Operating Systems Challenges for GPU Resource Management [*]

Shinpei Kato and Scott Brandt
University of California, Santa Cruz

Yutaka Ishikawa
University of Tokyo

Ragunathan (Raj) Rajkumar
Carnegie Mellon University

Abstract

The graphics processing unit (GPU) is becoming a very powerful platform to accelerate graphics and data-parallel compute-intensive applications. It significantly outperforms traditional multi-core processors in performance and energy efficiency. Its application domains also range widely from embedded systems to high-performance computing systems. However, operating systems support is not adequate, lacking models, designs, and implementation efforts of GPU resource management for multi-tasking environments.

This paper identifies a GPU resource management model to provide a basis for operating systems research using GPU technology. In particular, we present design concepts for GPU resource management. A list of operating systems challenges is also provided to highlight future directions of this research domain, including specific ideas of GPU scheduling for real-time systems. Our preliminary evaluation demonstrates that the performance of open-source software is competitive with that of proprietary software, and hence operating systems research can start investigating GPU resource management.

1 Introduction

Performance and energy are major concerns for today's computer systems. In the early 2000s, chip manufacturers had competed on processor clock rate to continue performance improvements in their product lines. For instance, the Intel Pentium 4 processor was the first commercial product that exceeded a clock rate of 3 GHz in 2002. This performance race on clock rate, however, came to end due to power and heat problems that prevent the chip design from increasing clock rate in classical single-core technology. Since the late 2000s, performance improvements have continued to come through innovations in multi-core technology, rather than clock-rate increases. This paradigm shift was a breakthrough to achieve high-performance with low-energy. Today, we are getting into the "many-core" era, in order to meet the further performance requirements of emerging data-parallel and compute-intensive applications. Not only high-performance computing (HPC) applications but also embedded applications, such as autonomous vehicles [39, 41] and robots [26], benefit from the power of many-core processors to process a large amount of data obtained from their operating environments.

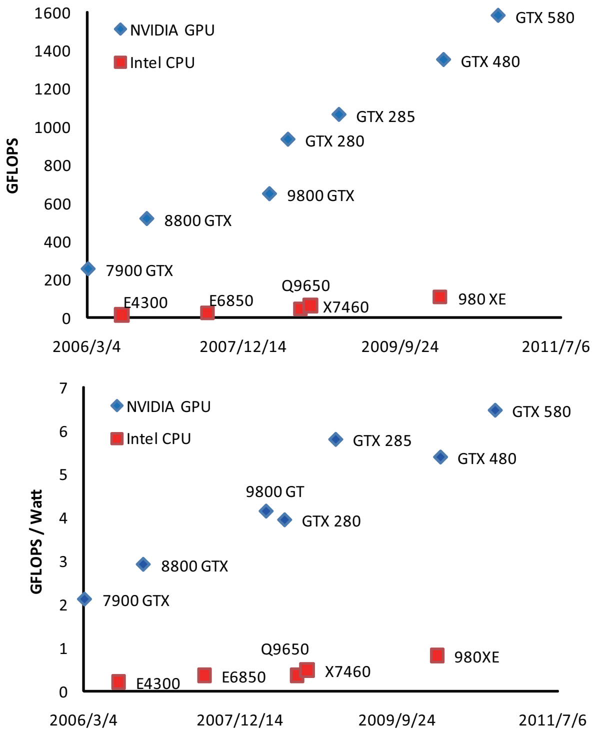

Figure 1. Performance trends on the well-known GPU and CPU architectures.

The graphics processing unit (GPU) has become one of the most powerful platforms embracing the concept of many-core processors. Figure 1 illustrates recent performance trends on the well-known GPU and CPU architectures from NVIDIA and Intel. The single-chip peak performance of the state-of-the-art GPU architecture exceeds 1500 GFLOPS, whereas that of a traditional microprocessor is around 100 GFLOPS at best. This non-trivial performance advantage of the GPU comes from hundreds of processing cores integrated on a chip. The GPU is also more preferable than the CPU in performance per watt. Specifically, the GPU is about 7 times more energy-efficient than the CPU today.

A recent announcement from the TOP500 supercomputing sites disclosed [40] that three of the top five supercomputers comprise GPU clusters, and significant performance improvements are provided by such GPU clusters for scientific applications [38]. Large-scale storage systems also benefit from the GPU [1, 9, 19]. In fact, the Amazon EC2 cloud computing ser-

[*] This work is supported by the fund of Research Fellowships of the Japan Society for the Promotion of Science for Young Scientists.

vice leverages GPU clusters to build their data centers. In the embedded systems domain, a new version of Carnegie Mellon's autonomous vehicle [41] equips four NVIDIA's GPUs to enhance its computing power required for autonomous driving tasks, including vision-based perception and motion planning. A case study from Stanford [39] revealed that the GPU can speed up computer vision applications for autonomous driving by 40 times compared to CPU execution. Such a rapid growth of general-purpose computing on GPUs, also known as *GPGPU*, is supported by recent advances in programming technology enabling the GPU to be used easily for general "compute" problems.

Despite the success of GPU technology, operating systems support in commodity software [6, 28, 33] is very limited for GPU resource management. Multi-tasking concepts, such as fairness, prioritization, and isolation, are not supported at all. The research community has developed several approaches to GPU resource management recently. In particular, notable contributions include TimeGraph [17] providing capabilities of prioritization and isolation, and GERM [2] with fairness support, for multi-tasking GPU applications. Despite the very limited information of GPU hardware details available to the public, these studies made efforts to develop GPU resource management primitives at the device-driver level. However, their functionality is limited to specific workloads. There are also other research projects on GPU resource management provided on the layers above the device driver, including CPU schedulers [8], virtual machine monitors [7, 11, 12, 20], and user-space programs [3, 10, 36], but their basic performance and capabilities are limited to underlying commodity software. We believe that operating systems research must explore and address GPU resource management problems to enable GPU technology in multiple application domains.

This paper identifies several directions towards operating systems challenges for GPU resource management. Currently, we lack even a fundamental GPU resource management model that could underlie prospective research efforts. The missing information of open-source software is particularly a critical issue to explore systems design and implementation of GPU resource management. In this paper, we present initial ideas and potential solutions to these open problems. We also demonstrate that open-source software is now ready to be used reliably for research.

