **DtoH:** Copy time from device to host (H+D and $DmapH$).
- **DataRead:** Time to read the result.

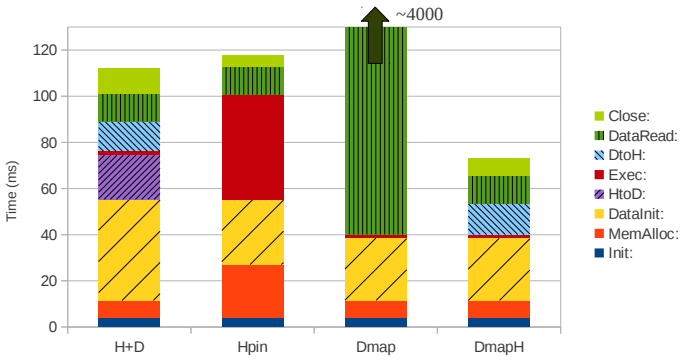

Figure 11: Matrix addition

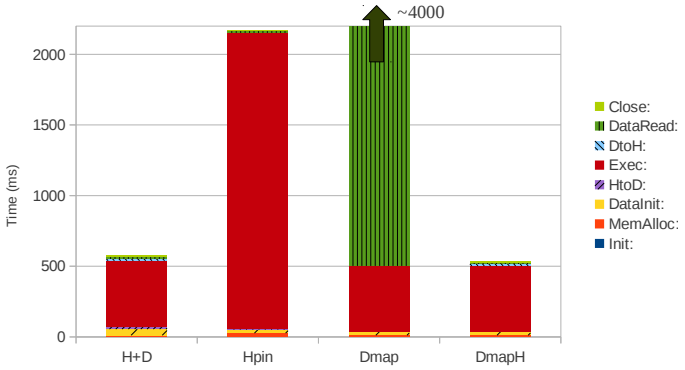

Figure 12: Matrix multiplication

- **Close:** Free memory and close GPU.

First, looking at figure 11 it is clear that using our *DmapH* method the total time to completion is less than the others (34% less than its nearest competitor, *H+D*). More interesting is the comparison of how much time is spent for each sub-task.

For most time categories *DmapH* seems to enjoy the best of each of the other three: The memory allocation time for *DmapH* is nearly identical to that of *H+D* and *Dmap* and clearly less than *Hpin*. The same is true for the execution times. Similarly, for data initialization, *DmapH* is just as good as *Hpin* and *Dmap* which are superior to *H+D*.

The most notable difference among the four methods is the data read time for *Dmap* which is orders of magnitude greater than the rest. This represents 16MB of data, read 4 bytes at a time across the system buses and is the motivation for our *DmapH* method. Instead of reading one matrix element at a time, *DmapH* first copies the data to the host before reading.

The same anomaly is present in the execution time for *Hpin* – the GPU must read one element at a time from the host memory (in the other three methods the data reside on the GPU during computation). This makes

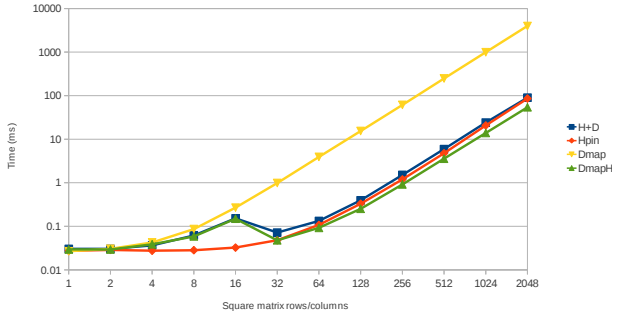

Figure 13: Total Matrix Addition Times

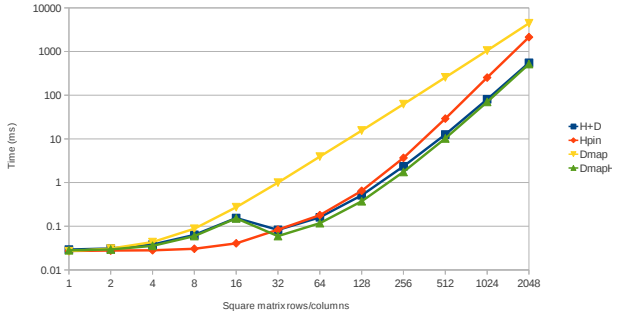

Figure 14: Total Matrix Multiplication Times

Hpin increasingly inferior as data sizes grow.

