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.

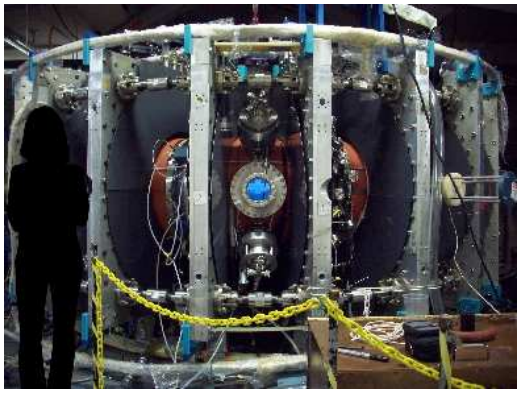

Figure 1: The HBT-EP "Tokamak" at Columbia University.

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 sampling 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 [15] – 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.

In order to utilize the GPU for applications of CPS, the system is required to support a method of bypassing data transfer between the GPU and the CPU, 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 presents our zero-copy I/O processing scheme, and differentiates it from the exisiting schemes.

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 in our system 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.

3. I/O PROCESSING SCHEMES

In this work we focus on three primary methods of data communication (illustrated in figure 2). The first two are provided by the CUDA API, and the third is our own extension of the API. We also introduce a fourth which is a hybrid of ours and an existing method.

3.1 Host Memory and Device Memory (H+D)

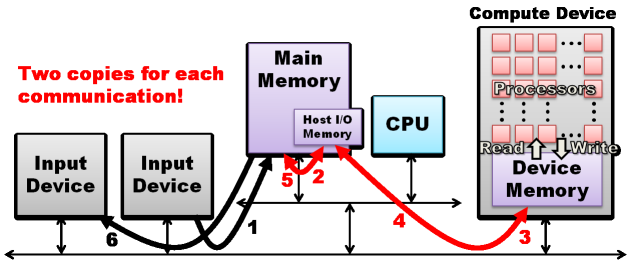

Figure 2: Data communication using different memory allocation methods

The most common and straightforward model for GPGPU computing is as follows:

1. Memory is allocated on the host and is initialized with the input data.
2. Sufficient memory is also allocated on the GPU for the input data as well as any space needed for output data.
3. The host initiates a memory copy from its main memory to the GPU.
4. The kernel function is launched on the GPU.
5. When computation is complete, the host initiates a copy from the GPU back to the host to retrieve the result.
6. Memory is de-allocated on the GPU and host.

In this model there is space overhead in that the input and output data must exist in two places at once. Furthermore, there is a time penalty incurred for copying data that is proportional to its size.

3.2 Host Pinned Memory (Hpin)

An alternative to allocating memory on both the host and GPU, the device can allocate page-locked memory (also known as 'pinned' memory) on the host. It is mapped into the address space of the device and can be referenced directly by the GPU code. Since this memory is not page-able it is always resident in the host RAM. It does, however, reduce the amount of available physical memory on the host which can adversely affect system performance. Using this model data can be written directly into this space with no need for an intermediate host buffer or copying[15].

3.3 Device Memory Mapped to Host (Dmap)

The key to our method is having the host point directly to data on the GPU. This simplifies the programming paradigm in that no explicit copying needs to take place – the data can be referenced directly by the host. This means that input data can be written directly to the device without the need for intermediate buffers while the GPU maintains the performance benefit of having the data on board.

This model is not limited to mapping GPU memory space to the host, it is actually mapped to the PCI address space for the GPU which enables other devices to access it. This is how direct communication between the GPU and I/O devices is achieved.

3.4 Device Memory Mapped Hybrid (DmapH)

This method is the same as *Dmap* as far as memory allocation and mapping. Like *Dmap* the host references the GPU memory directly when writing data. For reading, however, we perform an explicit copy from GPU to host. The motivation for this is explained in our evaluation in section **??**.

Although this memory mapping ability is used by NVIDIA's proprietary drivers it is not accessible to the programmer via the publicly available APIs (CUDA and OpenCL). To use this functionality our system requires the use of an open-source device driver such as nouveau or PathScale's pscnv[16].

