Gdev: First-Class GPU Resource Management in the Operating System

Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Scott Brandt
Department of Computer Science, UC Santa Cruz

Abstract

Graphics processing units (GPUs) have become a very powerful platform embracing a concept of heterogeneous many-core computing. However, application domains of GPUs are currently limited to specific systems largely due to a lack of "first-class" GPU resource management for general-purpose multi-tasking systems.

We present Gdev, a new ecosystem of GPU resource management in the operating system (OS), which allows the user space as well as the OS itself to use GPUs as first-class computing resources. More specifically, Gdev provides a virtual memory management scheme to allow GPU contexts to allocate device memory exceeding the physical capacity, and share memory space with other contexts for inter-process communication (IPC). It also supports a GPU scheduling scheme to virtualize a single GPU into multiple logical GPUs, providing an isolation among specific user groups in multi-tasking systems.

Our evaluation using an open-source implementation of Gdev with Linux and an NVIDIA GPU shows that Gdev is competitive to proprietary software even in basic performance. In addition, for instance, we gain about a 2x speedup for an encrypted file system by allowing the OS to use the GPU. The shared device memory scheme also improves makespans of data-flow programs by up to 49%, and an error in the utilization of virtualized GPUs provided by Gdev is limited within 7%.

1 Introduction

Recent advances in many-core technology have achieved an order-of-magnitude gain in computing performance. Examples include *graphics processing units* (GPUs) – mature compute devices that best embrace a concept of heterogeneous many-core computing. In fact, TOP500 Supercomputing Sites disclosed in November 2011 [29] that three of the top five supercomputers employ clusters of GPUs as primary computing resources. Of particular note is that scientific climate applications have achieved 80x speedups leveraging GPUs [27]. Such a continuous wealth of evidence on performance benefits of GPUs has encouraged application domains of GPUs to expand to general-purpose and embedded computing. For instance, previous work have demonstrated that GPU-accelerated systems achieved an order of 10x speedups for software routers [10], 20x speedups for encrypted networks [12], and 15x speedups for motion planning [19]. This rapid growth of general-purpose computing on GPUs, *a.k.a.*, GPGPU, is brought by emergence of new programming languages, such as CUDA [21].

Seen from these trends, GPUs are becoming more and more applicable for general-purpose systems. However, system software support for GPUs in today's market is tailored to accelerate particular applications dedicated to the system; it is not well-designed to integrate GPUs into general-purpose *multi-tasking* systems. Albeit speedups of individual application programs, the previous research raised above [10, 12, 19] could not provide performance or quality-of-service (QoS) management without system software support. Given that networked and embedded systems are by nature composed of multiple clients and components, it is essential that GPUs should be managed as first-class computing resources so that various tasks can access GPUs concurrently in a reliable manner.

The research community has articulated the needs of enhancement in the operating system (OS) [2, 15, 24], hypervisor [9], and runtime library [14] to make GPUs available in interactive and/or virtualized multi-tasking environments. However, all these pieces of work depend highly on the user-space runtime system, often included as part of proprietary software, which provides the user space with an application programming interface (API). This framework indeed limits the potential of GPUs to the user space. For example, it prevents the file system or network stack in the OS from using GPUs directly. There is another issue of concern with this framework: the device driver needs to expose resource management primitives to the user space, since the runtime system is employed in the user space, implying that non-privileged user-space programs may abuse GPU resources. As a matter of fact, we can launch any program on an NVIDIA GPU without using any user-space runtime libraries, but using an `ioctl` system call directly. This explains that GPUs should be protected by the OS as well as CPUs.

In addition to those conceptual issues, there exist more fundamental and practical issues with publicly-available GPGPU software. For example, memory allocation for GPU computing is not allowed to exceed the physical capacity of device memory. We are also not aware of any API that allows GPU contexts to share memory resources with other contexts. Such data management constraints may not be acceptable in general-purpose programming.

Contribution: We present **Gdev**, a new approach to GPU resource management in the OS that addresses the current limitations of GPU computing. Gdev integrates runtime support for GPUs into the OS, which allows the user space as well as the OS itself to use GPUs with the identical API set, while protecting GPUs from non-privileged user-space programs at the OS level. Building on this runtime-unified OS model, Gdev further provides first-class GPU resource management schemes for multi-tasking systems. Specifically, Gdev allows programmers to share device memory resources among GPU contexts using an explicit API. We also use this shared memory functionality to enable GPU contexts to allocate memory exceeding the physical size of device memory. Finally, Gdev is able to virtualize the GPU into multiple logical GPUs to enhance an isolation among working groups of multi-tasking systems. As a proof of concept, we provide an open-source implementation of Gdev. In summary, this paper makes the following contributions:

- Identifies the advantage/disadvantage of integrating runtime support for GPUs into the OS.
- Enables the OS itself to use GPUs.
- Makes GPUs "first-class" computing resources in multi-tasking systems – memory management for inter-process communication (IPC) and scheduling for GPU virtualization.
- Provides open-source implementations of the GPU device driver, runtime/API libraries, utility tools, and Gdev resource management primitives.
- Demonstrates the capabilities of Gdev using real-world benchmarks and applications.

Organization: The rest of this paper is organized as follows. Section 2 provides the model and assumptions behind this paper. Section 3 outlines the concept of Gdev. Section 4 and 5 present Gdev memory management and scheduling schemes. Section 6 describes our prototype implementation, and Section 7 demonstrates our detailed experimental results. Section 8 discusses related work. We provide our concluding remarks in Section 9.

2 System Model

This paper focuses on a system composed of a GPU and a multi-core CPU. GPU applications use a set of the API supported by the system, typically taking the following steps: (i) allocate space to device memory, (ii) copy data to the allocated device memory space, (iii) launch the program on the GPU, (iv) copy resultant data back to host memory, and (v) free the allocated device memory space. We also assume that the GPU is designed based on NVIDIA's *Fermi* architecture [20]. The concept of Gdev, however, is not limited to Fermi, but is also applicable to those based on the following model.

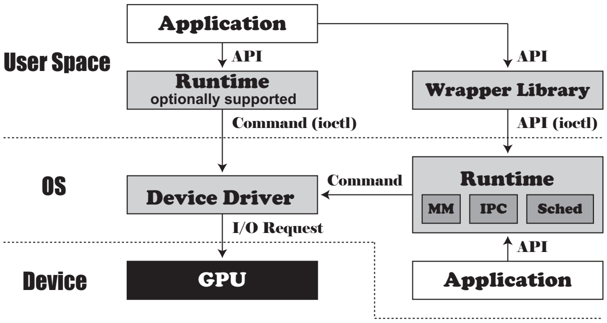


Figure 1: Logical view of Gdev's ecosystem.

Command: The GPU operates using the architecture-specific commands. Each GPU context is assigned with a FIFO queue to which the program running on the CPU submits the commands. Computations and data transfers on the GPU are triggered only when the corresponding commands are dispatched by the GPU itself.

