Implementing Open-Source CUDA Runtime

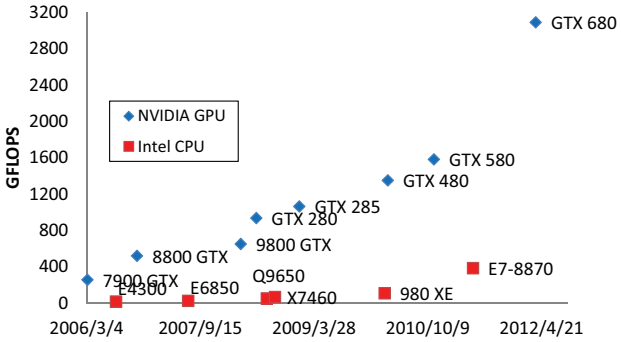
Shinpei Kato
Department of Information Engineering, Nagoya University
Furo-cho, Chikusa-ku, Nagoya 464-8603, JAPAN
shinpei@is.nagoya-u.ac.jp

Abstract

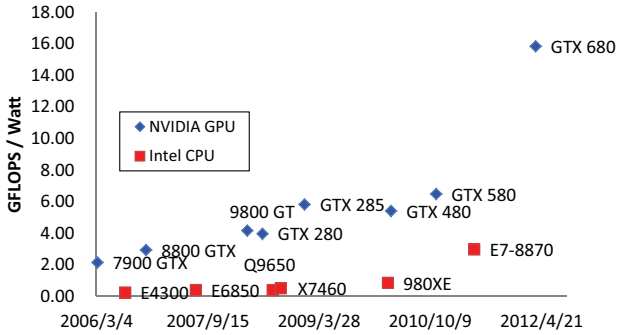
Graphics processing units (GPUs) are the state of the art embracing the concept of many-core technology. Their significant advantage in performance and performance-per-watt compared to traditional microprocessors has facilitated development of GPUs in many compute applications. However, GPUs are often treated as "black-box" devices due to proprietary strategies of hardware vendors. One of the greatest challenges of this research domain is the in-depth understanding of GPU architectures and runtime mechanisms so that the systems research community can tackle fundamental problems of GPUs. In this paper, we present an open-source implementation of CUDA runtime, which is the most widely-recognized programming framework for GPUs, as well as a documentation of "how GPUs work" investigated by our reverse engineering work. Our implementation is based on Linux and is targeted at NVIDIA GPUs.

1 Introduction

Graphics processing units (GPUs) have become very powerful platforms in the parallel computing market, best embracing the concept of many-core technology. Application domains of GPUs now spread across embedded and high-performance computing. Examples include autonomous vehicles [17], software routers [11], encrypted networks [12], storage and file systems [7, 9, 19], and a plenty of scientific applications. Such a rapid growth of GPUs is brought by recent advances in programming languages and hardware architectures. CUDA [1] and OpenCL [3] are particularly powerful programming frameworks for general-purpose computation on GPUs, *a.k.a.*, GPGPU, which facilitated deployment of GPUs in many compute applications. Thanks to the emergence of GPGPU techniques, GPUs are getting more and more generalized for compute workload. Peak performance of GPUs in the current state of the art exceeds 3 TFLOPS, integrating more than 1,500 cores on a single chip, which is nearly equivalent of 19 times that of traditional microprocessors such as Intel *Core i* series. Trends on the performance and the performance-per-watt of well-known NVIDIA GPUs and Intel CPUs depicted in Figure 1 explain an advantage of massively parallel computing achieved by GPUs.



(a) Single-precision performance.



(b) Single-precision performance-per-watt.

Figure 1. Trends on NVIDIA GPUs and Intel CPUs.

Currently the most common use of GPUs is to accelerate a particular application program dedicated to the system such as a game or scientific simulation. In this scenario,

stand-alone performance is a major concern, and there is no need of multiprogramming with GPUs. This is why current systems can meet requirements of applications using black-box proprietary driver and runtime software: the problem definition is often made at the programming level rather than the system level. However recent work in the systems research community have argued that GPUs could be first citizens in the system such as CPUs, if multiprogramming is well-supported [10, 15, 16, 18]. There are many applications that benefit from this paradigm shift. It is high time to start looking into system-level approaches to GPU computing. What is required to this end is the in-depth understanding of GPU architectures and runtime mechanisms. Unfortunately there are very limited documentations of GPU architecture and open-source software for runtime systems. This is due to proprietary production of GPUs in the market.