The rest of this paper is organized as follows. Assumptions behind this paper are described in Section 2. Section 3 presents the state-of-the-art GPU programming model, and Section 4 introduces a basic GPU resource management model for the operating system. Section 5 provides operating systems challenges for GPU resource management, including a preliminary evaluation of existing open-source software. The concluding remarks of this paper is presented in Section 6.

2 Our System Assumptions

This paper considers heterogeneous systems composed of multi-core CPUs and GPUs. Several GPU architectures exist today. While many traditional microprocessors designed based on the X86 CPU architecture compatible across many generations over decades, GPU architectures tend to change in years. NVIDIA has released the Fermi architecture [30] as of 2011, supporting both compute and graphics programs. This paper focuses on the Fermi architecture, but the concept is also applicable to other architectures. We also assume an *on-board* GPU. Although Intel provides a new X86-based architecture, called Sandy Bridge [13], which integrates the GPU on a chip, GPUs on a board are still more popular today. In future work, however, we will study implications of on-board and on-chip GPUs from the operating systems point of view.

Given an on-board GPU model, we assume that the CPU and the GPU operate asynchronously. In other words, CPU contexts and GPU contexts are separately processed. Once user programs launch a piece of code onto the GPU to get accelerated, it is offloaded from the CPU. The user programs may continue to execute on the CPU while this piece of code is processed on the GPU, and may even launch another piece of code onto the GPU before the completion of preceding GPU code. The GPU queues this launch request, and executes the code later when it is available.

3 Programming Model

The GPU is a device to accelerate particular program code rather than a control unit like the CPU. User programs hence start execution on the CPU, and launch pieces of code, often referred to as *GPU kernels*[1], onto the GPU to get accelerated. There are *at least* three major steps for user programs to take to accelerate on the GPU.

1. **Memory Allocation**: First of all, user programs must be allocated memory spaces on the host and device memory that are required for computation. There are several types of memory for the GPU: *shared*, *local*, *global*, *constant*, and *heap*.
2. **Data Copy**: Input data must be copied from the host to the device memory before the GPU kernel starts on the GPU. Usually output data is also copied back from the device to the host memory to return the computed result to user programs.
3. **Kernel Launch**: GPU-accelerated program code must be launched from the CPU to the GPU at runtime, as the GPU itself is not a control unit.

The memory-allocation phases do not likely access the GPU, and must manage the device memory address regions available for each request. The data-copy and kernel-launch phases, on the other hand, need to access the GPU to move data between the host and the device memory, and launch GPU program code. Figure 2 illustrates an example showing a brief execution flow of GPU-accelerated matrix multiplication, i.e., $A[] \times B[] = C[]$. The GPU kernel image must be loaded on the

[1] To avoid misunderstandings, a term "kernel" always indicates GPU program code and never points to the operating system kernel in this paper.

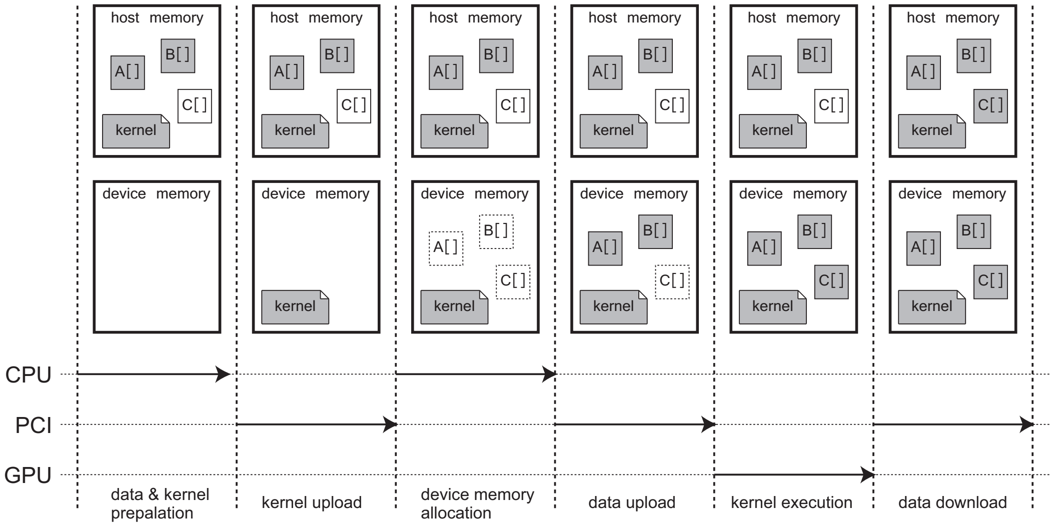

Figure 2. Example of an execution flow of matrix multiplication $A[] \times B[] = C[]$.

host memory. Two input buffers, *A*[] and *B*[], must also hold valid values for computation, while an output buffer *C*[] may be empty. The kernel image is usually uploaded at the beginning. Since the GPU uses the device memory for data access, the data spaces must be allocated on the device memory. The input buffers are then copied onto these allocated data spaces on the device memory via the PCI bus. After the input data are ready on the device memory, the GPU kernel starts execution. The output data are usually copied back onto the host memory. This is a generic flow to accelerate the program on the GPU.

GPU programming requires the device and the host parts. The device part contains kernels coded by GPU instructions, while the host part is more like a main thread running on the CPU to control data copies and kernel launches. The host part can be written in an existing programming language, such as C and C++, but must be aligned with an application programming interface (API) defined by the programming framework to communicate with the device part. The following are well-known GPU programming frameworks:

- **Open Graphics Language (OpenGL)** provides a set of library functions that allow user-space applications to program GPU shaders and upload it to the GPU to accelerate 2-D/3-D graphics processing.
- **Open Computing Language (OpenCL)** is a C-like programming language with library functions support. It can parallelize programs conceptually on any device, such as GPUs, Cell BE, and multi-core CPUs.
- **Compute Unified Device Architecture (CUDA)** is also a C-like programming language with library functions support. It can parallelize programs like OpenCL, but is dedicated to the GPU.
- **Hybrid Multicore Parallel Programming (HMPP)** is a compiler pragma extension to parallelize programs conceptually on any device. OpenMP also employs this programming style but is dedicated to multi-core CPUs.