Channel: Each GPU context is assigned with a GPU hardware channel within which command dispatching is managed. Fermi does not permit multiple channels to access the same GPU functional unit simultaneously, but allow them to coexist being switched automatically in hardware. This constraint may however be removed in the future architectures or product lines.

Address Space: Each GPU context is assigned with virtual address space managed through the page table configured by the device driver. Address translations are performed by he memory management unit on the GPU.

Compute Unit: The GPU maps *threads* assigned by programmers to *cores* on the compute unit. This thread assignment is not visible to the system, implying that GPU resource management at the system level should be context-based. Multiple contexts cannot execute on the compute unit at once due to the channel constraint, but multiple requests issued from the same context can be processed simultaneously. We also assume that GPU computation is non-preemptive.

DMA Unit: There are two types of DMA units for data transmission: (i) synchronous with the compute unit and (ii) asynchronous. Only the latter type of DMA units can overlap their operations with the compute unit. We also assume that DMA transaction is non-preemptive.

3 Gdev Ecosystem

Gdev aims to (i) enhance GPU resource management and (ii) extend a class of applications that can leverage GPUs. To this end, we integrate the major portion of runtime support into the OS. Figure 1 illustrates the logical view of Gdev's ecosystem. For a compatibility issue, we still support the conventional stack where applications make API calls to the user-space runtime library, but system designers may disable this stack to remove the concern discussed in Section 1. A new ecosystem introduced by Gdev is runtime support integrated in the OS, allowing the user space as well as the OS to use the identical API set. This ecosystem prevents non-privileged user-space programs from bypassing the runtime system to access GPUs. The wrapper library is a small piece of software provided for user-space applications, which relays API calls to the runtime system employed in the OS.

Leveraging this ecosystem, we design an API-driven GPU resource management scheme. Figure 1 shows that Gdev allows the OS to manage API calls, whereas the traditional model translates API calls to GPU commands before the OS receives them. As discussed in previous work [15], it is very hard to analyze GPU commands and recognize the corresponding API calls in the OS. Hence, the existing GPU resource management schemes in the OS [2, 15] compromise overhead to invoke the scheduler at every GPU command submission, unless an additional programming abstraction is provided [24]. On the other hand, Gdev can manage GPU resources along with API calls, without any additional programming abstractions.

Programming Model: We provide a set of low-level functions for GPGPU programming, called "Gdev API". Gdev API is a useful backend for high-level APIs, such as CUDA. The details of Gdev API can be found at our project website [25]. Programmers may use either Gdev API directly or high-level APIs built on top of Gdev API. This paper particularly assumes that programmers use the well-known CUDA Driver API 4.0 [21].

Gdev uses an existing programming framework and commodity compiler, such as NVIDIA CUDA Compiler (NVCC) [21]. When a program is compiled, two pieces of binary are generated. One executes on the CPU, and loads the other binary onto the GPU. The CPU binary is provided as an executable file or loadable module, while the GPU binary is an object file. Hence, both user-space and OS-space applications can use the same framework: (i) read the GPU binary file and (ii) load it onto the GPU. The detailed information embedded in the object file, such as code, static data, stack size, local memory size, and parameter format, may depend on the programming language, but the framework does not depend on it once the object file is parsed.

Resource Management: We provide device memory management and GPU scheduling schemes to manage GPUs as first-class computing resources. Specifically, we provide shared device memory for IPC, data swap for large memory demands, resource-based queuing for throughput, and bandwidth-aware resource partitioning for virtual GPU isolation. Since some of these features require access to low-level system information, such as I/O regions, DMA pages, and task control blocks, it is not straightforward for the traditional user-space runtime system to manage these pieces of information. Hence, we claim that Gdev is a suitable approach to first-class GPU resource management. The concept of Gdev is also not limited to GPUs, but can be generalized for a broad class of heterogeneous compute devices.

4 Device Memory Management

Gdev manages device memory using the virtual memory management unit supported by the GPU. Virtual address space for GPU contexts can be set through the page table. Gdev stores this page table in device memory, though it can also be stored in host memory. Beyond such basic pieces of memory management, this section seeks how to improve memory-copy throughput. We also explore how to share memory resources among GPU contexts, and support data swap for excessive memory demands.

4.1 Memory-Copy Optimization

Given that data move across device and host memory back and forth, memory-copy throughput could govern the overall performance of GPU applications. While the primary goal of this paper is to enhance GPU resource management, we respect standalone performance as well for practical use. Hence, we first study the characteristic of memory-copy operations.

Split Transaction: We often need to copy the same data set *twice* to communicate with the GPU, unless we allocate buffers to host I/O memory directly. One copy happens within host memory, moving data between main memory and host I/O memory, *a.k.a.*, pinned pages of host memory. The other copy happens between device and host I/O memory. In order to optimize this two-stage memory-copy operation, we split the data buffer into a fixed size of multiple chunks. Using split transactions, while some chunk is transferred within host memory, the preceding chunk can be transferred between device and host I/O memory. Thus, only the first and last pieces of chunks need to be transferred alone, and other chunks are all overlapped, thus reducing a total makespan almost half. An additional advantage of this method is that only the same size of an intermediate "bounce" buffer as the chunk size is required on host I/O memory, thus reducing the usage of host I/O memory significantly. It should be noted that "pinned" pages do not use split transaction.

Direct I/O Access: The split transaction is effective for a large size of data. For a small size of data, however, the use of DMA engines incurs non-trivial overhead by itself. Hence, we also employ a method to read/write data one by one by mapping device memory space onto host I/O memory space, rather than send/receive data in burst mode by using DMA engines. We have found that direct I/O access is much faster than DMA transaction for a small size of data. In Section 7, we will identify a boundary on the data size that inverts the latency of I/O access and DMA, and also derive the best chunk size to optimize memory-copy throughput.

4.2 Shared Device Memory

Existing GPU programming languages do not support an explicit API for IPC. For example, data communications among GPU contexts incur significant overhead due to copying data back and forth between device and host memory. Currently, an OS dataflow abstraction [24] is a useful approach to reduce such data movement costs; users are required to use a dataflow programming model. We believe that it is more flexible and straightforward for programmers to use a familiar POSIX-like method.

Gdev supports a set of API functions to share device memory space among GPU contexts respecting POSIX IPC functions of `shmget`, `shmat`, `shmdt`, and `shmctl`. As a high-level API, we extend CUDA to provide new API functions of `cuShmGet`, `cuShmAt`, `cuShmDt`, and `cuShmCtl` in our CUDA implementation so that CUDA applications can easily leverage Gdev's shared device memory functionality.

Our shared memory design is straightforward, though its implementation is challenging. Suppose that we use the above extended CUDA API for IPC. Upon the first call to `cuShmGet`, Gdev allocates new space to device memory, and holds an identifier to this memory object. After the first call, Gdev simply returns this identifier to this call. When `cuShmAt` is called, the allocated space is mapped to the virtual address space of the corresponding GPU context. This address mapping is done by setting the page table so that the virtual address points to the physical memory space of this shared memory object. The allocated space can be unmapped by `cuShmDt` and freed by `cuShmCtl`. If the shared memory object needs exclusive access, the host program running on the CPU must use traditional mutex/semaphore mechanisms.