In this paper, we present an open-source implementation of CUDA runtime as well as a documentation of "how GPUs work". We restrict our attention to Linux, CUDA, and NVIDIA GPUs, because they are the most recognized environments for systems researchers and developers. Our documentation is based on the reverse engineering work conduced by the Linux open-source community, while our implementation is made from scratch. We believe that this paper contains many pieces of useful information to address open problems of GPU computing.

The rest of this paper is organized as follows. Section 2 describes the current GPU computing model with the system details. Section 3 presents our open-source implementation of CUDA runtime. Section 4 provides a summary of this paper.

2 GPU Computing Model

The GPU is a compute device designed to accelerate a particular function of the program often referred to as a compute kernel. This kernel may contain a number of threads that execute in parallel on the GPU using a massive set of compute cores. The mapping of threads and cores is typically managed by hardware, while it is programmer's responsibility to parallelize the kernel allocating a suitable set of threads. In case that the program launches a kernel multiple times, performance of the program is governed by the programmer, because the system has to pay some overhead to offload a kernel to the GPU and each kernel likely requires the data set to be copied between the device and the host memory, which can be an expensive operation. These programming issues have been extensively studied in the literature.

We focus on system issues of GPU computing. Currently hardware vendors enclose implementations of the device driver and the runtime library in proprietary binary software whereas the compiler source code has been open-released to a limited extent. Systems research on GPU computing therefore is forced to build a solution on top of black box software. This limits the scope of systems research and prevents us from tacking fundamental problems of operating systems and system software.

Recently the Linux kernel community has developed *Nouveau* [2], an open-source device driver for NVIDIA GPUs. This is part of the unified GPU computing framework of the Linux kernel called the *direct rendering module* (DRM) [8]. The development of Nouveau is encouraged by reverse engineering of black box hardware engines. Albeit partly limited functionality, Nouveau is now reliable enough to underlie high-quality research on operating systems and system software [13, 14, 15, 16].

In the reminder of this section, we explain how GPUs actually work based on the reverse-engineered information of NVIDIA GPUs. Understanding that leads to being able to develop the device driver and the runtime library. For simplicity of description, we assume the Fermi architecture [4], but the following documentation is also useful and even applicable to different architectures such as Kepler [6].

2.1 PCI BARs

The current form of the GPU is a PCI device. Although on-chip GPUs are emerging, we focus on off-chip GPUs as graphics cards and assume that all communications between the GPU and the CPU are via the PCI bus. The NVIDIA GPU exposes the following base address registers (BARs) to the system through PCI in addition to the PCI configuration space and VGA-compatible I/O ports.

BAR0 Memory-mapped I/O (MMIO) registers.

BAR1 Device memory aperture (windows).

BAR2/3 I/O port or complementary space of BAR1.

BAR5 I/O port.

BAR6 PCI ROM.

The most significant area is the BAR0 presenting MMIO registers. This is the main control space of the GPU, through which all hardware engines are controlled. Its space is sparsely populated with areas representing individual hardware engines, which in turn are sparsely populated with control registers. The list of hardware engines is architecture-dependent.

2.2 MMIO

The MMIO registers are 32 bits long. They are of course accessible to both the GPU and the CPU. From the engineering point of view, a particularly important subarea

Table 1. Fermi's MMIO regions.

000000:001000	master control
001000:002000	bus control
002000:004000	channel control
007000:008000	access to BAR0 from real mode
009000:00a000	time measurement and timers
00e000:00e800	GPIOs, I2C buses, PWM fan control
020000:021000	thermal sensor
022400:022800	control over enabled units
040000:060000	subchannel control
060000:061000	indirect virtual memory access
070000:071000	to flush BAR writes
084000:085000	VP3 BSP
085000:086000	VP3 video decoding
086000:087000	VP3 video postprocessing
088000:089000	PCI configuration space access
104000:105000	memory copy control #1
105000:106000	memory copy control #2
106000:107000	memory copy control #3
108000:108800	HDA codec (HDMI audio)
109000:10a000	efuses storing secret key stuff
10a000:10b000	background daemon process
10f000:120000	memory controller backends
137000:138000	clock setting
139000:13b000	peer to peer memory access
13b000:13f000	crossbar b/w memory and GPCs
140000:180000	compression and L2 cache
180000:1c0000	performance monitoring counters
1c2000:1c3000	H.264 video encoding
200000:201000	mediaport
300000:380000	ROM access window
400000:600000	2-D/3-D drawing and compute
610000:6c0000	display
700000:800000	indirect device/host memory access
800000:810000	channel table

of the MMIO registers is the master control engine. This subarea is present on all NVIDIA GPUs at particular addresses. It contains the chipset information, the registers to activate/deactivate each hardware engine, and the controller of top-level interrupt routing.

