Toward Fine-grained GPU Resource Management using Microcontrollers

Yusuke Fujii, Takuya Azumi, and Nobuhiko Nishio
Department of Computer Science
Ritsumeikan University

Shinpei Kato
Department of Information Engineering
Nagoya University

***Abstract*—Recent graphics processing units (GPUs) integrate wimpy microcontrollers on a chip. These microcontrollers are highly available to extend the functionality of GPU resource management, launching firmware code to control GPU executions and data transfers without accessing the host CPUs at all. In this paper, we develop a compiler and debugging environment for NVIDIA's GPU microcontrollers in order to enhance the productivity of GPU firmware development. Our compiler is implemented using the well-known portable LLVM compiler infrastructure, while together providing a debugging subsystem that can individually execute firmware on the microcontroller. As a proof of concept, we develop fully-functional firmware using our compiler and debugging environment, and evaluate it using an NVIDIA's graphics card. Experimental results demonstrate that the overhead of introducing our firmware is suppressed to within 2.3%, as compared to the native proprietary firmware. It is also identified that the impact of overhead is no greater than 0.01% of the total execution time of microbenchmark programs.**

***Keywords*-GPGPU; LLVM; Microcontrollers**

I. Introduction

Graphics processing units (GPUs) are becoming more and more commonplace to support compute-intensive and data-parallel computing. In many application domains, GPU-accelerated systems provide significant performance gains over traditional multi-core CPU-based systems. As shown in Table I, the peak performance of the state-of-the-art GPUs exceeds 3,000 GFLOPS, integrating more than 1,500 cores on a chip, which is nearly equivalent of 19 times that of traditional microprocessors, such as Intel Core i7 series. Such a rapid growth of GPUs is due to recent advances in programming support, such as CUDA [1] and OpenCL [2], for general-purpose computing on GPUs, also known as GPGPU.

In recent years, real-time systems have been augmented with the GPU [3]–[8]. The motivation of using the GPU in real-time systems is mainly found in emerging applications of cyber-physical systems [9]–[11], where a large amount of data acquired from the physical world needs to be processed in real-time. Given that the workload of such applications is highly compute-intensive and data-parallel, many-core computing on the GPU is best suited to meet the real-fast requirements of computation. What is challenging in this line of work is to control the GPU under real-time constraints. The GPU is a coprocessor independent of the CPU, and hence two different pieces of code are running concurrently on the GPU and the CPU, respectively. This heterogeneity poses a core challenge in resource management. Since the GPU is designed to accelerate particular workload, resource management functions are often performed on the CPU. In other words, the GPU and the CPU must be synchronized in some way to ensure timeliness. Unfortunately, this could be a major source of latency that makes real-time systems unpredictable [3], though the previous work are forced to take this approach due to a lack of functionality that enables resource management functions to offload on to the GPU. While compute cores or shaders on the GPU are not available to perform resource management, recent GPUs integrate microcontrollers on a chip where firmware code is launched to control the functional units of the GPU. These microcontrollers are highly available to extend the functionality of GPU resource management, launching special pieces of firmware code to control GPU executions and data transfers.

This paper presents a compiler and debugging environment for NVIDIA's GPU microcontrollers based on the well-known portable LLVM compiler infrastructure. The main purpose of this environment is to enhance the productivity of GPU firmware development so that the community can facilitate future research on fine-grained GPU resource management using microcontrollers. Firmware is self-contained within the GPU, and there will be interference from background jobs running on the CPU, once it is uploaded by the device driver. Therefore, we believe that GPU computing would be more timely and reliable for real-time systems, if the firmware can support GPU resource management by itself. In this paper, we develop an initial stage of the firmware, and evaluate its basic performance.

The rest of this paper is organized as follows. Section I introduces the underlying platform technology. Section II describes the design and implementation of our compiler and debugging environment for NVIDIA's GPU microcontrollers, and Section IV evaluates its basic performance. This paper is concluded in Section V.

