Supporting Low-Latency CPS using GPUs and Direct I/O Schemes

Jason Aumiller, Scott Brandt
Department of Computer Science
University of California
Santa Cruz, United States
{*jaumille, scott*}*@soe.ucsc.edu*

Shinpei Kato
Department of Information Science
Nagoya University
Nagoya, Japan
shinpei@is.nagoya-u.ac.jp

Nikolaus Rath
Department of Applied Physics and
Applied Mathematics
Columbia University
New York, United States
nikolaus@rath.org

***Abstract*—Graphics processing units (GPUs) are increasingly being used for general purpose parallel computing. They provide significant performance gains over multi-core CPU systems, and are an easily accessible alternative to supercomputers. The architecture of general purpose GPU systems (GPGPU), however, poses challenges in efficiently transferring data among the host and device(s). Although commodity many-core devices such as NVIDIA GPUs provide more than one way to move data around, it is unclear which method is most effective given a particular application. This presents difficulty in supporting latency-sensitive cyber-physical systems (CPS).**

In this work we present a new approach to data transfer in a heterogeneous computing system that allows direct communication between GPUs and other I/O devices. In addition to adding this functionality our system also improves communication between the GPU and host. We analyze the current vendor provided data communication mechanisms and identify which methods work best for particular tasks with respect to throughput, and total time to completion.

Our method allows a new class of real-time cyber-physical applications to be implemented on a GPGPU system. The results of the experiments presented here show that GPU tasks can be completed in 34 percent less time than current methods. Furthermore, effective data throughput is at least as good as the current best performers. This work is part of concurrent development of Gdev [6], an open-source project to provide Linux operating system support of many-core device resource management.

***Keywords*-GPGPU; real time systems; GPU communication;**

I. Introduction

It has been established by the scientific computing community that the use of graphics processing units (GPUs) for parallel processing can yield great performance gains over single and multi-core CPU systems alone. Certain tasks such as software routing, and network encryption can achieve more than an order of magnitude performance gain [2], [4]. Distributed storage systems that employ hash based primitives enjoy a speed up of 8x [1].

Not only an affordable alternative to supercomputers, GPUs are actually found in many supercomputers today [13]. One particular weather modeling application running on such a system reports an 80x speedup achieving 15 TFlops [12].

Until very recently accomplishments such as these have been made possible only by application specific, low-level programming of the host and GPU. Resource management must be performed by the programmer and with the target hardware platform in mind. GPU manufacturers have responded to this by implementing new tools in their APIs to support general purpose computing, however, such support is still quite limited and solutions for new applications typically need to be hand crafted. Recent works to integrate GPU programming abstractions into the operating system have underscored the importance of such support [6], [11].

One key aspect of such efforts is data communication between the host and GPU. Since a GPU has its own memory that is separate from the system main memory, data must be transferred between the two. The overhead incurred to do this can, in some cases, exceed the savings in compute time and must be considered in deciding which applications will benefit.

Currently, to move data between any device and a GPU, it must first pass through the host. In a CPS with an I/O module, for instance, the host code would be responsible for acquiring input from the module, storing the data in a buffer and then copying it to the GPU for processing. This overhead presents a significant problem for real time tasks that require extremely low-latencies. One example is a tokamak feedback control system for plasma fusion which requires matrix operations in the microsecond range [10].

Our system allows direct communication between GPUs and I/O devices, completely bypassing the host CPU and memory. Besides eliminating the raw transfer overhead, this also exempts GPU tasks from latencies due to the host servicing the operating system.

To the best of our knowledge, there is currently no support for direct communication between GPUs and I/O devices. There are multiple GPU-host data communication methods available from current programming frameworks, but it is unclear which is optimal given a particular application. With the addition of our approach we answer this question with respect to a couple of benchmarks and data throughput tests. By making data communication more efficient we aim to not only improve the overall performance but to also broaden the scope of CPS that can benefit from the use of GPUs.