3.5 System Implementation

GPGPU programs typically access the *virtual address space* of GPU device memory to allocate and manage data. In CUDA programs, for instance, a device pointer acquired through memory allocation functions such as `cuMemAlloc()` and `cuMemAllocHost()` represents the virtual address. Relevant data-copy functions such as `cuMemcpyHtoD()` and `cuMemcpyDtoH()` also use these pointers to virtual addresses instead of those to physical addresses. As long as the programs remain within the GPU and CPU this is a suitable addressing model. However, embedded systems often require I/O devices to be involved in sensing and actuation. The existing API designs for GPGPU programming force these embedded systems to use main memory as an intermediate buffer to bridge across the I/O device and GPU – there are no available system implementations that enable data to be transfered directly between the I/O device and GPU.

In this section we present our system implementation of the *Dmap* and *DmapH* methods using Gdev [**?**]. A key challenge in the implementation is to overcome the limitation that I/O devices are accessible to only the I/O address space. Given that the GPU is typically connected to the system via the PCI bus, we take an approach that maps the virtual address space of GPU device memory to the I/O address space of the PCI bus. By this means, an I/O device can directly access data present in the GPU device memory. More specifically, the device driver can configure the DMA engine of the I/O device to target the PCI bus address associated

with the mapped space of GPU device memory.

Our system implementation adds three API functions to CUDA: (i) `cuMemMap()`, (ii) `cuMemUnmap()`, and (iii) `cuMemGetPhysAddr()`. The `cuMemMap()` and `cuMemUnmap()` functions are introduced to map and un-map the virtual address space of GPU device memory to and from the user buffer allocated in main memory. These functions are somewhat complicated since main memory and GPU device memory cannot share the same virtual address space. We must first map the virtual address space of GPU device memory to one of the PCI I/O regions specified by the base address registers (BARs), and next remap this PCI BAR space to the user buffer. When using host pinned memory, on the other hand, we simply allocate I/O memory pages, also known as DMA pages in the Linux kernel, and map the allocated pages to the user buffer. The Linux kernel, for instance, supports `ioremap()` for the former case and `mmap()` for the latter case. This paper is focused on the latter case of GPU device memory mapping, but the concept of our method described below is also applicable to hot memory mapping. This is because recent GPUs support unified addressing which allows the same virtual address space to describe both GPU device memory and host pinned memory.

An I/O device driver may use the memory-mapped user buffer directly. For example, if the size of I/O data is small enough, the device driver can simply read and write data using this user buffer. If the system deals with a large size of data, however, we need to obtain the physical address of the PCI BAR space corresponding to the target space of GPU device memory so that the device driver can configure the DMA engine of the I/O device to transfer data in burst mode to/from the PCI bus. We provide the `cuMemGetPhysAddr()` function for this purpose. This function itself simply returns the physical address of the PCI BAR space associated with the requested target space of GPU device memory. Note that we must make the PCI BAR space contiguous with respect to the data set transferred by the DMA engine. The DMA transfer would fail otherwise.

There is an exception in the case where the size of I/O data to be mapped and transferred is greater than the maximum size of the PCI BAR space. For instance, NVIDIA *Fermi* GPUs limit the PCI BAR space to be no greater than 128MB. This is the current limitation of our implementation.

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.

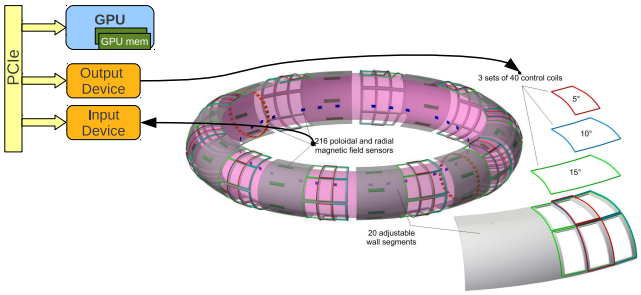

Figure 3: HBT-EP magnetic sensors and control coils connected with the GPU.

Figure 3 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

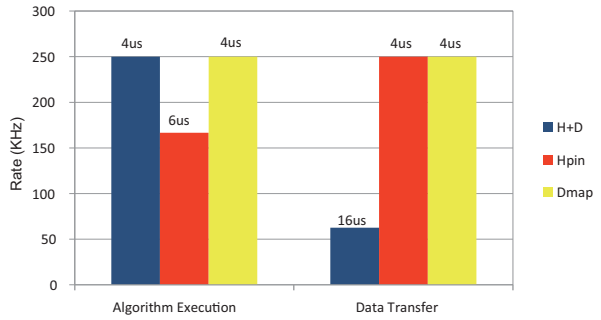

Figure 4: Cycle time and latency of the plasma control system.