4.3 Data Swapping

We have found that proprietary software in Linux [21] fails to allocate device memory exceeding the physical memory capacity, while the Windows display driver [23] supports data swapping to some extent. In either case, however, a framework of data swapping with GPUs has not been well studied so far. This section explores how to swap data in the presence of multiple GPU contexts.

Gdev uses the shared device memory functionality to achieve data swapping. When memory allocation fails due to a short of free memory space, Gdev seeks memory objects whose allocated size is greater than the requested size, and selects one owned by a low-priority context, where ties are broken arbitrarily. This "victim" memory object is shared by the caller context *implicitly*. Unlike an explicit shared memory object obtained through the API presented in Section 4.2, an implicit shared memory object must evict data when accessed by other contexts, and retrieve them later when the corresponding context is resumed. Since Gdev is designed API-driven, it is known when contexts may access the shared memory object:

- The memory-copy API will affect specific address space given by the API parameters. Hence, we need to evict only such data that cover this range.
- The compute-launch API may also be relevant to some address space, but its address range is not all specified when the API is called, since the program may use dynamic memory allocation. Hence, we need to evict such data that are associated with all the memory objects owned by the context.

We allocate swap buffers to host main memory for evicted data. Swapping itself is a simple asynchronous memory-copy operation, but is not visible to application programs. It should be noted that swapping never occurs when copying data from device to host memory. If the corresponding data set is evicted in the swap space, it can be retrieved from the swap space directly, and there is no need to swap it back to device memory.

Reducing Latency: It is apparent that the swapping latency could be non-trivial, depending on the data size. In order to reduce this latency, Gdev reserves a certain amount of device memory space as *temporal swap space*. Since a memory-copy operation within device memory is much faster than that between device and host memory, Gdev first tries to evict data to this temporal swap space. This temporarily-evicted data set is eventually evicted to host memory after a while to free up the swap space for other contexts. Gdev also tries to hide this second eviction latency by overlapping it with GPU computation launched by the same context. We create a special GPU context that is dedicated to memory-copy operations for eviction, since the compute and DMA units cannot be used by the same context simultaneously. This approach is quite reasonable because data eviction is likely to be followed by GPU computation. Evicted data, if exist, must be retrieved before GPU computation is launched. If they remain in the swap space, they can be retrieved at low cost. Else, Gdev retrieves them from host memory.

5 GPU Scheduling

The goal of the Gdev scheduler is to correctly assign computation and data transmission times for each GPU context based on the given scheduling policy. Although we make use of some previous techniques [14, 15], Gdev provides a new queuing scheme and virtual GPU support for multi-tasking systems. Gdev also propagates the task priority used in the OS to the GPU context.

5.1 Scheduling and Queuing

Gdev uses a similar scheme to TimeGraph [15] for GPU scheduling. Specifically, it allows GPU contexts to use GPU resources only when no other contexts are using the corresponding resources. The stalling GPU contexts are queued by the Gdev scheduler while waiting for the current context to leave the resources. In order to notify the completion of the current context execution, Gdev uses additional GPU commands to generate an interrupt from the GPU. Upon every interrupt, the highest-priority context is dispatched to the GPU from the waiting queue. Computation and data transmission times are separately accumulated for resource accounting. For computations, we allow the same context to launch multiple compute instances simultaneously, and the total makespan from the first to the last instance is deemed as the computation time. PTask [24] and RGEM [14] also provide similar schedulers, but do not use interrupts, and hence resource accounting is managed by the user space via the API.

Gdev is API-driven where the scheduler is invoked only when computation or data transmission requests are submitted, whereas TimeGraph is command-driven, which invokes the scheduler whenever GPU commands are flushed. In this regard, Gdev is similar to PTask [24] and RGEM [14]. However, Gdev differs from these two approaches in that it can separate queues for accounting of computations and data transmissions, which we call the *Multiple Resource Queues* (MRQ) scheme. On the other hand, what we call the *Single Device Queue* (SDQ) scheme uses a single queue per device for accounting.

The MRQ scheme is apparently more efficient than the SDQ scheme, when computations and data transmissions can be overlapped. Suppose that there are two contexts both requesting 50% of computation and 50% of data transmission demands. The SDQ scheme presumes that the demand of each context is 50 + 50 = 100%, implying a total demand of 200% by the two contexts. As a result, this workload looks overloaded under the SDQ scheme. The MRQ scheme, on the other hand, does not consider the total workload to be overloaded due to overlapping but each resource to be fully utilized.

Gdev creates two different scheduler threads to control the resource usage of the GPU compute unit and DMA unit separately. The compute scheduler thread is invoked by GPU interrupts generated upon the completion of each GPU compute operation, while the DMA scheduler thread is awakened by the Gdev runtime system when the memory-copy operation is completed, since we do not use interrupts for memory-copy operations.

`vgpu->bgt`: budget of the virtual GPU.
`vgpu->utl`: actual GPU utilization of the virtual GPU.
`vgpu->bw`: bandwidth assigned to the virtual GPU.
`current/next`: current/next virtual GPU selected for run.

```
void on_arrival(vgpu, ctx) {
  if (current && current != vgpu)
    suspend(ctx);
  dispatch(ctx);
}
VirtualGPU on_completion(vgpu, ctx) {
  if (vgpu->bgt < 0 && vgpu->utl > vgpu->bw)
    move_to_queue_tail(vgpu);
  next = get_queue_head();
  if (!next) return null;
  if (next != vgpu && next->utl > next->bw) {
    wait_for_short();
    if (current) return null;
  }
  return next;
}
```

Figure 2: Pseudo-code of the BAND scheduler.

5.2 GPU Virtualization

Gdev is able to virtualize a physical GPU into multiple logical GPUs to protect working groups of multi-tasking systems from interference. Virtual GPUs are activated by specifying weights of GPU resources assigned to each of them. GPU resources are classified to *memory share*, *memory bandwidth*, and *compute bandwidth*. Memory share is the weight of physical memory available for the virtual GPU. Memory bandwidth is the amount of time in a certain period allocated for memory-copy operations using the virtual GPU, while compute bandwidth is that for compute operations. Regarding memory share, Gdev simply partitions physical memory space. Meanwhile, we provide the GPU scheduler to meet the requirements of compute and memory-copy bandwidth. Considering similar characteristics of non-preemptive computations and data transmissions, we apply the same policy to the compute and memory-copy schedulers.

The challenge for virtual GPU scheduling is raised by the non-preemptive and burst nature of GPU workloads. We have implemented the Credit scheduling algorithm supported by Xen hypervisor [1] to verify if an existing virtual CPU scheduling policy can be applied for a virtual GPU scheduler. However, we have found that the Credit scheduler fails to maintain the desired bandwidth for the virtual GPU, largely attributed to the fact that it presumes preemptive constantly-working CPU workloads, while GPU workloads are non-preemptive and bursting.