Graphics processing is typically more complicated than general computing. A graphics pipeline comprises dozens of stages, where vertex data come in from one end of the pipeline, got processed in each stage, and the rendered frame comes out the other end. Some stages are programmable and others are not. For example, vertex and pixel shaders are programmable, but rasterization and format conversion are fixed functions. General computing, meanwhile, uses only shader units, also known as compute (or CUDA) cores. It depends on user-space programs to upload GPU code. The GPU code can be compiled at runtime or offline. Compute programs are often compiled offline as just parallelized on compute cores, while graphics programs would need runtime compilation, since they use many shaders for graphics operations. Once compiled as GPU code binaries, however, they are loaded onto the GPU at runtime in the same manner. Hence, the software stack needs no modification in the operating system.

4 Resource Management Model

In this section, we present a basic model of GPU resource management, particularly along with the Linux system stack, accommodating NVIDIA's proprietary driver [28], PathScale's open-source driver [33], and Linux's open-source driver [6]. Windows Display Driver Model (WDDM) [35] may also be applicable to our model, since NVIDIA could share about 90% of code between Linux and Windows [34]. As mentioned in Section 2, the following discussion assumes the NVIDIA's Fermi architecture [30], but is also conceptually applicable to most of today's GPU architectures.

4.1 System Stack

The GPU architecture defines a set of *GPU commands* to enable the device driver and the user-space runtime engine to control data copies and kernel launches. The device driver provides primitives for user-space programs to send GPU com-

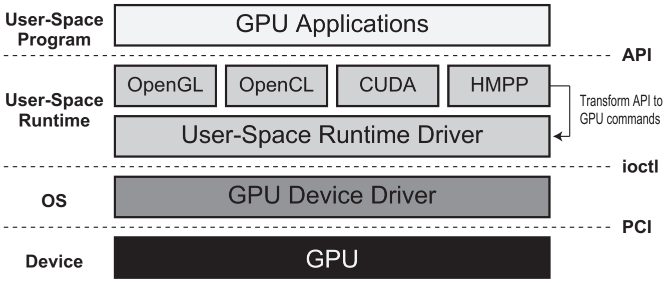

Figure 3. System stack for GPU processing.

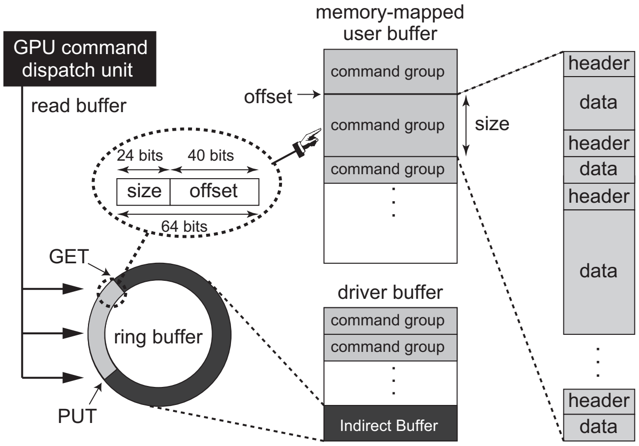

Figure 4. GPU command submissions.

mands to the GPU, and the user-space runtime engine provides a specific API to write user programs, abstracting the low-level primitives at the device driver. An `ioctl` system call is often used to interface between the device driver and the runtime engine. GPU commands are different GPU instructions that code GPU kernels, and there are many types of GPU commands. The device driver sends GPU commands to manage GPU resources including GPU contexts and device memory management units, while the runtime engine generates GPU commands to control the execution flows of user programs, e.g., data copies and kernel launches.

Figure 3 illustrates the system stack in our GPU resource management model. Applications call API library functions provided by the runtime engine. The front-end of the runtime engine is dependent on the programming framework, which transforms the API calls to GPU commands that are executed by the GPU. The runtime driver is the back-end of the runtime engine that abstracts the `ioctl` interface at the device driver, simplifying the development of programming frameworks. Performance optimization of the runtime engine can be unified in this layer. The device driver is responsible for submitting the GPU commands, received through the `ioctl` system call, to the GPU via the PCI bus.

4.2 GPU Channel Management

The device driver must manage GPU channels. The GPU channel is an interface that bridges across the CPU and the GPU contexts, especially when sending GPU commands from the CPU to the GPU. It is directly attached to the dispatch unit inside the GPU, which passes incoming GPU commands to the compute (or rendering) unit where GPU code is executed. The GPU channel is the only way to send GPU commands to the GPU. Hence, user programs must be allocated GPU channels. Multiple channels are supported in most GPU architectures. For instance, the NVIDIA's Fermi architecture supports 128 channels. Since each context requires at least one channel to use the GPU, at most 128 contexts are allowed to exist at the same time. GPU channels are independent of each other, and represent separate address spaces.

Figure 4 illustrates how to submit GPU commands to the GPU within a channel. The GPU channel uses two types of buffers in the operating-system address space to store GPU commands. One is memory-mapped onto the user-space buffer, into which the user-space runtime engine pushes GPU commands. The other buffer is directly used by the device driver to send specific GPU commands that control the GPU, such as status initialization, channel synchronization, and mode setting. GPU commands are usually grouped into multiple units, often referred to as GPU command groups. There are no constraints on how to compose GPU command groups. A single GPU command group may contain only one GPU command or numbers in the thousands, as far as the memory space allows. Regardless of how many GPU command groups are submitted, the GPU treats them as just a sequence of GPU commands. However, the number of GPU command groups could affect throughput, since GPU commands in each group are dispatched by the GPU in a burst manner. The more GPU commands are included in a group, the less communication is required between the CPU and the GPU. Instead, it could cause long blocking durations, since the device driver cannot *directly* preempt the executions of GPU contexts launched by preceding sets of GPU commands.