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.

We now demonstrate that our zero-copy I/O processing scheme reduces both the cycle time of algorithm execution and the latency of data transfer in this control system. Figure 4 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.

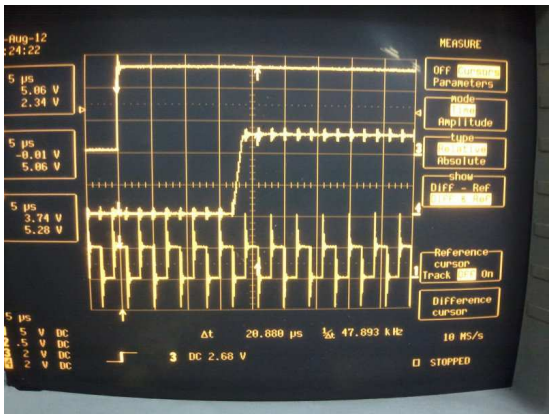

Figure 5: Screenshot of the oscilloscope.

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 5 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 3, 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 6. Now, we measure the phase difference with the output signal time-shifted by 16μs. In other words, the effective control system

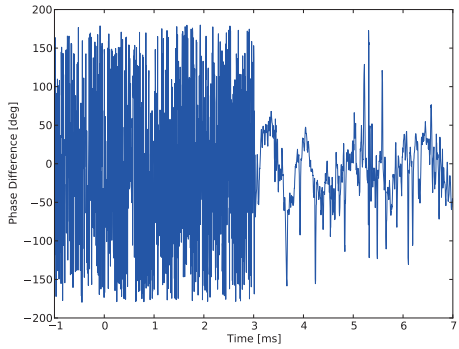

Figure 6: Phase difference observed with the base output signal.

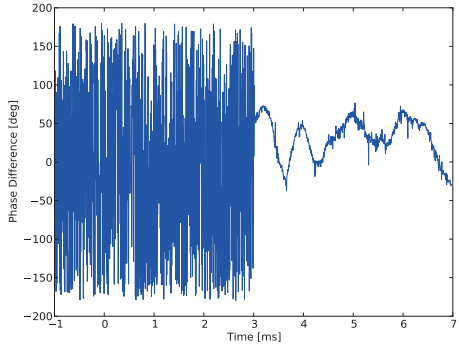

Figure 7: Phase difference observed with the output signal shifted by 16μs.

latency is reduced by 16μs. As shown in Figure 7, 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 8 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.

5. BENCHMARKING

To evaluate our system we compared it with the vendor provided methods mentioned in section **??**. Timing

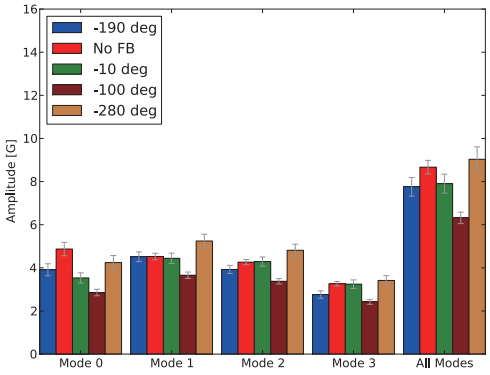

Figure 8: Practical findings of plasma control.

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 9 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.
- **Close:** Free memory and close GPU.

First, looking at figure 9 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.