To overcome the virtual GPU scheduling problem, we propose a *bandwidth-aware non-preemptive device* (BAND) scheduling algorithm. The pseudo-code of the

BAND scheduler is shown in Figure 2. The on_arrival function is called when a GPU context (ctx) running on a virtual GPU (vgpu) attempts to access GPU resources for computations or data transmissions. The context can be dispatched to the GPU only if no other virtual GPUs are using the GPU. Otherwise, the corresponding task is suspended. The on_completion function is called by the scheduler thread upon the completion of the context (ctx) assigned to the virtual GPU (vgpu), selecting the next virtual GPU to operate.

The BAND scheduler is based on the Credit scheduler, but differs in the following two points. First, the BAND scheduler lowers the priority of the virtual GPU, when its budget (credit) is exhausted *and* its actual utilization of the GPU is exceeding the assigned bandwidth, whereas the Credit scheduler always lowers the priority, when the budget is exhausted. This prioritization compensates for credit errors posed due to non-preemptive executions.

The second modification to the Credit scheduler is that the BAND scheduler waits for a certain amount of time specified by the system designer, if the GPU utilization of the virtual GPU selected by the scheduler is exceeding its assigned bandwidth. This "time-buffering" approach works for non-preemptive burst workloads. Suppose that the system has two virtual GPUs, both of which run some burst-workload GPU contexts, but their non-preemptive execution times are different. If the contexts arrive in turn, they are also dispatched to the GPU in turn, but the GPU utilization could not be fair due to different lengths of non-preemptive executions. If the scheduler waits for a short interval, however, the context with a short length of non-preemptive execution could arrive with the next request, and the on_arrival function can dispatch it to the GPU while the scheduler is waiting. Thus, resource allocations could become fairer. In this case, we need not to select the next virtual GPU, since the on_arrival function has already dispatched one. If no contexts have arrived, however, we return the selected virtual GPU. This situation implies that there are no burst workloads, and hence no emergency to meet the bandwidth.

6 System Implementation

Our prototype implementation is fully open-source and available for NVIDIA Fermi GPUs with the Linux kernel 2.6.33 or later, without any kernel modifications. It does not depend on proprietary software except for compilers. Hence, it is well self-contained and easy-to-use.

Interface: Gdev is a Linux kernel module (a character device driver) composing the device driver and runtime library. The device driver manages low-level hardware resources, such as channels and page tables, to operate the GPU. The runtime library manages GPU commands and API calls. It directly uses the device-driver functions to control hardware resource usage for first-class GPU resource management. The Gdev API is implemented in this runtime library. The kernel symbols of the API functions are exported so that other OS modules can call them. These API functions are also one-to-one mapped to the ioctl commands defined by Gdev so that user-space programs can also be managed by Gdev.

We provide two versions of CUDA Driver API: one for the user space and the other for the OS. The former is provided as a typical user-space library, while the latter is provided as a kernel module, called *kcuda*, which implements and exports the CUDA API functions. They however internally use Gdev API to access the GPU.

We use /proc filesystem in Linux to configure Gdev. For example, the number of virtual GPUs and their maps to physical GPUs are visible to users through /proc. The compute and memory bandwidth and memory share for each virtual GPU are also configurable at runtime through /proc. We further plan to integrate the configuration of priority and reserve for each single task into /proc, using the TimeGraph approach [15].

Gdev creates the same number of character device files as virtual GPUs, *i.e.*, /dev/{gdev0,gdev1,...}. When users open one of these device files using Gdev API or CUDA API, it behaves as if it were one for the physical GPU.

Resource Parameters: The performance of Gdev is governed by resource parameters, such as the page size for virtual memory, temporal swap size, waiting time for the Band scheduler, period for virtual GPU budgets, chunk size for memory-copy, and boundary between I/O access and DMA. We use a page size of 4KB, as the Linux kernel uses the same page size for host virtual memory by default. The swap size is statically set 10% of the physical device memory. The waiting time for the Band scheduler is also statically set 500 microseconds. For the period of virtual GPU budgets, we respect Xen's default setup, *i.e.*, we set it 30ms. The rest of resource parameters will be determined in Section 7.

Portability: We use Direct Rendering Infrastructure (DRI) [18] – a Linux framework for graphics rendering with the GPU – to communicate with the Linux kernel. Hence, some Gdev functionality may be used to manage not only compute but also 3-D graphics applications. Our implementation approach also abstracts GPU resources by device, address space, context, and memory objects, which allows other device drivers and GPU architectures to be easily ported.

Limitations: Our prototype implementation is still partly experimental. In particular, it does not yet support texture and 3-D processing. Hence, our CUDA Driver API implementation is limited to some extent, but many CUDA programs can execute with this limited set of functions, as we will demonstrate in Section 7. CUDA Runtime API [21], a more high-level API than CUDA

Driver API, is also not supported yet, but we could use Ocelot [5] to translate CUDA Runtime API to CUDA Driver API. Despite such limitations, we believe that our prototype implementation contributes greatly to future research on GPU resource management, given that open-source drivers/runtimes for GPUs are very limited today.

7 Experimental Evaluation

We evaluate our Gdev prototype, using the Rodinia benchmarks [3], GPU-accelerated eCryptfs encrypted file system from KGPU [28], FAST database search [16], and some dataflow microbenchmarks from PTask [24]. We disclose that the basic performance of our prototype is practical even compared to proprietary software, and also demonstrate that Gdev provides significant benefits for GPU applications in time-sharing systems.

Our experiments are conducted with the Linux kernel 2.6.39 on NVIDIA GeForce GTX 480 graphics card and Intel Core 2 Extreme QX9650 processor. GPU programs are written in CUDA and compiled by NVCC [21], while CPU programs are compiled by gcc 4.4.6.

7.1 Basic Performance

We evaluate the standalone performance of applications achieved by our Gdev prototype to argue that the rest of our evaluation is practical in the real world. To this end, first of all, we need to find the best parameters used for memory-copy optimization, using simple test code that copies data between device and host memory.

Figure 3 shows the impact of the chunk size on data transfer times for host-to-device (HtoD) and device-to-host (DtoH) directions respectively, when using DMA-based memory-copy operations with 256MB and 512MB of data. Since each chunk incurs some overhead in DMA configuration, a smaller chunk size producing a greater number of chunks increases a transfer time. On the other hand, there is a constraint that the first and last pieces of chunks cannot be overlapped with others, as described in Section 4.1. Hence, a larger chunk size leading to a longer blocking time with these pieces of chunks also increases a transfer time. According to our observation, a chunk size of 4MB is the best trade-off for both HtoD and DtoH directions. We therefore set the chunk size to 4MB for our experiments.

Figure 4 shows the relative speed of direct I/O access to DMA for a small size of data. Due to some hardware effect, HtoD and DtoH directions show different transfer times, but it clearly explains the advantage of direct I/O access for small data. According to our observation, the data transfer speed inverses around a data size of 4KB and 1KB for HtoD and DtoH directions respectively. We therefore set the boundary of direct I/O access and DMA to 4KB and 1KB for them respectively.