While the runtime engine pushes GPU commands into the memory-mapped buffer, it also writes *packets*, each of which is a (*size* and *address*) tuple to locate the corresponding GPU command group, into a specific ring buffer provided in the operating-system buffer, often referred to as the indirect buffer. The device driver configures the GPU command dispatch unit to read this ring buffer to pull GPU commands. This ring buffer is controlled by `GET` and `PUT` pointers. The pointers start from the same place. Every time packets are written to the buffer, the device driver moves the `PUT` pointer to the tail of the packets, and sends a signal to the GPU command dispatch unit to pull the GPU command groups located by the packets between the `GET` and `PUT` pointers. Afterward, the `GET` pointer is automatically updated to the same place as the `PUT` pointer. Once these GPU command groups are submitted to the GPU, the device driver does not manage them any longer, and just continues to submit the next set of GPU command groups, if any. As a consequence, this ring buffer plays a role of a command queue for the device driver.

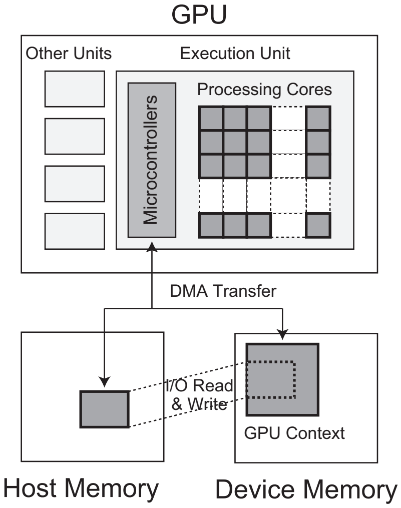

Figure 5. GPU context management model.

Each GPU command group may include multiple GPU commands. Each GPU command is composed of the header and data. The header contains *methods* and the data size, while the data contain the values being passed to the methods. Some methods are shared between compute and graphics, and others are specific to each. GPU command execution is out-of-order within the same GPU channel, and GPU channels could be switched implicitly by the GPU itself. It should be noted that the GPU channel is a path to send GPU commands and some other structures to control the GPU. GPU kernel images and their data buffers are uploaded onto the device memory through direct memory access (DMA), while DMA operation itself is controlled by GPU commands.

4.3 GPU Context Management

GPU contexts consist of memory-mapped I/O registers and hardware registers, which must be initialized by the device driver at the beginning. While memory-mapped I/O registers can be directly read and written by the device driver through the PCI bus, hardware registers need GPU sub-units to be read and written. There are multiple ways provided to access the context values. Reading from and writing to memory-mapped I/O registers on the CPU is the most straightforward way, but PCI bus communications are generated for all such operations. The GPU alternatively provides several hardware units to transfer data among the host memory, the device memory, and GPU registers in a burst manner.

Figure 5 illustrates a conceptual model of how to manage the GPU context. Generally, the GPU context is stored on the device memory. It could also be stored on the host memory, as the GPU can access both the host and device memory, but the device memory is strongly recommended due to performance issues, i.e., the GPU accesses the device memory much faster than the host memory. The host memory is often used to store one-time accessed data, such as the firmware program image of microcontrollers. Some parts of the GPU context are not directly accessible to the host memory. DMA transfers are needed to manage such parts of the GPU context through the microcontroller. It should be noted that the microcontroller also contains memory spaces where some data accessed by the GPU context are stored. Hence, DMA transfers are also needed to manage such data. The GPU has many hardware units other than the execution unit, some of which are used by the GPU context, but we do not describe details.

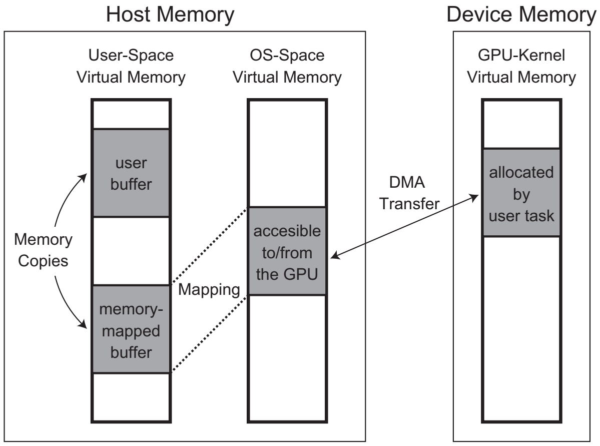

Figure 6. Host-device data copy model.

4.4 Memory Management

Memory management for GPU applications is associated with at least three address spaces. Given that a user program starts on the CPU first, the user buffer is created within the user-space virtual memory on the host memory. This buffer must be copied to the operating-system virtual memory, since the device driver must access it to transfer data onto the device memory. The destination of the data transfer on the device memory must match the address space allocated by the user program beforehand for the corresponding GPU kernel program. As a consequence, there are three address spaces associated with memory management: (i) the user-space virtual memory, (ii) the operating-system virtual memory, and (iii) GPU-kernel virtual memory.

Figure 6 depicts how to copy data from the user buffer on the host memory to the allocated buffer on the device memory. The user buffer must be first copied to the operating-system virtual memory space accessible to the device driver. A memory-mapped buffer may be used for this purpose, which is supported by the POSIX standard as the `mmap` system call. Once the buffer is memory-mapped, the user program can use it quite flexibly. Another approach to this data copy is that the user program communicates with the device driver, using I/O system calls, given that most operating systems provide a function to copy data from the user buffer to the operating-system virtual memory. Figure 6, however, assumes the first approach, which is adopted by Nouveau and PSCNV (and perhaps NVIDIA's proprietary driver). Once the buffer

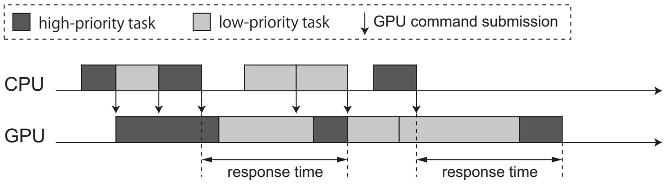

Figure 7. GPU processing without scheduling.