The most relevant subareas to resource management of compute applications include the channel engine and the compute engine. The channel engine maintains the states of GPU contexts including FIFO queues of GPU commands, while the compute engine executes parallel threads. They are controlled through the particular regions of MMIO. For reference, Table 1 shows representative subareas of MMIO and their role. The details of MMIO are outside the scope of this paper.

2.3 GPU Context

Resource management of the GPU is context-based. To use the GPU for computation, the program must create a context. The context information is constructed through the MMIO registers. The first thing to do is to allocate memory space for the page directory, the page tables, and the channel descriptor of the context. They are referenced by physical device memory addresses. The page directory address must be written to a particular entry of the channel descriptor, while the channel descriptor address must be written to a particular MMIO register. There is an upper bound on the number of allocatable channels. For example, the Fermi architecture supports 128 channels at most. This means that there are 128 fixed entries of the channel descriptor in the MMIO channel control region shown in Table 1.

Now we can allocate virtual memory space within the context. Note that recent NVIDIA GPUs support unified memory addressing (UMA). All memory objects allocated to the device and the host memory can be referenced by the same address space. This is due to the graphics address remapping table (GART) employed by the GPU. We can specify physical host memory addresses directly in the GPU page table as far as they are associated with PCI-mapped pages. The virtual memory space is used by both the user program and the system. For example, the user program references the code and data by virtual memory addresses, while the system maintains GPU *command buffers* in the same virtual memory address space.

Typically the GPU is controlled by the CPU using some *commands*. There are hundreds of commands defined by the architecture. For example, when we copy data from the host to the device memory, we send a set of commands to the GPU, specifying the source and the destination virtual addresses together with the mode of direct memory access (DMA). Similarly when we launch a kernel, we compose another set of commands to the GPU, specifying the code and stack information. Each command is 32 bits long with a specific format. As aforementioned, the system maintains GPU command buffers. The CPU writes the commands to them, while the GPU reads the commands from them. To do so, the system must allocate them so as to be accessible to both the CPU and the GPU. This can be done by one of the following two methods.

1. Allocate device memory space, and map this space to the PCI BARs so that the CPU can access it.
2. Allocate host memory space, and have the GPU access it through UMA.

Figure 2 illustrates a block diagram of the GPU context management model. Assuming that the code and data is already placed on the virtual memory address space of

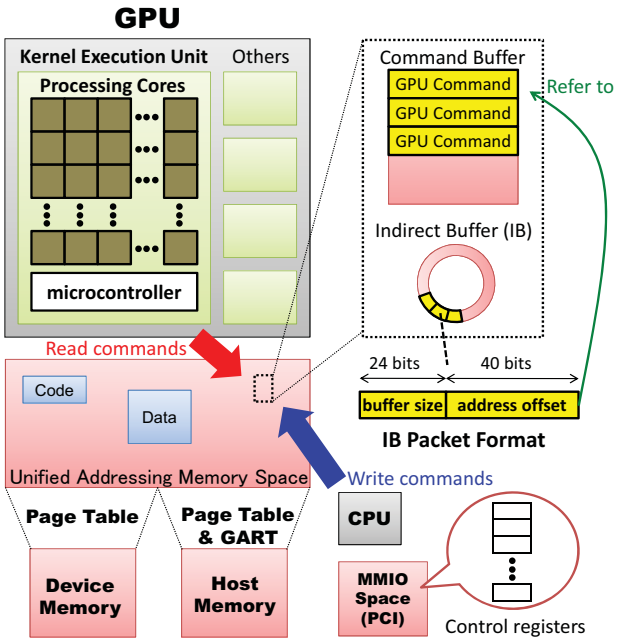


Figure 2. The GPU context management model.

the corresponding context, this model explains how to operate the GPU. First, the CPU writes GPU commands to the command buffer allocated by the system a priori. Although old GPU architectures were designed to use this command buffer directly, recent GPU architectures provide another ring buffer called the index buffer to simplify the mechanism of command dispatching. When the commands are written to the command buffer, the system writes *packets*, each of which is a (*size* and *address*) tuple to locate the corresponding GPU commands, into the index buffer. The GPU reads this index buffer instead of the command buffer and dispatches the GPU commands pointed to by the packets. The index buffer is controlled by `GET` and `PUT` pointers. The pointers start from the same place. Every time packets are written to the buffer, the system moves the `PUT` pointer to the tail of the packets via a particular MMIO register. The GPU always dispatches the GPU commands pointed to by the packets between the `GET` and `PUT` pointers. The `GET` pointer is then automatically updated to the same place as the `PUT` pointer.