II. Platform Technology

First of all, we describe the platform technology underlying our development. We intensively focus on NVIDIA's

Table I
COMPARISON OF THE INTEL CPU ARCHITECTURES AND THE NVIDIA GPU ARCHITECTURES

	Core i7 980XE	Core i7 3960X	GeForce GTX285	GeForce GTX480	GeForce GTX680
# of processing cores	6	6	240	480	1536
Single-precision performance (GFLOPS)	108.0	158.4	933.0	1350.0	3090.0
Memory bandwidth (GB/sec)	37.55	51.2	159.0	177.0	192.2
Power consumption (watt)	130	278	183	250	195
Release date	2010/03	2011/11	2009/01	2010/04	2012/03

GPU architectures, while the idea of integrating GPU resource management into on-chip microcontrollers is not limited to these specific architectures. All pieces of technology presented herein are open-source, and may be downloaded from the corresponding websites, respectively.

A. Assembler for GPU microcontrollers

The assembler is comprised in package of the Envytools suite [12]. The Envytools suite is a rich set of open-source tools to compile or decompile GPU shader code, firmware code, macro code, and so on. It is also used to generate header files of GPU command definitions used by the device driver and the runtime library. There are many other useful tools and documentations for NVIDIA's GPU architectures enclosed in the Envytools suite.

B. GPU Device Driver

In general, the application programming interface (API) for the GPU is provided by the runtime library. GPU resource management, on the other hand, is often supported by the device driver and the operating system (OS) module [3], [13], [14]. As part of resource management, the device driver communicates with microcontrollers integrated on the GPU. The communication is typically managed by specific commands, which can be handled by firmware running on each microcontroller.

The firmware is built into the device driver by a shape of byte code, and is uploaded on to the GPU at boot time. To do so, we require open-source software, because we have to build the firmware into the device driver. In this paper, we use Gdev [13], an open-source module of the GPGPU device driver and runtime library.

C. LLVM Infrastructure

The LLVM (Low Level Virtual Machine) project is a collection of open-source modular and reusable compiler tool sets. Since the microcontroller has its own instruction set architecture, we develop an architecture-dependent backend of LLVM so that we can make use of all the front-end modules of LLVM.

Figure 1 illustrates the structure of LLVM. It first generates the LLVM IR (Intermediate Representation) from the source code. This IR code is assembled by the LLVM backend. The assembly code is finally translated to the object code for the target machine.

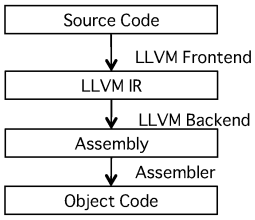

Figure 1. Compilation stages of LLVM.

1) LLVM IR: The LLVM IR is an intermediate language used in LLVM, also called bitcode or LLVM assembly languages. This intermediate language is very powerful, scalable, light-weight, and low-level enough to underlie many languages on top of many architectures. LLVM uses an expression of SSA (Static Single Assignment), which is suitable for a lot of compiler optimization algorithms.

2) LLVM frontend: The LLVM frontend generates an intermediate language from a high-level language in LLVM. It is mainly used for code generation and its optimization. In particular, we use Clang for our development, which is an open-source compiler for the C family of programming languages provided by LLVM.

3) LLVM backend: The LLVM backend generates target code from an intermediate language in LLVM. The backend of LLVM features a target-independent code generator that may create output for several types of target processors including X86, PowerPC, ARM, and SPARC. This backend framework may also be used to generate code targeted at accelerators such as Cell B/E and GPUs. In fact, NVIDIA has announced recently that they use LLVM for the basis of their CUDA compiler. The backend of LLVM is composed of the LLC (LLVM static Compiler) and the LLI (LLVM Interpreter). LLI is an interpreter of the LLVM IR, also available as a JIT compiler, while LLC is a static compiler to generate code. We use this backend part of LLVM to generate code targeted at NVIDIA's GPU microcontrollers.

III. COMPILER AND DEBUGGING ENVIRONMENT

This section describes the design and implementation of our compiler and debugging environment for NVIDIA's GPU microcontrollers.

Table II
SPECIFICATION OF GF100 MICROCONTROLLER.