The rest of this document is arranged as follows: Section II describes briefly the aspects of the hardware architecture

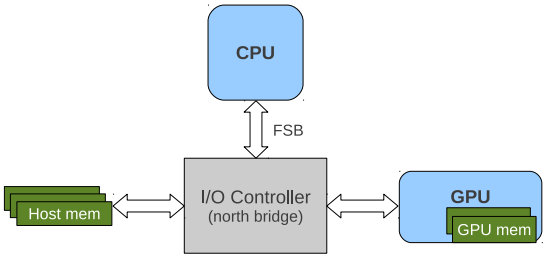

Figure 1. Typical system architecture

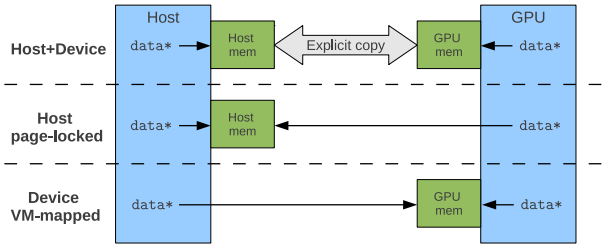

Figure 2. Data communication using different memory allocation methods

of a typical CPU/GPU system that motivate this work. Section III describes the current mechanisms used for data communication and contrasts them with ours. Section IV gives an explanation of the architecture of our system and how it interacts with the hardware and the OS. We evaluate our system with respect to the existing techniques in section V and discuss related work and our conclusions in sections VI and VII.

In this paper the terms GPU, device, and many-core device are used interchangeably. For readers unfamiliar with GPU terminology 'kernel function' refers to a function that is executed on the GPU – it is not related to the operating system kernel.

II. System Model

Since the processing of graphics is inherently parallel GPUs are designed with many processing units. Graphics processing algorithms are typically comprised of simple arithmetic operations that are easily performed by RISC processors. Since the same operations need to be performed on many different pieces of data GPUs employ a single instruction, multiple data (SIMD) architecture. Instead of each processor maintaining its own instruction register, on a GPU groups of processing units all share the same execution control and perform in lock-step. These factors allow GPUs to use processing units that are far less complicated than modern CPUs and therefore provide a powerful yet cost-effective means of parallel processing.

Aside from data communication hurdles, GPUs are at a distinct advantage for use in real-time systems because they are dedicated to compute tasks only. A single- or multi-core CPU system needs to schedule operating system processes periodically regardless of the priority level of any other tasks. GPUs, on the other hand, have no OS. Besides an overall improvement in performance, a possibly more valuable feature that this affords is *consistency* of performance – time to completion can be estimated with higher precision and achieved with higher regularity.

Figure 1 shows the architecture of a typical CPU+GPU system and is representative of our test platform. The CPU interacts with an I/O controller (typically referred to as a 'north bridge') which controls data flow among CPU(s), RAM, and peripheral devices. Our test platform consists of a dual-core processor and an NVIDIA GPU on the PCI-express bus. Data copies are initiated by the CPU and the north bridge facilitates the transfer of data between the host memory and GPU memory.

III. Data Communication Methods

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.

A. Host Memory and Device Memory (H+D)

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.

B. 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 [7].

C. 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.

D. 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 V.

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 [9].

IV. 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 [6]. 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 unmap 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.

V. Evaluation

To evaluate our system we compared it with the vendor provided methods mentioned in section III. Timing analysis of addition and multiplication of varying sized matrices was performed (two common benchmarks also used in