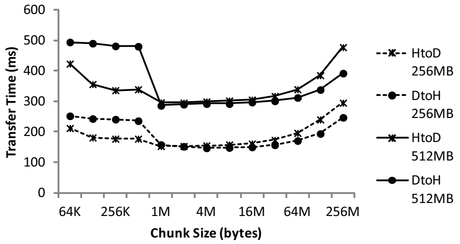


Figure 3: Impact of the chunk size on DMA speeds.

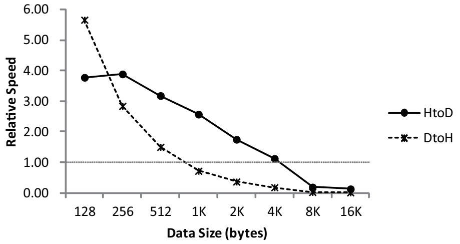


Figure 4: Relative speed of I/O access to DMA.

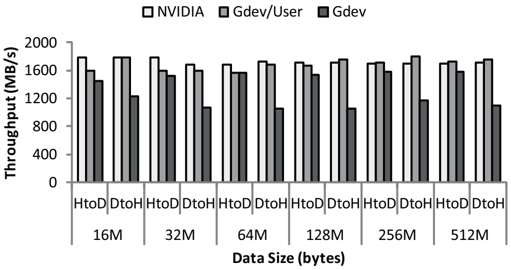


Figure 5: Memory-copy throughput.

Figure 5 shows memory-copy throughput achieved by our Gdev prototype compared to NVIDIA's proprietary software. "Gdev/User" employs a runtime library in the user-space, while "Gdev" integrates runtime support into the OS. Interestingly, user-space runtime achieves higher throughput than OS-space runtime, particularly for DtoH direction. This difference comes from `memcpy`'s effect within host memory. In fact, the `memcpy` implementation in the Linux kernel is slower than that in the user-space GNU library, when copying data from host I/O to main memory. This could be a disadvantage of our approach. We will investigate this effect more in depth. Apart from the DtoH memory-copy throughput, however, our Gdev prototype and NVIDIA's proprietary software are almost competitive.

Figure 6 demonstrates the standalone performance of benchmarks achieved by our Gdev prototype compared to NVIDIA's proprietary software. Table 1 describes the microbenchmarks and Rodinia [3] benchmarks used in this evaluation. First of all, we have found that NVIDIA GPUs have some "performance mode" to boost hardware performance that we do not use for our Gdev prototype implementation. As observed in the LOOP benchmark result, our Gdev prototype incurs about 20% of decrease

Table 1: List of benchmarks.

Benchmark	Description
LOOP	Long-loop compute without data
MADD	1024x1024 matrix addition
MMUL	1024x1024 matrix multiplication
CPY	256MB of HtoD and DtoH
PINCPY	CPY using pinned host I/O memory
BP	Back propagation (pattern recognition)
BFS	Breadth-first search (graph algorithm)
HW	Heart wall (medical imaging)
HS	Hotspot (physics simulation)
LUD	LU decomposition (linear algebra)
NN	K-nearest neighbors (data mining)
NW	Needleman-wunsch (bioinformatics)
SRAD	Speckle reducing anisotropic diffusion (imaging)
SRAD2	SRAD with random pseudo-inputs (imaging)

in performance compared to the proprietary software due to a lack of performance mode. However, the impact of performance mode is workload dependent. If workloads are very compute-intensive, such as the HW and SRAD benchmarks, this impact appears clearly, whereas some friendly workloads, such as the BFS and HS benchmarks, are not influenced very much. In either case, however, this is an implementation issue, but is not a conceptual limitation of Gdev. These benchmark results also imply that Gdev's runtime-unified OS approach would not be appreciated by data-intensive workloads. For example, the BP benchmark is associated with a very large size of data, though its compute demand is not very high. This type of workload would not perform well with our Gdev prototype, since the `memcpy` function of the Linux kernel becomes a bottleneck. On the other hand, the PINCPY benchmark does not need `memcpy` operations. Hence, performance does not depend on implementations.

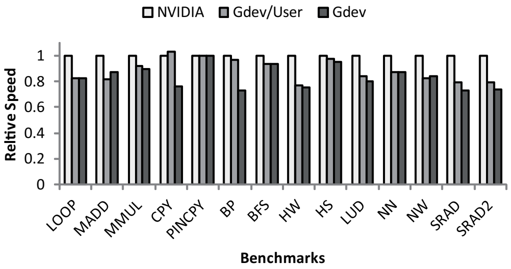


Figure 6: Basic standalone performance.

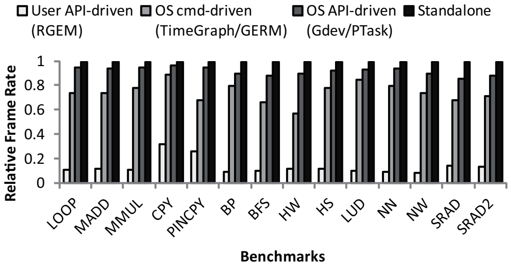


Figure 7: Unconstrained real-time performance.

7.2 Reliability

We next evaluate reliability of runtime support integrated in the OS. Figure 7 compares the performances of the OS-space API-driven scheme (Gdev and PTask [24]), the OS-space command-driven scheme (TimeGraph [15] and GERM [2]), and the user-space API-driven scheme (RGEM [14]). We run Rodinia benchmarks recursively as fast as possible as real-time tasks, contending with such background tasks that bypass the user-space library and launch many meaningless GPU commands. The user-space API-driven scheme severely suffers from this situation, since it cannot schedule these bypassing tasks at all. The OS-space command-driven scheme is able to sustain the interference to some extent by using the GPU command scheduler, but the overhead is non-trivial due to many scheduler invocations. On the other hand, the OS-space API-driven scheme can reject such command submission that is not submitted through the API. Gdev and PTask are both API-driven, but PTask exposes the system call to user-space programs, which could allow misbehaving tasks to abuse GPU resources. This evinces that Gdev's approach that integrates runtime support into the OS is a more reliable solution.

7.3 GPU Acceleration for the OS

We now evaluate the performance of the Linux encrypted file system accelerated by the GPU. In particular, we use KGPU's implementation of eCryptfs [28]. KGPU is a framework that allows the OS to access the user-space runtime library to use GPUs for computations. We have modified KGPU's eCryptfs implementation to call the CUDA API functions provided by Gdev directly instead of sending requests to the KGPU user-space daemon.

Figure 8 and 9 show the read and write throughput of several versions of eCryptfs. "CPU" represents the CPU implementation, while "KGPU & NVIDIA" and "KGPU & Gdev/User" represent those using KGPU with NVIDIA's library and Gdev's library respectively in the user space. "Gdev" is our contribution that enables the eCryptfs module to use the GPU directly within the OS. Due to some page cache effect, read and write are not identical in throughput, but an advantage of using the GPU is clearly depicted. One may observe that Gdev's runtime-unified OS approach does not really outperform KGPU's approach. This is not surprising at all, because a magnitude of improvements in latency achieved by our OS approach would be at most microseconds, while the AES/DES operations of eCryptfs performed on the GPU are orders-of-milliseconds. Nonetheless, Gdev provides a significant benefit that the OS is freed from the user space, and thus is more secure.