Name	HUB	GPC
Number of units	1	4
Bit	32bit	32bit
Code size	16,384 byte	8,192 byte
Data size	4,096 byte	2,048 byte

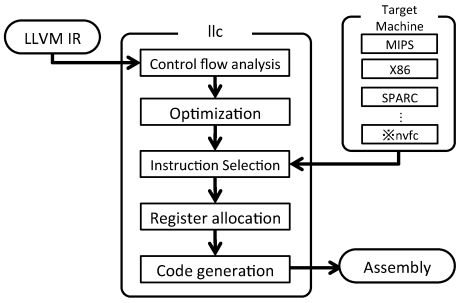

Figure 3. Code generation stages of LLC.

A. Microcontroller

This paper presumes the microcontroller of NVIDIA's Fermi architecture. In particular, we target the GeForce GTX 480 graphics card designed based on the GF100 architecture. In this architecture, a streaming multiprocessor (SM) consists of 32 CUDA cores, while a graphics processing cluster (GPC) consists of 4 SM's. There are four GPC's in total equipped in the GF100 architecture, and hence the maximum number of CUDA cores is 512.

Table II illustrates the specification of the GF100 microcontroller. There are two types of microcontrollers, HUB and GPC, relevant to CUDA engines. HUB is broadcasting the access to all GPC's, while the GPC represents a specific microcontroller for each GPC engine. Since the maximum code size is limited to 16KB as indicated in Table II, developers should carefully design the firmware.

B. Compiler Implementation

Figure 2 shows an overview of our compiler implementation. The main flow of compilation is done by Clang. It generates the LLVM IR from the C source file. The LLC next generates assembly code, which contains code and data in separate files. Finally, the Envytools outputs an executable file. This executable file can be launched by the device driver, and can also be tested by our debugging tool described in the later section. To summarize, our compiler takes the following stages:

(1) Clang
This is a frontend of C language that generates LLVM IR code from the source file.

(2) LLC with nvuc
This is a backend of LLVM that compiles LLVM IR code into assembly code. As shown in Figure 3, there are five steps to exploit compilation: (i) flow analysis, (ii) optimization, (iii) instruction selection, (iv) register allocation, and (v) code generation. This flow is not dependent on the target machine. The LLC reads a configuration of the target machine at the time of instruction selection, and selects a set of the instruction and register to meet the specifications of each machine. Our implementation adds a new configuration called nvuc (NVIDIA Micro-Controller) to support NVIDIA's GPU microcontrollers under LLVM.

(3) LLVM to envyas
This stage divides the generated assembly code into code and data sections so that we can create binary images using "envyas", which is a microcontroller assembler provided by the Envytools suite. The bootstrap code is also unified into the binary images in this stage.

(4) envyas
This is a final assembly stage for the microcontroller, which generates the byte code of the firmware.

(5) hex to bin
This stage translates the hexadecimal byte code to the binary format so that the firmware can execute on the microcontroller.

(6) Running the microcontroller
The compiled firmware is loaded on the microcontroller by the device driver. We also support a debugging tool that launches the firmware in the same way as the device driver for development purposes.

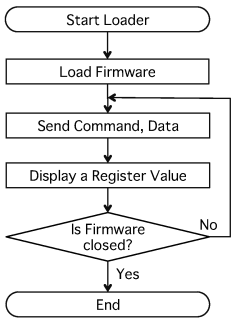

Figure 4. Flowchart of our debugging tool.

C. Debugging Support

We support a debugging tool to load the firmware, send commands and data, monitor the status, and display register and memory values of the microcontroller. Figure 4 shows control flow of our debugging tool. The following are the details of each block in the flow:

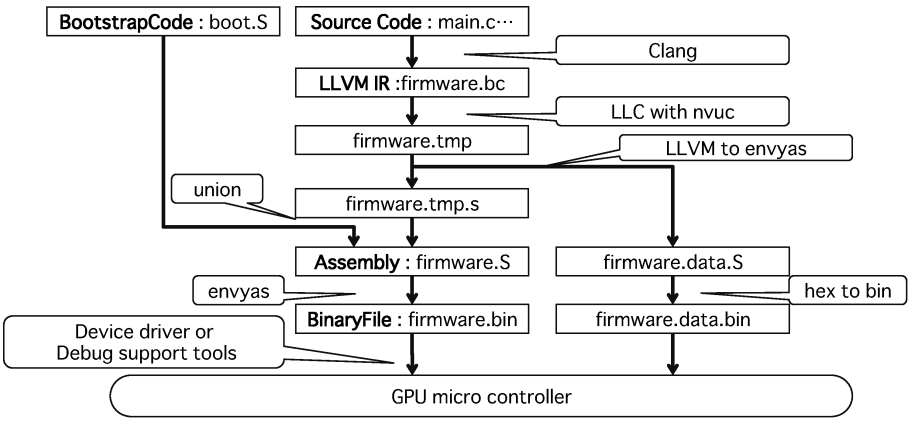

Figure 2. Overview of Compiler Implementation.

(1) Load Firmware
Our debugging tool uploads a set of firmware programs on to the HUB and the GPC microcontrollers. The uploaded firmware programs start execution when a flag is set in the specified register.

(2) Send Command and Data
The microcontroller is event-driven. It is totally suspended in an idle state. When it receives a command from the debugging tool through the PCI bus, the interrupt handler is invoked and its execution is resumed.

(3) Display Register Value
The microcontroller has a set of registers that may be used by the debugging tool for any purpose. There are also several important registers relevant to firmware execution. The debugging tool hence displays the values of these registers to notify what is happening.

D. Firmware Development

In this paper, we present the most basic firmware program for NVIDIA's GPU microcontrollers. We develop this firmware entirely using our compiler and debugging environment. This is indeed the initial step toward fine-grained GPU resource management using microcontrollers, and enhanced functions could build upon this work.

Figure 5 shows control flow of the basic firmware developed in this paper. The following are the details of each block in the flow.

(1) initialize
The firmware configures the interrupt handler, and receives the default set of data when started.

(2) sleep
The firmware enters the standby mode in the main event loop, waiting for the next command issued by the device driver or the debugging tool. Upon every arrival of the command, an interrupt is generated on the microcontroller, awakening the firmware in the "ihbody" function.

(3) ihbody
This is an interrupt handler invoked by the command. All we have to do here is to enqueue the corresponding command, and releases the standby mode to resume firmware execution.

(4) work
This is a main body of the firmware. It is called when the firmware is released from the standby mode. The basic procedure of this function is to dequeue a pending command one by one, and call the function corresponding to the command. If the specified flag is cleared, we destroy the firmware.

Table III
EXPERIMENTAL SETUP.

CPU	Intel core i7 2600
GPU	NVIDIA GeForce GTX480
Memory	8GB
Kernel	Linux 2.6.42.12-1.fc15.x86_64
Device driver	Gdev [13]

IV. EXPERIMENTS

We now evaluate the basic performance of the firmware developed using our compiler and debugging environment. We also discuss what we have found in our experiments.

A. Performance Evaluation

We evaluate the performance of our firmware as compared with NVIDIA's proprietary firmware blob. Table III shows our experimental setup. For fair comparison, we use Gdev [13] as the underlying GPU device driver and runtime, given that our firmware is available for only the open-source environment. The performance metric of our experiments is a total execution time including GPU executions and data transfers between the host and the device.

To average variations in the execution time, we run each test program 100 times. Our observation is that our firmware and NVIDIA's firmware exhibit similar behavior -