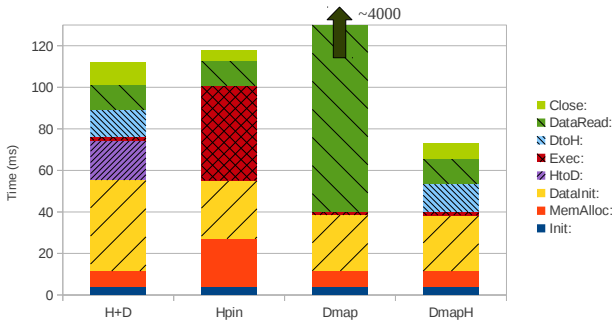

Figure 3. Matrix addition

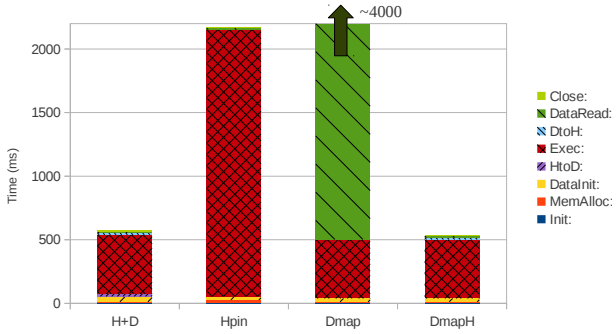

Figure 4. Matrix multiplication

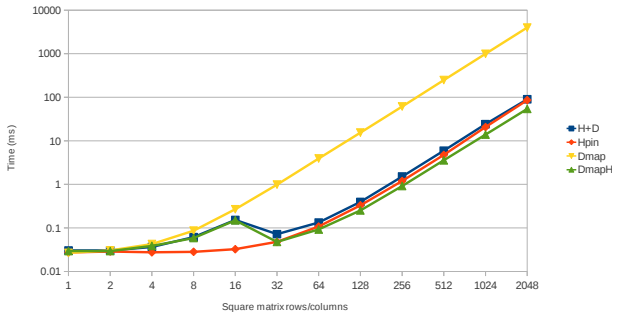

Figure 5. Total Matrix Addition Times

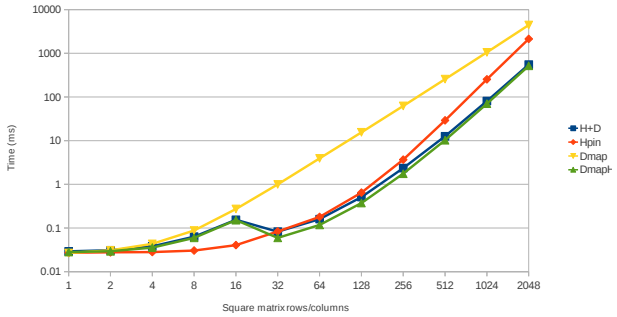

Figure 6. Total Matrix Multiplication Times

related work [11]). 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.

A. Matrix Operations

Figure 3 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 3 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 *Hpin* increasingly inferior as data sizes grow.

Figure 4 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 3 and 4 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

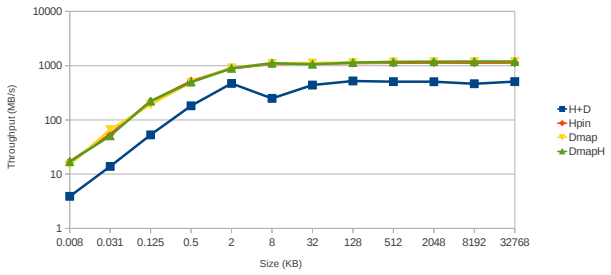

Figure 7. Host Write Throughput

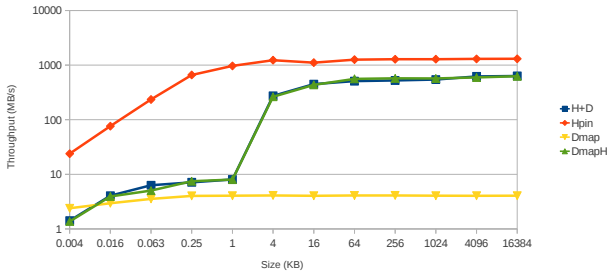

Figure 8. Host Read Throughput

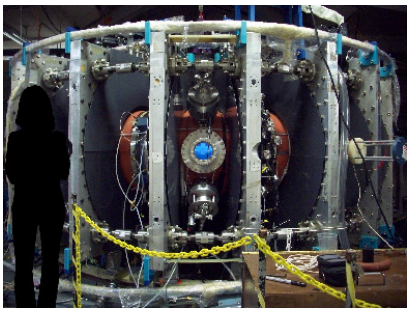

Figure 9. HBT-EP Tokamak at Columbia University

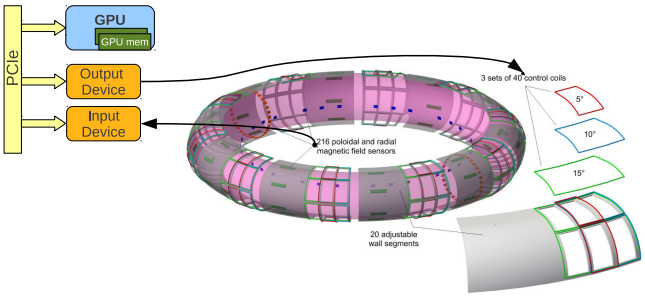

Figure 10. HBT-EP control system architecture

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 5 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 6). 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.