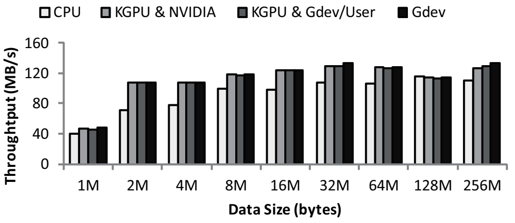


Figure 8: eCryptfs read throughput.

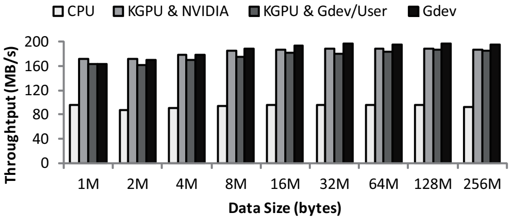


Figure 9: eCryptfs write throughput.

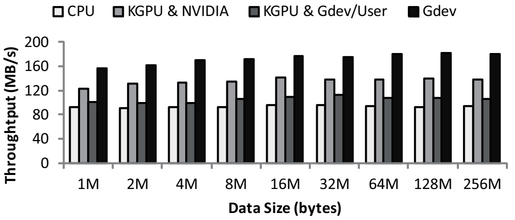


Figure 10: eCryptfs write throughput with priorities.

A further advantage of using Gdev appears in a multi-tasking scenario. Figure 10 shows the write throughput of eCryptfs when the FAST search task [16] is competing for the GPU. Since Gdev supports priorities in the OS, we can assign the eCryptfs task with the highest priority, while the search task is still assigned a higher priority than other tasks. Using KGPU in this scenario, however, the performance of eCryptfs is affected by the search task due to a lack of prioritization, as observed in "KGPU & NVIDIA". Even with priorities, KGPU could suffer from a priority inversion problem, where the high-priority eCryptfs task is reduced to the KGPU priority level when accessing the GPU, while the search task is executing at the higher priority level. We could assign a high priority to the user-space KGPU daemon to avoid this priority inversion problem, but it affects all user-space GPU applications performance. On the other hand, Gdev can assign each GPU application with an identical priority, which addresses the priority inversion problem fundamentally.

7.4 Effect of Shared Device Memory

Figure 11 shows the speedups of dataflow benchmarks brought by Gdev's shared device memory functionality. Respecting PTask's setup [24] for a similar evaluation, we make a dataflow by a 6x32 tree or a 6x10 rectangle.

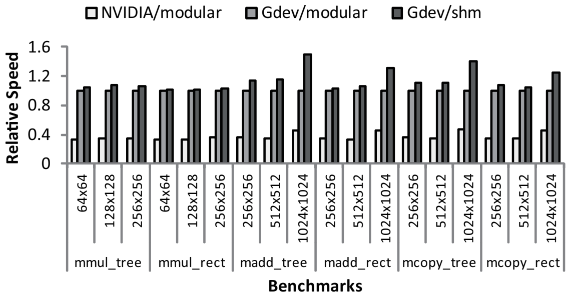


Figure 11: Impact of shared memory on dataflow tasks.

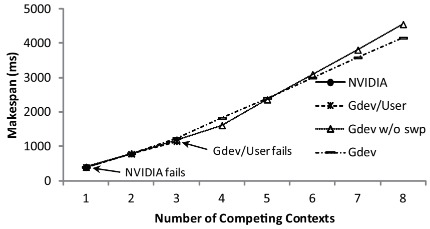


Figure 12: Impact of swapping latency.

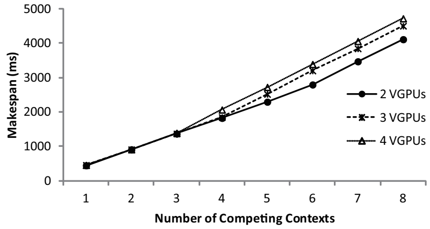
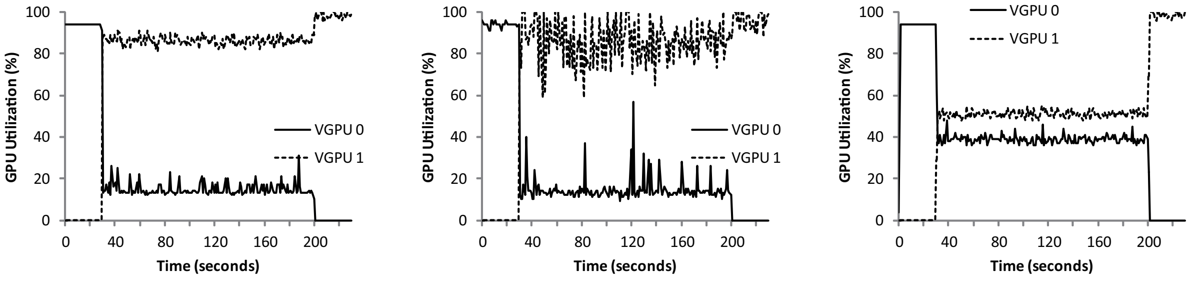


Figure 13: Impact of swapping latency on virtual GPUs.

"NVIDIA/modular" and "Gdev/modular" use NVIDIA's and Gdev's CUDA implementations respectively, where a dataflow program is implemented in such a way that allocates a self-contained context to each graph node as a module, and connects its output and input by copying data between host and device memory back and forth. "Gdev/shm" uses shared device memory, *i.e.*, it connects output and input by sharing the same "key" associated with the same memory space. According to the results, shared device memory is fairly effective for dataflows with large data. For example, it gains a 49% speedup for the 1024x1024 madd tree. Specifically, "Gdev/modular" took 1424ms while "Gdev/shm" took 953ms to complete this dataflow. This indeed makes sense. The average data transfer time for a 1024x1024 integer value was about 8ms, and we can reduce data communications by a total of 32+16+8+4+2=62 intermediate nodes for a 6x32 tree, which results in a total reduced time of 8x62=496ms. It should be noted that PTask achieves more speedups due to advanced dataflow scheduling [24]. However, we provide users with a first-class API primitive to manage shared device memory, which could be used as a generic IPC method to address different problems. Therefore, we distinguish our contribution from PTask. In addition, it



(a) FIFO scheduler

(b) Credit scheduler

(c) BAND scheduler

Figure 14: Util. of virtual GPUs under unfair workloads.

is surprising that our prototype system outperforms the proprietary software significantly. We suspect that the proprietary one takes a long time to initialize contexts when there are many active contexts, though an in-depth investigation is required.