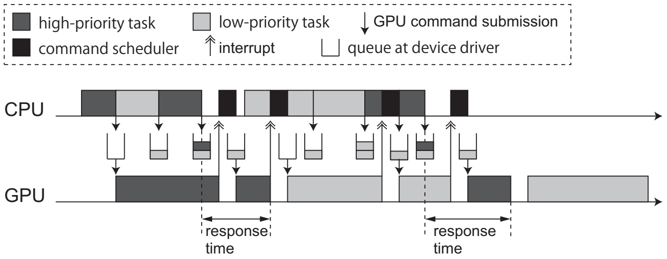

Figure 8. GPU processing with scheduling.

is copied to the operating-system virtual memory, the device driver transfers it to the device memory space allocated by the user program. Usually, the GPU provides virtual memory for each GPU channel to support multi-tasking, and the address of this device virtual memory is visible to user-space so that they can handle where to send data for computation.

5 Operating Systems Challenges

In this section, we consider operating systems challenges for GPU resource management. The following discussion is based on our experience and favorite research perspective, and does not cover the complete area of GPU resource management. The list of challenges provided herein must extend to advance further operating systems research for GPU technology. We however believe that the following discussion will lead to ideas of where we are at and where to go.

5.1 GPU Scheduling

GPU scheduling is perhaps the most important challenge to leverage the GPU in multi-tasking environments. Without GPU scheduling, GPU kernel programs are launched in first-in-first-out (FIFO) fashion, since the GPU command dispatch unit pulls GPU command groups in their arrival order. Hence, GPU processing becomes non-preemptive in a strong sense. Figure 7 illustrates a response-time problem caused due to the absence of GPU scheduling support, where two tasks with different priorities launch GPU code three times each as depicted on the CPU time line. The first launch of the high-priority task is serviced immediately, since the GPU has been idle. However, this is not always the case in multi-tasking environments. For instance, the second and the third launches of the high-priority task are blocked by the preceding executions of GPU contexts launched by the low-priority task. This blocking problem appears due to the nature of FIFO dispatching. Thus, tasks accessing the GPU need to be scheduled appropriately to avoid interference on the GPU.

Design Concept: The design of GPU schedulers falls into two categories. One approach implements a scheduler at the device-driver level to reorder GPU command groups submissions, given that the executions of GPU contexts are launched by GPU commands. GPU schedulers in the state of the art [2, 17] are designed based on this approach. For instance, TimeGraph [17] queues GPU command groups in the device driver space, and configures the GPU to generate interrupts to the CPU from the GPU upon completions of GPU code launched by prior GPU command groups so that the scheduler can be invoked to dispatch the next GPU command groups. Scheduling points are hence created at GPU command group boundaries. TimeGraph particularly dispatches GPU command groups according to task priorities, as depicted in Figure 8. The high-priority task can thereby respond quickly on the GPU whereas introducing additional overhead for the scheduling process. This is a trade-off, and it was demonstrated that this overhead is inevitable to protect important GPU applications from performance interference [16, 17].

Unfortunately, this device-driver approach still suffers from the non-preemptive nature of GPU processing. Specifically, the device driver can reorder GPU command groups submissions, but cannot directly preempt GPU contexts. For instance, the example in Figure 8 shows that the third launch of the high-priority task needs to wait for the completion of the second launch of the low-priority task. To make the GPU fully preemptive, GPU context switching needs to be supported at the microcontroller level, as mentioned in [17]. The design and implementation of such microcontrollers firmware, however, are very challenging issues. Some ideas could be extended from satellite kernels [27].

There are several other approaches to GPU scheduling. As studied in [8], the CPU scheduler is still effective in controlling GPU execution, since GPU code is launched from the CPU, and interrupts from the GPU are received on the CPU. However, GPU resource management is inevitably coarse-grained with this approach, due to the fact that the CPU scheduler is not aware of GPU command submissions and GPU contexts. We may also use compile-time and application-programming approaches [3, 10, 36], if modifications and recompilations of programs, using specific compilers, APIs, and algorithms, are acceptable. GPU resource management at the device-driver level, on the other hand, is finer-grained, and there is no need to compromise the generality of programming frameworks.

Scheduling Algorithm: Assuming that GPU schedulers are provided, the next question is: what scheduling algorithms are desired? We believe that at least two scheduling algorithms are required due to the hierarchy of the CPU and the GPU.

First, we insist that the CPU scheduling algorithm should consider the presence of the GPU. Particularly for real-time systems, classical deadline-driven algorithms, such as Earliest Deadline First (EDF) [23], are not effective as they are. Figure 9 shows an example where two tasks, one accesses the

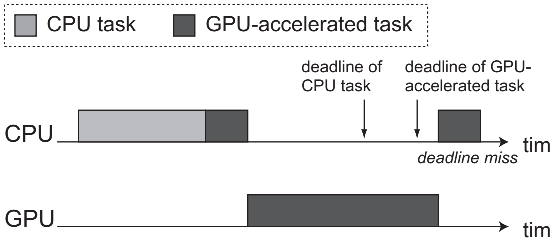

Figure 9. Deadline-driven CPU scheduling in the presence of the GPU.

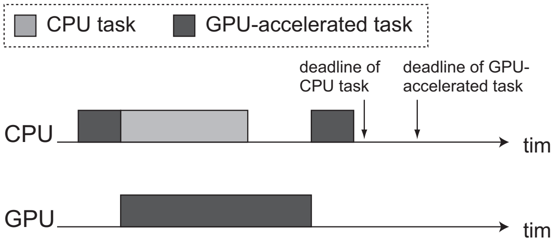

Figure 10. GPU-aware CPU scheduling in the presence of the GPU.

GPU (“GPU-accelerated task”) and the other does not (“CPU task”), are scheduled on the CPU using the EDF algorithm. Consider that the CPU task is assigned a higher priority at some time, while the GPU-accelerated task has a large computation time on the GPU. Since GPU code is never launched until the CPU task is completed, the GPU remains idle while the CPU task is executing, and the CPU remains idle while the GPU-accelerated task is executing on the GPU. A more efficient algorithm should be developed to generate such a schedule that is depicted in Figure 10, where CPU and GPU times are effectively overlapped. Laxity-driven algorithms, such as Earliest Deadline Zero Laxity (EDZL) [4] and Earliest Deadline Critical Laxity (EDCL) [18], could alternate EDF, but an in-depth investigation is desirable.