B. Data Throughput

Figure 7 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 7 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 8 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).

C. Tokamak Control System

Used in the field of thermonuclear fusion research, a tokamak is a device that uses magnetic fields to contain plasma. Our method is currently being integrated into a feedback control system for the High-Beta Tokamak (HBT-EP) at the University of Columbia [10]. A CPU+GPU system has been developed in which the input and output modules for the control system reside on the PCIe bus with the GPU (see figure 10). The direct communication between the I/O devices and the GPU keeps the CPU and host memory out of the control cycle. Preliminary testing has shown that a cycle frequency of 200kHz can be maintained which corresponds to a matrix operation every 5μs.

VI. Related Work

NVIDIA's *Fermi* architecture [8] 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 [3]. 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 [11] 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 [5] 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.

VII. Conclusion

The data communication method we have presented in this paper allows particular GPU compute tasks to complete significantly faster than with currently available methods. For data sizes 4KB and above the benefit of using *DmapH* increases.

With respect to data write throughput we have shown that *DmapH* is at least as good as the current best method (*Hpin*). In cases where *Hpin* cannot be used (due to host memory limitations) *DmapH* is also as good as the best performer when it comes to read throughput.

Most notable is the fact that our method is the first to support direct data transfer between GPUs and arbitrary I/O devices. We have accomplished this by exposing existing capabilities of the GPU that are not currently available via the vendor provided APIs. This functionality should prove to be invaluable for real-time CPS that require low-latency.

The results presented here illustrate how the performance of different methods are impacted by the overhead of communicating across system buses. While our experiments only evaluate communication between a GPU and the host we expect that the *relative* performance of these methods are maintained in a GPU to I/O device scenario.

Acknowledgment

We thank members of the HBT-EP Team for providing us with knowledge of plasma and fusion use cases. This work is also supported in part by U.S. Department of Energy (DOE) Grant DE-FG02-86ER53222. We also thank PathScale Inc. for providing us with their open-source GPU driver.

References

[1] S. Al-Kiswany, A. Gharaibeh, E. Santos-Neto, G. Yuan, and M. Ripeanu. StoreGPU: Exploiting Graphics Processing Units to Accelerate Distributed Storage Systems. In *Proceedings of the ACM International Symposium on High Performance Distributed Computing*, pages 165–174, 2008.

[2] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. Packetshader: a gpu-accelerated software router. *SIGCOMM Comput. Commun. Rev.*, 40(4):195–206, August 2010.

[3] Thomas B. Jablin, Prakash Prabhu, James A. Jablin, Nick P. Johnson, Stephen R. Beard, and David I. August. Automatic cpu-gpu communication management and optimization. *SIGPLAN Not.*, 46(6):142–151, June 2011.

[4] Keon Jang, Sangjin Han, Seungyeop Han, Sue Moon, and KyoungSoo Park. Sslshader: cheap ssl acceleration with commodity processors. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, pages 1–1, Berkeley, CA, USA, 2011. USENIX Association.

[5] 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.

[6] Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Scott Brandt. Gdev: First-class gpu resource management in the operating system. In *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC'12)*, 2012 (in press).

[7] NVIDIA. CUDA C Programming Guide. http://developer.nvidia.com/nvidia-gpu-computing-documentation.

[8] 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.

[9] PathScale. ENZO. http://www.pathscale.com/.

[10] N. Rath, J. Bialek, P.J. Byrne, B. DeBono, J.P. Levesque, B. Li, M.E. Mauel, D.A. Maurer, G.A. Navratil, and D. Shiraki. High-speed, multi-input, multi-output control using gpu processing in the hbt-ep tokamak. *Fusion Engineering and Design*, (0):–, 2012.

[11] 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.

[12] T. Shimokawabe, T. Aoki, C. Muroi, J. Ishida, K. Kawano, T. Endo, A. Nukada, N. Maruyama, and S. Matsuoka. An 80-Fold Speedup, 15.0 TFlops, Full GPU Acceleration of Non-Hydrostatic Weather Model ASUCA Production Code. In *Proceedings of the ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis*, 2010.

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