Figure 12 depicts the impact of memory swapping on the makespan of multiple 128MB-data FAST search tasks, when another 1GB-data FAST search task runs at the highest priority level. Given that the GPU used in this evaluation supports 1.6GB of device memory, we cannot create more than three 128MB-data search tasks at once if memory swapping is not provided. Memory swapping uses shared device memory, which needs access to the page table. Hence, our prototype implementation does not support memory swapping as well as shared device memory for "Gdev/User", and it fails when the number of the small search tasks exceeds three. It is interesting to observe that NVIDIA' proprietary software fails when the number of the small search tasks exceeds one. This is because NVIDIA's proprietary software reserves some amount of device memory for other purposes. Unlike the user-space runtime approaches, Gdev's runtime-unified OS approach can support memory swapping, and all the 128MB-data search tasks can survive under this memory pressure. However, a reflection point where the slope of increase in the makespan changes is different, depending on whether the temporal swap space allocated on device memory is used or not. When the temporal swap space is not used, the reflection point is clearer as observed in "Gdev w/o swp", because the swapping latency is not trivial due to data movement between host and device memory. Using the temporal swap space, on the other hand, we can reduce the impact of memory swapping on the makespan of the search tasks, but the reflection point appears slightly earlier, since the temporal swap space itself occupies certain space on device memory.

Figure 13 shows the impact of memory swapping on virtual GPUs. In this experiment, we introduce virtual GPUs, and execute 128MB-data search tasks on the first virtual GPU. The memory size available for the virtual GPU is more restricted in the presence of more virtual GPUs. We confirm that the makespans become longer and their reflection points appear earlier for a greater number of virtual GPUs, but all the search tasks can still complete. This explains that memory swapping is also useful on virtual GPUs.

7.5 Isolation among Virtual GPUs

We now evaluate Gdev in terms of the isolation among virtual GPUs. Figure 14 demonstrates the actual GPU utilization of two virtual GPUs, achieved by the FIFO, Credit, and BAND schedulers under the SDQ scheme. VGPU 0 executes the LUD benchmark to produce short-length tasks, while VGPU 1 executes the HW benchmark to produce long-length tasks. These tasks run repeatedly for 200 seconds to impose high workloads on the entire system. To see a workload change clearly, VGPU 1 is started 30 seconds after VGPU 0. Our observation is that the FIFO scheduler is not capable of enforcing isolation at all. The Credit scheduler also fails to provide isolation, since it is not designed to handle non-preemptive burst workload. The BAND scheduler, however, can almost provide the desired GPU utilization, thanks to the time-buffering policy that allows short-length tasks to meet the assigned bandwidth. An error in the utilization of two virtual GPUs is retained within 7% on average.

We next study the effectiveness of the MRQ scheme that separates the queues for compute and memory-copy operations. Figure 15 illustrates the utilization of two virtual GPUs under the BAND scheduler, executing the SRAD benchmark tasks with different sizes of image. We noticed that the compute and memory-copy operations can be overlapped, but they affect the run-to-completion time with each other. When VGPU 1 uses more compute resources due to a large size of computation, the length of memory-copy operations requested by VGPU 0 is prolonged due to overlapping. As a result, it requires more memory-copy bandwidth. However, the available bandwidth is capped by the BAND scheduler,

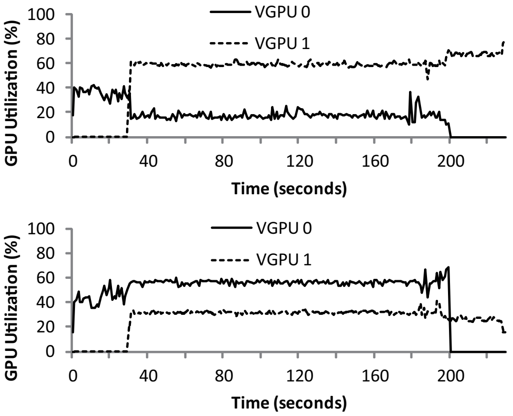


Figure 15: Util. of virtual GPUs with the MRQ scheme (upper for compute and lower for memory-copy).

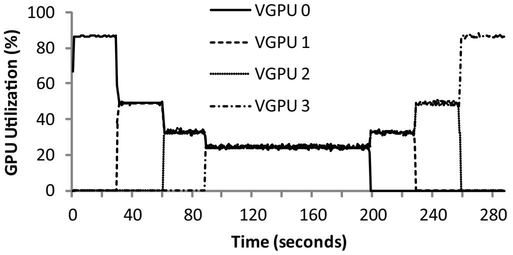


Figure 16: Util. of virtual GPUs under fair workloads.

i.e., both the compute and memory-copy operations are limited to about 50% of bandwidth at most. One can also observe that the MRQ scheme allowed the sum of compute and memory-copy bandwidth to exceed 100%.

We finally demonstrate the scalability of our virtual GPU support. Figure 16 shows the utilization of four virtual GPUs under the BAND scheduler, where all virtual GPUs execute four instances of the LUD benchmark task exhaustively to produce fair workloads. The workloads of each virtual GPU begin in turn at an interval of 30 seconds. Under such sane workloads, our virtual GPU support can provide fair bandwidth allocations, even if the system exhibits non-preemptive burst workloads.

8 Related Work

GPU Resource Management: TimeGraph [15] and GERM [2] provide a GPU command-driven scheduler integrated in the device driver. Specifically, TimeGraph achieves a prioritization and isolation scheme, extended with a resource sharing scheme [13] later on, whereas GERM enhances fair-share GPU resource control. Gdev also respects a prioritization, isolation, and fairness scheme, similar to TimeGraph and GERM, but adopts an API-driven scheduler model to reduce the number of scheduler invocations. The overhead of the command-driven scheduler model was also discussed in [15].

PTask [24] is an OS abstraction for GPU applications that optimizes data transfers between host and device memory using a data-flow programming model and a GPU scheduler. CGCM [11] is another solution based on the compiler and runtime library that dynamically and automatically optimizes the same sort of data trasfers. In contrast, Gdev does not support data-flow programming or automatic code generation. Alternatively, it provides programmers with an explicit API set to share device memory among GPU contexts. Such an IPC scheme can similarly reduce data transfer overhead.

RGEM [14] is a user-space runtime model for real-time GPGPU applications. It creates preemption points with data transfers between host and device memory in order to bound blocking times imposed on high-priority tasks. It also provides separate queues to demultiplex the schedulings of data transfers and computations. Albeit using a similar separate-queue scheme, Gdev addresses a core challenge of GPU resource management integreated in the OS to overcome the user-space limitations.

In addition to the aforementioned differences, Gdev can virtualize the GPU in the OS, which enables users to view a physical GPU as multiple logical GPUs for strong resource isolation. None of the previous work has also devided compute and memory bandwidth reservations, whereas Gdev accounts for these bandwidth reservations independently to maximize the overall GPU utilization. Furthermore, the previous work depend more or less on proprietary software or existing software stack, which could force design and implementation, if not concept, to adhere to user-space runtime libraries. Our prototype design and implementation of Gdev, in contrast, is fully self-contained, allowing the OS to fully control and even use GPUs as first-class computing resources.

GPUs as OS Resources: A significant limitation on the current GPU programming framework is that GPU applications must reside in the user space. KGPU [28] is a combination of the OS kernel module and user-space daemon, which allows the OS to use GPUs by up-calling the user-space daemon from the OS to access the GPU. On the other hand, Gdev provides OS modules with a set of traditional API functions for GPU programming, such as CUDA. Hence, a legacy GPU application program can execute in the OS, as it is, without any modifications and additional communications between the user space and the OS. In addition, we have shown that runtime support integrated in the OS is more reliable.