Scheduling parallel tasks on the GPU is another issue of concern. Gang scheduling and co-scheduling [31] are well-known concepts for such a parallel multi-tasking model. The real-time systems community has also explored these scheduling methods recently [14, 21]. We believe that the concepts of gang scheduling and co-scheduling are useful to design the GPU scheduling algorithm. However, we must consider the constraints of the GPU. In general, the GPU is designed to execute threads in some group, also known as a warp in the Fermi architecture, simultaneously. Hence, the minimum unit of scheduling is a block of threads instead of a single thread. We still believe that existing concepts of parallel job scheduling are applicable, while algorithms implementation is a very challenging issue.

Data-copy Scheduling: As shown in Figure 2, data must be copied on to the device memory before GPU code is executed, and it is often required to copy the computation result back on to the host memory. The amount of time taken to perform the data copy depends on the data size. Furthermore, data-copy operations generate non-preemptive regions, which affect the performance and responsiveness of involved applications. Specifically, since data copies between host and device memory spaces are performed by DMA, as shown in Figure 6, and the device driver provides no way to preempt these DMA transfers once they are launched. Therefore, GPU scheduling must also be involved in data copies as well as the executions of GPU contexts.

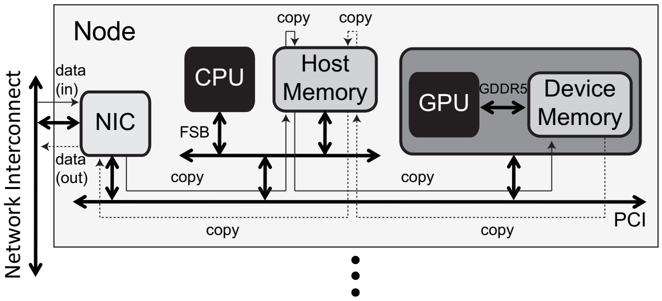

Figure 11. Data communication with the GPU.

5.2 GPU Clustering

Further research challenges include support for clustered multiple GPUs. GPU clustering is a key technology to use GPUs for HPC applications. Currently, we are developing this technology based on our GPU resource management model and GPU scheduling schemes to provide first-class support for GPU-based interactive data centers, supercomputers, and cyber-physical systems. These applications are data-intensive as well as compute-intensive. Hence, performance is affected by data communications across multiple GPUs.

GPU clusters are generally hierarchical. Each node is composed of a small number of GPUs clustered on a board. Many such nodes are further clustered as a system. Since these two types of clustering use different technologies [37], operating systems support must be provisioned differently.

On-board GPU clusters: The management of multiple GPUs on a board may be either in the user-space runtime or the operating system. It is usually an application task that determines which GPU to use for computation. Data copies between GPUs can also be handled in user-space by copying data via the host memory. However, the operating system is responsible to configure the GPUs, if fast direct data copies between two different device memory spaces via the PCI bus or the SLI interface are required. For instance, the CUDA 4.0 specification requires such a data communication interface, often referred to as *GPU Direct*. In GPU clustering, hence, scheduling must be involved in device-to-device data copies in addition to the executions of GPU contexts and data copies between the host and the device memory.

Networked GPU clusters: The management of multiple GPUs connected over the network is more challenging, as it involves networking. GPU-based HPC applications could scale to use thousands of nodes [38]. We believe that data communications over the network will be a bottleneck to scale the performance of GPU clusters in the number of nodes. Figure 11 illustrates how to send and receive data between the network interface card (NIC) and the GPU in a basic model. Generally, when the NIC receives data, the NIC device driver transfers this data to the host (main) memory via DMA. The GPU device driver next needs to copy this data to the device memory, but the address spaces visible to the NIC and the GPU are different, as these device drivers are usually developed individually. Therefore, the GPU device driver needs to copy the data to another space on the host memory accessible to the GPU. The same data-copy path is used from the opposite side, when the GPU sends data to the NIC. The NVIDIA's GPU Direct technology enabled this data communication stack to skip host-to-host data copy [24], but non-trivial overheads caused by data copies among the NIC, the host memory, and the device memory are still imposed on networked GPU clusters. Coordination of the NIC and the GPU device drivers is needed to reduce such communication overhead.

The same performance issue would appear in distributed systems exploiting GPUs. For example, autonomous vehicles using GPUs and camera sensors need to get data from the camera sensors to the GPUs. In storage systems, data may also come through the network to the GPUs. Coordination of heterogeneous devices and resources is therefore an important problem for future work.

5.3 GPU Virtualization

Virtualization is a useful technique widely adopted in many application domains to isolate clients in the system, and make the system compositional and dependable. Virtualizing GPUs hence provides the same benefits for GPU-accelerated systems. GPU virtualization support has been provided by runtime engines [20], VMMs [11, 12], and I/O managers [7] in the literature. We however believe that there is a problem space for operating systems to support GPU virtualization. In fact, VMs eventually access the GPU via the device driver in the host operating system. Hence, GPU resource management at the device-driver level plays a vital role for GPU virtualization as well. For instance, prioritization and isolation capabilities provided at the device driver level [17] could be very powerful to run GPU VMs.

Our major concern for GPU virtualization appears when different guest operating systems are installed in VMs. The GPU is typically controlled by microcontrollers as depicted in Figure 5. These microcontrollers require firmware to operate correctly, which must be uploaded by the device driver. The firmware image must match the assumption of the device driver. However, GPU device drivers in different guest operating systems may use different firmware images and assumptions. For instance, one guest operating system may provide firmware with context switching support while another may provide that with power management support. In such a case, the device driver in the host operating system needs to switch firmware accordingly among these different guest operating systems, or provide "all-in-one" firmware that provides all necessary functions.

5.4 GPU Device Memory Management

Device memory spaces allocated by user programs are typically pinned. They never become available for different programs unless freed explicitly, resulting in the allocatable memory size limited to the device memory size. This is not an efficient memory management model. The GPU often supports virtual memory to isolate address spaces among GPU channels (contexts). Operating systems should utilize this virtual memory functionality to expand the allocatable device memory spaces, just as they support it for the host memory through memory management units (MMUs). We could establish a hierarchical model of device memory, host memory, and storage to virtualize the device memory as a nearly infinite size of memory, which significantly improve the flexibility and availability of GPU programming.