3 Open-Source CUDA Runtime

We now present an open-source implementation of CUDA runtime. There are two types of API in CUDA. The CUDA Driver API provides a low-level set of functions while the CUDA Runtime API is very high-level with compiler support. We consider the CUDA Driver API in this section, and the CUDA Runtime API must be able to build on it. The main functions of the CUDA Driver API are that (i) create and destroy a context, (ii) load and unload the binary image (module), (iii) allocate and free memory, (iv) copy data between the device and the host memory, (v) set parameters (kernel arguments), (vi) launch a kernel, and (vii) synchronize with the GPU. There are many other functions as well, but we exclude them for simplicity of description.

3.1 Frontend

In this section, we provide a high-level idea of how to implement the main functions of the CUDA Driver API.

Create Context This function creates a GPU context by configuring the MMIO registers as mentioned in Section 2.3. The list of heap memory is also initialized in this function. Another important thing to do is to enable compute engines and memory copy engines within this context. This is done by sending a specific set of commands to the GPU.

Destroy Context This function kills the specified context by clearing the MMIO registers. We also have to make sure that the allocated memory space is all freed from the heap.

Load Module This function reads an CUDA binary object file (*.cubin) to parse the code and stack information of the composed functions. We assume that the program is compiled using the NVIDIA CUDA Compiler (NVCC) [5], and the object file is an ELF format. Because this function is called only once for the program, we allocate device memory space to all the code and static data sections at once, and upload them to the device memory by sending a specific set of commands to the GPU.

Unload Module All we have to do for this function is to free the allocated device memory space.

Allocate Memory This function allocates virtual memory space. The memory management nodes must be managed to track the remaining region of virtual memory space. The page table must also be managed in this function. We can use the PCI BARs to directly access the page table on the device memory.

Free Memory This function frees the specified virtual memory space by deleting the corresponding entry of the page table. The memory management nodes must also be updated accordingly.

Copy Data There are synchronous and asynchronous copy functions. We use the same, but different instances of, memory copy engines for both of the functions. They can be managed by a pretty simple set of commands specifying the source and the destination addresses of the copy operation together with the mode of DMA such as linear and split transactions. It is important to note that multiple copy operations cannot be overlapped within the same channel. If needed, this function internally creates another channel associated with the same context to perform different copy operations concurrently.

Set Parameter This function just stores the parameter data in some buffer maintained by the runtime library. All the data stored in this buffer are sent to the device memory together when the corresponding kernel is launched.

Launch Kernel This function sends the longest sequence of commands to the GPU. It sets local/shared/global memory space, and the parameters are sent to constant memory space. The rest of the context information including grids, blocks, barriers, and registers are also set in this function followed by some preliminary setup for the kernel. Finally the kernel is launched on the GPU. All these procedures are included in the sequence of commands.

Synchronize GPU This function coordinates with the "Launch Kernel" function. Every time a kernel is launched, we generate another sequence of commands that write a sequential number to a specific virtual address. When the synchronization function is called, the program polls on this address. If the value at this address changes, it means that the preceding kernel execution completes. Thus the program is synchronized with the GPU. This technique is often called a fence.

3.2 Backend

The backend implementation of CUDA runtime could take several approaches. This is because the MMIO space can be accessed by both the user space and the kernel space. In other words, the resource manager could exist in both the runtime library and the device driver. However, there are pros and cons. Having the CUDA runtime in the kernel space makes the system more secure. This approach also allows kernel modules to access the GPU. For example, file systems could use CUDA to accelerate file encryption or RAID parity checking. The shortcoming of this approach is that the program could be blocked for a long time in the kernel space, because a blocking copy operation is performed by the device driver.