GPU Virtualization: VMGL [17] virtualizes GPUs at the OpenGL API level, and VMware's Virtual GPU [6] exhibits I/O virtualization through graphics runtimes. On the other hand, Pegasus [9] uses a hypervisor, Xen [1] in particular, to co-schedule GPUs and virtual CPUs in VMs. Nonetheless, these virtualization systems rely on user-space runtimes provided by proprietary software, preventing the system from managing GPU resources in a fine-grained manner. In addition, they are mainly designed to make GPUs available in virtualized environments, but are not tailored to isolate GPU resources among users. Gdev provides virtual GPUs with strong time and space partitioning, and hence could underlie these GPU virtualization systems.

I/O Scheduling: GPU scheduling deals with a non-preemptive nature of execution as well as traditional I/O scheduling. Several disk bandwidth-aware schedulers [8, 22, 30], for example, contain a similar idea to the Gdev scheduler. Unlike typical I/O devices, however, GPUs are coprocessors operating asynchronously with own sets of execution contexts, registers, and memory. Therefore, Gdev adopts a scheduling algorithm more appropriate for compute-intensive workload.

Compile-Time and Application Approaches: GPU resources can also be managed by application programs without using drivers and libraries [4, 7, 26]. However, these approaches essentially need to modify or recompile the programs using specific compilers and/or algorithms. Thus, a generality of programming frameworks need to be compromised. In contrast, Gdev allows applications to use traditional GPU programming frameworks.

9 Conclusion

This paper has presented Gdev, a new approach to GPU resource management that integrates runtime support into the OS. This runtime-unified OS approach realizes new memory management and scheduling schemes that enable a wide class of applications to GPUs as first-class computing resources in general-purpose multi-tasking systems. We implemented a prototype system of Gdev, and conducted thorough experiments to demonstrate the advantage and disadvantage of using our Gdev approach. Our conclusion is that Gdev needs to compromise some basic performance due to incorporating runtime support in the OS, but can enhance GPU resource management for multi-tasking systems and allow the OS itself to use GPUs for computations.

Our prototype system and application programs used in the performance evaluation are all open-source, and may be downloaded from our website [25].

References

[1] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. In *Proc. of ACM Symposium on Operating Systems Principles* (2003).

[2] Bautin, M., Dwarakinath, A., and Chiueh, T. Graphics engine resource management. In *Proc. of Annual Multimedia Computing and Networking Conference* (2008).

[3] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J., Lee, S.-H., and Skadron, K. Rodinia: A benchmark suite for heterogeneous computing. In *Proc. of IEEE International Conference on Workload Characterization* (2009), pp. 44–54.

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

[5] Diamos, G., Kerr, A., Yalamanchili, S., and Clark, N. Ocelot: A dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *Proc. of ACM International Conference on Parallel Architectures and Compilation Techniques* (2010), pp. 353–364.

[6] Dowty, M., and Sugeman, J. GPU virtualization on VMware's hosted I/O architecture. *ACM Operating Systems Review 43*, 3 (2009), 73–82.

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

[8] Gulati, A., Ahmad, I., and Waldspurger, C. PARDA: Proportional allocation of resources for distributed storage access. In *Proc. of USENIX Conference on File and Storage Technology* (2009).

[9] Gupta, V., Schwan, K., Tolia, N., Talwar, V., and Ranganathan, P. Pegasus: Coordinated scheduling for virtualized accelerator-based systems. In *Proc. of USENIX Annual Technical Conference* (2011).

[10] Hand, S., Jang, K., Park, K., and Moon, S. PacketShader: a GPU-accelerated software router. In *Proc. of ACM SIGCOMM* (2010).

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

[12] Jang, K., Han, S., Han, S., Moon, S., and Park, K. SSLShader: Cheap SSL acceleration with commodity processors. In *Proc. of USENIX Conference on Networked Systems Design and Implementation* (2011).

[13] Kato, S., Lakshmanan, K., Ishikawa, Y., and Rajkumar, R. Resource sharing in GPU-accelerated windowing systems. In *Proc. of IEEE Real-Time and Embedded Technology and Aplications Symposium* (2011), pp. 191–200.

[14] Kato, S., Lakshmanan, K., Kumar, A., Kelkar, M., Ishikawa, Y., and Rajkumar, R. RGEM: A responsive GPGPU execution model for runtime engines. In *Proc. of IEEE Real-Time Systems Symposium* (2011), pp. 57–66.

[15] Kato, S., Lakshmanan, K., Rajkumar, R., and Ishikawa, Y. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *Proc. of USENIX Annual Technical Conference* (2011).

[16] Kim, C., Chhugani, J., Satish, N., Sedlar, E., Nguyen, A., Kaldewey, T., Lee, V., Brandt, S., and Dubey, P. FAST: Fast architecture sensitive tree search on modern CPUs and GPUs. In *Proc. of ACM International Conference on Management of Data* (2010).

[17] Lagar-Cavilla, H., Tolia, N., Satyanarayanan, M., and de Lara, E. VMM-independent graphics acceleration. In *Proc. of ACM/Usenix International Conference on Virtual Execution Environments* (2007), pp. 33–43.

[18] Martin, K., Faith, R., Owen, J., and Akin, A. *Direct Rendering Infrastructure, Low-Level Design Document.* Precision Insight, Inc., 1999.

[19] McNaughton, M., Urmson, C., Dolan, J., and Lee, J.-W. Motion Planning for Autonomous Driving with a Conformal Spatiotemporal Lattice. In *Proc. of IEE International Conference on Robotics and Automation* (2011), pp. 4889–4895.

[20] NVIDIA. NVIDIA's next generation CUDA compute architecture: Fermi. `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`, 2009.

[21] NVIDIA. CUDA 4.0. `http://developer.nvidia.com/cuda-toolkit-40`, 2011.

[22] Povzner, A., Kaldewy, T., Brandt, S., Golding, R., Wong, T., and Maltzahn, C. Efficient guaranteed disk request scheduling with Fahrrad. In *Proc. of ACM European Conference on Computer Systems* (2008), pp. 13–25.

[23] Pronovost, S., Moreton, H., and Kelley, T. Windows Display Driver Model (WDDM v2 and beyond. Windows Hardware Engineering Conference, 2006.

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

[25] S. Kato. Gdev Project. `http://sys.ertl.jp/gdev/`, 2012.

[26] Saba, A., and Mangharam, R. Anytime Algorithms for GPU Architectures. In *Proc. of IEEE Real-Time Systems Symposium* (2011).

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

[28] Sun, W., Ricci, R., and Curry, M. GPUstore: Harnessing GPU Computing for Storage Systems in the OS Kernel.

[29] TOP500 Supercomputing Site. `http://www.top500.org/`, 2011.

[30] Wang, Y., and Merchant, A. Proportional-share scheduling for distributed storage systems. In *Proc. of USENIX Conference on File and Storage Technology* (2007).