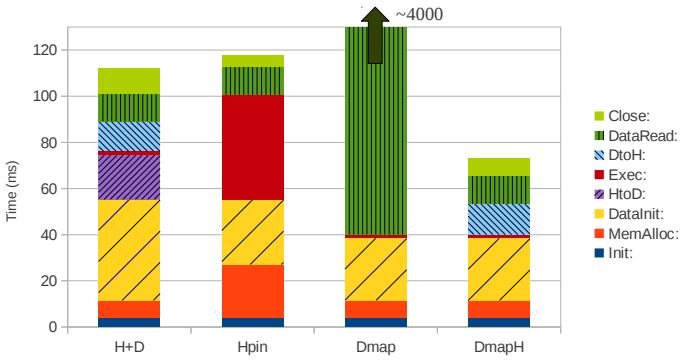

Figure 9: Matrix addition

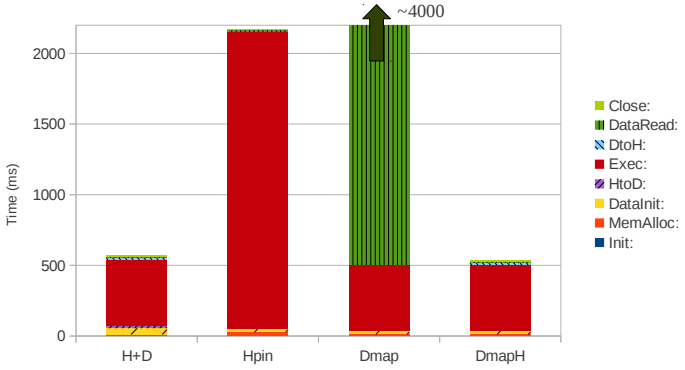

Figure 10: Matrix multiplication

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

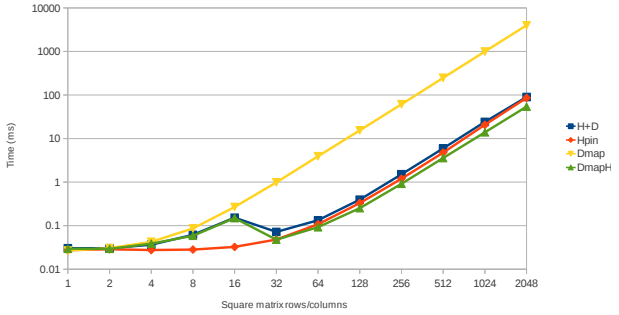

Figure 11: Total Matrix Addition Times

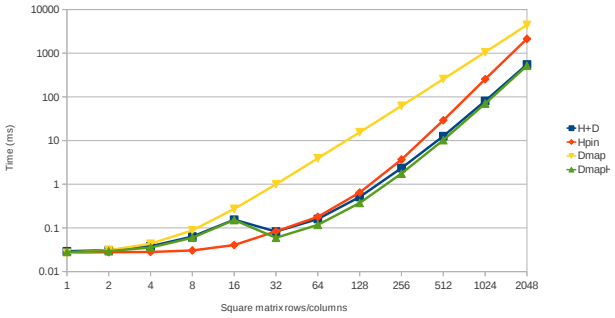

Figure 12: Total Matrix Multiplication Times

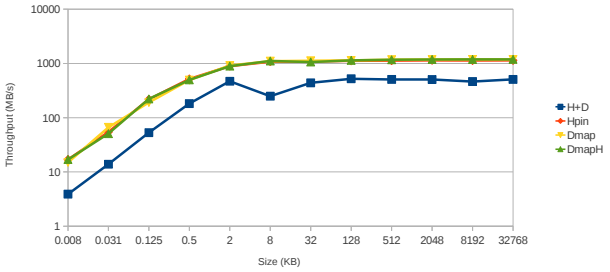

Figure 13: Host Write Throughput

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 *Hpin* increasingly inferior as data sizes grow.

Figure 10 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 9 and 10 would look identical.

While this does present a real-world scenario, in a

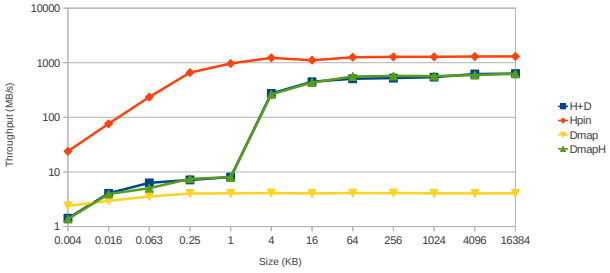

Figure 14: Host Read Throughput

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 11 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 12). One thing to note is the growth rate of *Hpin* in matrix multiplication; Since each thread must perform multiplication and addition 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 13 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 13 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 14 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. 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. CUDA C Programming Guide Version 4.2, 2012.

[16] PathScale. ENZO. `http://www.pathscale.com/`.

[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.