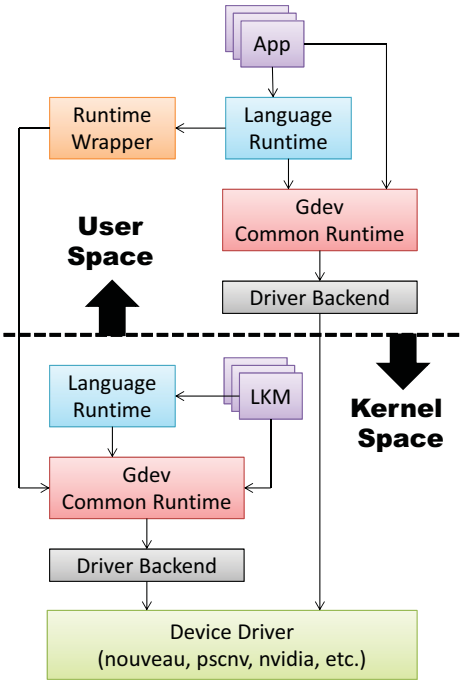


Figure 3. Our approach to open-source CUDA runtime implementation.

We implement the CUDA runtime software so that the system designer can choose the kernel space or the user space. Figure 3 illustrates an overview of our approach to open-source CUDA runtime implementation. We use Gdev [16] as the underlying software. There are the same set of "Language Runtime", "Gdev Common Runtime", and "Driver Backend" in both the user space and the kernel space. We implement them using almost the same piece of code. Some obvious differences between them include `malloc()/kalloc()`, `printf()/printk()`, and `yield()/sched_yield()`. Albeit different function names, they are very similar formats and it is quite easy to unify them by preprocessor macros. By providing the `MALLOC()`, `PRINTF()`, and `YIELD()` macros that are preprocessed to appropriate functions at compile time, we can use exactly the same piece of code for the both kernel-space and user-space runtimes. This solution reduces the maintenance overhead. We also extract the driver-dependent code from our CUDA runtime implementation. This modularity makes our implementation portable to different device drivers. Currently we support Nouveau, PSCNV (another open-source driver from PathScale) and NVIDIA's proprietary driver. Our contribution is useful to compare the advantages and disadvantages of these device drivers under the same runtime mechanism.

4 Summary

We have presented a detailed documentation of "how GPUs work" and an open-source implementation of CUDA runtime. Our documentation is a useful contribution to facilitate further systems research on GPU computing. Our open-source implementation of CUDA runtime opens up several approaches to GPU resource management. It may be downloaded from `http://github.com/shinpei0208/gdev`. We are currently working on virtualization and dynamic power management of GPUs to broaden applications of our project. We also plan to complement the implementation of missing CUDA API functions to make it more practical.

References

[1] Compute Unified Device Architecture. `http://www.nvidia.com`.

[2] Nouveau: Accelerated Open Source driver for nVidia cards. `http://nouveau.freedesktop.org`.

[3] Open Computing Language. `http://www.khronos.org`.

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

[5] CUDA TOOLKIT 4.2. `http://developer.nvidia.com/cuda/cuda-downloads`, 2011.

[6] NVIDIA GeForce GTX 680: The fastest, most efficient GPU ever built. `http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf`, 2012.

[7] P. Bhatotia, R. Rodrigues, and A. Verma. Shredder: GPU-Accelerated Incremental Storage and Computation. In *Proc. of the USENIX Conference on File and Storage Technologies*, 2012.

[8] R.E. Faith. *The Direct Rendering Manager: Kernel Support for the Direct Rendering Infrastructure*. Precision Insight, Inc., May 1999.

[9] A. Gharaibeh, S. Al-Kiswany, S. Gopalakrishan, and M. Ripeanu. A GPU-accelerated storage system. In *Proc. of the ACM International Symposium on High Performance Distributed Computing*, pages 167–178, 2010.

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

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

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

[13] S. Kato, K. Lakshmanan, Y. Ishikawa, and R. Rajkumar. Resource Sharing in GPU-accelerated Windowing Systems. In *Proc. of the IEEE Real-Time and Embedded Technology and Aplications Symposium*, pages 191–200, 2011.

[14] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar. RGEM: A Responsive GPGPU Execution Model for Runtime Engines. In *Proc. of the IEEE Real-Time Systems Symposium*, pages 57–66, 2011.

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

[16] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt. Gdev: First-Class GPU Resource Management in the Operating System. In *Proc. of the USENIX Annual Technical Conference*, 2012.

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

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

[19] W. Sun, R. Ricci, and M. Curry. GPUstore: Harnessing GPU Computing for Storage Systems in the OS Kernel. In *Proc. of Annual International Systems and Storage Conference*, 2012.