Figure 12 shows the same time analysis for matrix multiplication. The only difference occurs in the execution times. This is not surprising as the multiplication experiments are exactly the same as addition with the exception of the kernel function on the GPU. In fact, if execution times were omitted, figures 11 and 12 would look identical.

While this does present a real-world scenario, in a real-time system it is more likely that tasks will be performed repeatedly and therefore the GPU initialization and closing costs might occur only once – the same context with kernel function(s) loaded would remain active while only the data might change. Furthermore, 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 reads, writes, transfers, and execution.

Figure 13 shows the time to completion of matrix addition as a function of matrix size (note the logarithmic scale). *Hpin* appears to be the best performer until a matrix size of 32×32 (corresponding to a data size of 4KB for each matrix) at which point *DmapH* becomes superior. This is also reflected in the matrix multiplication times (figure 14). One thing to note is the growth rate of *Hpin* in matrix multiplication; Since each thread must perform multiplication and addition

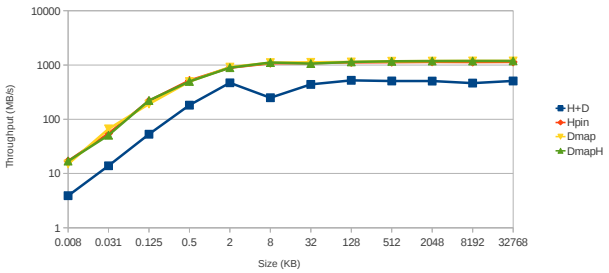

Figure 15: Host Write Throughput

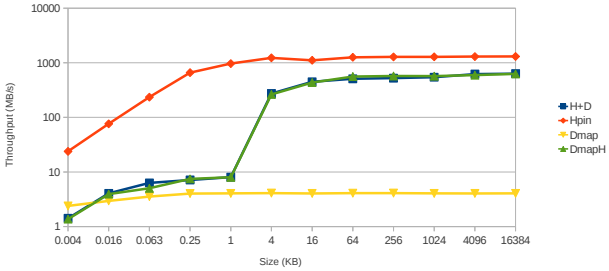

Figure 16: Host Read Throughput

n^2 times (compared with one addition in matrix addition), the number of reads that occur across the buses increases by a greater exponential factor. We expect that the time for *Hpin* would eventually surpass *Dmap* as the trend in the graph indicates.

5.2 Data Throughput

Figure 15 shows the effective host write throughput as a function of data size. 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 *H+D* this corresponds to the total time for the host to write to each element in the data structure (which is in main memory) plus the time to copy the data to the GPU.

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

6. RELATED WORK

NVIDIA's *Fermi* architecture[15] supports unified virtual addressing (UVA) which creates a single address space for host memory and multiple GPUs. This allows their `cuMemcpyPeer()` function to copy data directly between GPUs over the PCIe bus without the involvement of the host CPU or memory. It is, of course, restricted for use between NVIDIA GPUs – not arbitrary I/O devices.

An automatic system for managing and optimizing CPU-GPU communication has been developed called CGCM[**?**]. It uses compiler modifications in conjunction with a runtime library to manipulate the timing of events in a way that effectively reduces transfer overhead. By analyzing source code, they are able to identify operations that can be temporally promoted while still preserving dependencies. This is made possible in part by taking advantage of the ability to concurrently copy data and execute kernel functions.

PTask[18] is a project that provides GPU programming abstractions that are supported by the OS. One aspect of the project is the use of a data-flow programming model to minimize data communication between the host and GPU.

High priority compute tasks may have high blocking times imposed on them due to data transfers. RGEM[6] is a project that aims to bound these blocking times by dividing data into chunks. 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.

7. REFERENCES

[1] A. Boozer. Perturbed plasma equilibria. *Physics of Plasma*, 6, 1999.

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

[3] 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.

[4] 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 (to appear).

[5] S. Kato, K. Lakshmanan, Y. Ishikawa, and R. Rajkumar. Resource Sharing in GPU-accelerated Windowing Systems. In *Proceedings 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 *Proceedings 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 *Proceedings 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 (Whitepaper), 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 Compute Architecture: Fermi (Whitepaper). `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`.

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

[17] 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.

[18] 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.