GPU applications are often very data-intensive. Hence, virtual memory management should cope with frequent data swapping among the device memory, host memory, and storage to maintain performance and interactivity. We believe that prior memory management models [15, 42] are applicable to GPU device memory management to hide the penalty of data swapping. In the presence of multiple GPUs, especially, distributed shared memory (DSM) [22] systems could also be beneficial to transparently increase the available memory space for GPU programming.

5.5 Coordination with Runtime Engines

GPU operations are controlled by GPU command groups issued from user-space programs. For instance, GPU kernel launches and data copies between the host and the device memory are triggered by a specific sets of GPU commands. However, the operating system does not recognize what types of GPU commands are issued from user-space programs. It just controls the ring buffer pointing to the memory location where GPU command groups are stored by the user-space runtime engine so that the GPU can dispatch them, as illustrated in Figure 4. Prior work [2, 16, 17] hence queue and dispatch *all* GPU command groups. However, there are many trivial GPU command groups that do not generate GPU workload. Since the overhead for queuing and dispatching GPU command groups is sometimes significant [17], the operating system should select an appropriate set of GPU command groups to queue and dispatch, including those related to GPU kernel executions and data copies. To do so, the operating system must support an interface for the user-space runtime engine to specify what types of GPU command groups are submitted.

In fact, there are many pieces of useful information to share between the operating system and the user-space runtime engine. To support real-time systems, for instance, it is preferable to know execution times consumed on the GPU before

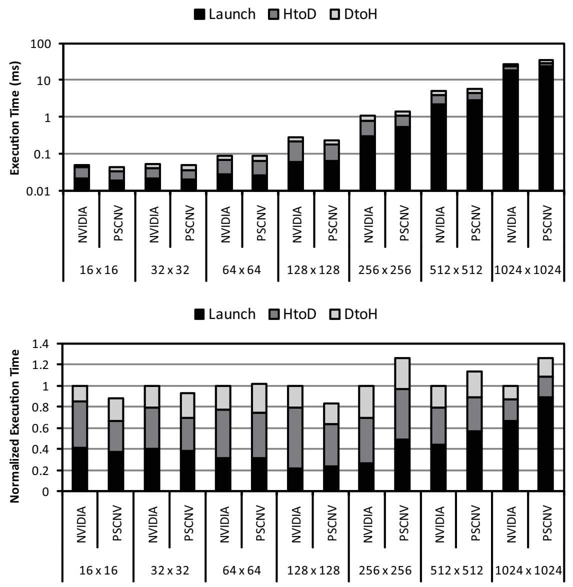

Figure 12. Performance comparison of NVIDIA's proprietary driver and PSCNV.

launching GPU kernels to account and enforce the executions of GPU contexts. It is possible to predict execution times based on a sequence of GPU command groups, but the predicted execution times could be imprecise depending on GPU workload [17]. The operating system should therefore preferably provide an interface to obtain such information from user-space programs.

5.6 Open-Source Implementation

Developing open-source tools is an essential duty to share ideas about systems implementation and facilitate research. Linux, for instance, is a well-known open-source software used in operating systems research. Nouveau [6] and PSCNV [33] are open-source GPU device drivers, available with Linux, for NVIDIA's GPUs. Our previous studies on TimeGraph [16, 17] particularly used Nouveau to implement and evaluate a new real-time GPU command scheduler.

Nouveau is often used in conjunction with an open-source OpenGL runtime engine, Gallium3D [25], for 3-D graphics applications. It should be noted that the performance of this open-source software stack is even competitive with NVIDIA's proprietary software [5], though its usage is limited to graphics applications for now.

PSCNV forked from Nouveau to support general compute programs, which is managed by PathScale Inc., as part of its GPGPU software solution [32]. We are currently involved in its development. Their corresponding runtime engine supports HMPP and CUDA for now, and OpenCL support is also in consideration. In fact, this PathScale's runtime engine can be used in conjunction with NVIDIA's proprietary driver as well. Therefore, fair performance comparisons of these proprietary and open-source device drivers can perform under the same runtime engine. The source code of this runtime engine is not yet open to the public, but it may be available upon request for the research community.

Figure 12 shows a performance comparison of NVIDIA's closed-source proprietary driver and the PSCNV open-source driver in integer matrix multiplication operation of variable sizes, using an NVIDIA GeForce GTX 480 graphics card, where "Launch" represents the execution time of the launched GPU kernel, and "HtoD" and "DtoH" represent data copy durations of host-to-device and device-to-host directions respectively. The GPU clock rate is set at maximum. The GPU kernel of matrix multiplication is compiled using NVIDIA CUDA Toolkit 3.2 [29]. Both drivers run under PathScale's runtime engine. According to our evaluation, the performance difference is very small for a small matrix, while NVIDIA's driver provides better performance for a large matrix. This is mainly attributed to the fact that this open-source device driver has not yet figured out how to activate "performance mode" that boosts the performance of the GPU aside from the clock rate. The performance difference, however, is limited to about 20% at most. Once we learn how to fully configure the GPU, such a performance difference would disappear. We hence believe that open-source software is now reliable enough to conduct operating systems research on GPU resource management.

6 Concluding Remarks

In this paper, we have presented the state of the art in GPU resource management. GPU technology is promising in many application domains due to its high performance and energy efficiency. Most current solutions, however, are focused on how to accelerate a single application task, and multi-tasking problems are not widely discussed. This paper identified core challenges for operating systems research to efficiently use the GPU in multi-tasking environments, and also provided some insights into their solutions. The identified list of challenges needs to expand as our understanding progresses. In particular, real-time systems need to address additional timing issues. We are fortunate to have open-source software that could underlie to explore such new domains of operating systems research. We believe that this paper will encourage research communities to further advance GPU resource management schemes for a grander vision of GPU technology.

Acknowledgment

We thank PathScale for sharing their technology and source code with us in this work.

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] M. Bautin, A. Dwarakinath, and T. Chiueh. Graphics Engine Resource Management. In *Proceedings of the Annual Multimedia Computing and Networking Conference*, 2008.

[3] L. Chen, O. Villa, S. Krishnamoorthy, and G. Gao. Dynamic Load Balancing on Single- and Multi-GPU Systems. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, 2010.

[4] H. Cho, B. Ravindran, and E.D. Jensen. Efficient Real-Time Scheduling Algorithms for Multiprocessor Systems. *IEICE Transactions on Communications*, 85:807–813, 2002.

[5] Linux Open-Source Community. Nouveau Companion 44. http://nouveau.freedesktop.org/.

[6] Linux Open-Source Community. Nouveau Open-Source GPU Device Driver. http://nouveau.freedesktop.org/.

[7] M. Dowty and J. Sugeman. GPU Virtualization on VMware's Hosted I/O Architecture. *ACM SIGOPS Operating Systems Review*, 43(3):73–82, 2009.

[8] G. Elliott and J. Anderson. Real-Time Multiprocessor Systems with GPUs. In *Proceedings of the International Confrence on Real-Time and Network Systems*, 2010.

[9] A. Gharaibeh, S. Al-Kiswany, S. Gopalakrishnan, and M. Ripeanu. A GPU Accelerated Storage System. In *Proceedings of the ACM International Symposium on High Performance Distributed Computing*, pages 167–178, 2010.

[10] M. Guevara, C. Gregg, K. Hazelwood, and K. Skadron. Enabling Task Parallelism in the CUDA Scheduler. In *Proceedings of the Workshop on Programming Models for Emerging Architectures*, pages 69–76, 2009.

[11] V. Gupta, A. Gavrilovska, N. Tolia, and V. Talwar. GViM: GPU-accelerated Virtual Machines. In *Proceedings of the ACM Workshop on System-level Virtualization for High Performance Computing*, pages 17–24, 2009.

[12] V. Gupta, K. Schwan, N. Tolia, V. Talwar, and P. Ranganathan. Pegasus: Coordinated Scheduling for Virtualized Accelerator-based Systems. In *Proceedings of the USENIX Annual Technical Conference*, 2011.

[13] Intel. Intel Microarchitecture Codename Sandy Bridge. http://www.intel.com/.

[14] S. Kato and Y. Ishikawa. Gang EDF Scheduling of Parallel Task Systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 459–468, 2009.

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

[16] 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.

[17] 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.

[18] S. Kato and N. Yamasaki. Global EDF-based Scheduling with Efficient Priority Promotion. In *Proceedings of the IEEE Embedded and Real-Time Computing Systems and Applications*, pages 197–206, 2008.

[19] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A.D. Nguyen, T. Kaldewey, V.W. Lee, S.A. Brandt, and P. Dubey. FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. In *Proceedings of the 2010 ACM SIGMOD/PODS Conference*, 2010.

[20] H.A. Lagar-Cavilla, N. Tolia, M. Satyanarayanan, and E. de Lara. VMM-Independent Graphics Acceleration. In *Proceedings of the ACM/Usenix International Conference on Virtual Execution Environments*, pages 33–43, 2007.

[21] K. Lakshmanan, S. Kato, and R. Rajkumar. Scheduling Parallel Real-Time Tasks on Multi-core Processors. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 259–268, 2010.

[22] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, 1989.

[23] L. Liu and J.W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, 20:46–61, 1973.

[24] Mellanox. NVIDIA GPUDirect Technology– Accelerating GPU-based Systems (Whitepaper). http://www.mellanox.com/pdf/whitepapers/TB_GPU_Direct.pdf.

[25] Mesa3D. http://www.mesa3d.org/.

[26] 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 *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 463–469, 2007.

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

[28] NVIDIA. Linux X64 (AMD64/EM64T) Display Driver. http://www.nvidia.com/.

[29] NVIDIA. NVIDIA CUDA Toolkit Version. http://developer.nvidia.com/cuda-toolkit-32-downloads.

[30] 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.

[31] J.K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Proceedings of the IEEE International Conference on Distributed Computing Systems*, pages 22–30, 1982.

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

[33] PathScale. PSCNV GPU Device Driver. https://github.com/pathscale/pscnv/.

[34] Phoronix. NVIDIA Developer Talks Openly About Linux Support. http://www.phoronix.com/scan.php?page=article&item=nvidia_qa_linux&num=2.

[35] S. Pronovost, H. Moreton, and T. Kelley. Windows Display Driver Model (WDDM v2) And Beyond. In *Windows Hardware Engineering Conference*, 2006.

[36] A. Saba and R. Mangharam. Anytime Algorithms for GPU Architectures. In *Proceedings of the Analytic Virtual Integration of Cyber-Physical Systems Workshop*, 2010.

[37] D. Schaa and D. Kaeli. Exploring the Multiple-GPU Design Space. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, 2009.

[38] 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.

[39] S. Thrun. GTC Closing Keynote. http://livesmooth.istreamplanet.com/nvidia100923/, 2010.

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

[41] C. Urmson, J. Anhalt, H. Bae, D. Bagnell, C. Baker, R. Bittner, T. Brown, M. Clark, M. Darms, D. Demitrish, J. Dolan, D. Duggins, D. Ferguson, T. Galatali, C. Geyer, M. Gittleman, S. Harbaugh, M. Hebert, T. Howard, S. Kolski, M. Likhachev, B. Litkouhi, A. Kelly, M. McNaughton, N. Miller, J. Nickolaou, K. Peterson, B. Pilnick, R. Rajkumar, P. Rybski, V. Sadekar, B. Salesky, Y-W. Seo, S. Singh, J. Snider, J. Struble, A. Stentz, M. Taylor, W. Whittaker, Z. Wolkowicki, W. Zhang, and J. Ziglar. Autonomous Driving in Urban Environments: Boss and the Urban Challenge. *Journal of Field Robotics*, 25(8):425–466, 2008.

[42] T. Yang, T. Liu, E.D. Berger, S.F. Kaplan, and J.E-B. Moss. Redline: First Class Support for Interactivity in Commodity Operating Systems. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pages 73–86, 2008.