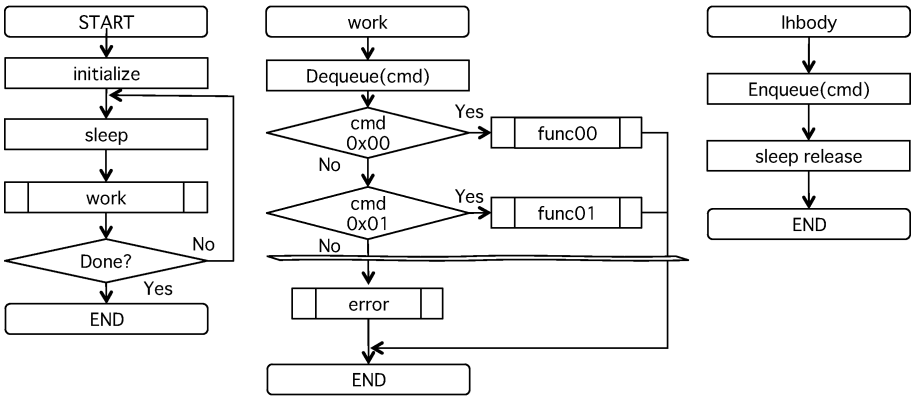

Figure 5. Flowchart of our basic firmware.

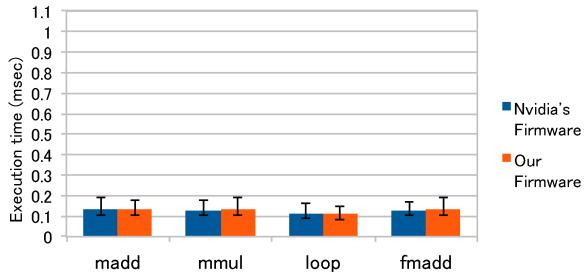

Figure 6. Execution time of microbenchmark programs: Case (i).

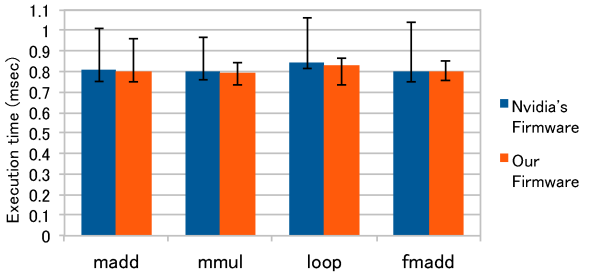

Figure 7. Execution time of microbenchmark programs: Case (ii).

the execution time is mostly less than 2 ms or more than 7 ms. We hence categorize them into two cases: (i) less than 2 ms and (ii) more than 7 ms. Figures 6 and 7 depict the experimental results of these two cases, respectively, where the X axis lists our microbenchmark programs and the Y axis shows their execution time.

As can be seen from the experimental results, the performance difference between our firmware and NVIDIA's firmware is very trivial in Case (i). For example, it is at most 0.003 ms in the madd program, which is equivalent to 2.31% of the time. In the case (ii), on the other hand, our firmware rather outperforms NVIDIA's firmware by 0.002 ms, i.e., 1.74% of the time. Such a performance difference, however, is negligible due to the following reasons.

- The observed performance difference is much smaller than their error margin.
- The total execution time is occupied by GPU executions rather than firmware execution. For instance, the total execution time of the madd program under NVIDIA's firmware is 21.842 ms, which is mostly dominated by the host-side execution, whereas the firmware execution time is no greater than 0.01% of the total execution time.

The above experimental results imply that our compiler and debugging environment for NVIDIA's GPU microcontrollers is reliable in performance. Given that firmware developers can use C language rather than hand assembling, we believe that the contribution of this paper is significant in this line of work.

B. Discussions

Through the experiment, we observed that the execution time of the madd program was 214 ms at the first trial, while that after the second time is around 20 ms. We found out that this big gap comes from the fact that the firmware has to generate the GPU context at the first run, while it can reuse the same context information from the second run. This explains the cost of generating the GPU context. Without self-firmware development, we would have never have these findings.

V. Conclusion

In this paper, we have presented a new compiler and debugging environment for NVIDIA's GPU microcontrollers. As a basis of future work toward fined-grained GPU resource management, we developed new firmware for those microcontrollers using our development environment. We executed several microbenchmark programs to demonstrate that the overhead introduced by our firmware was no greater than 2.31% of the total execution, as compared to NVIDIA's

proprietary firmware blob, while our firmware even outperformed NVIDIA's firmware depending on test cases. One of the interesting findings obtained through the experiments was the overhead of generating the GPU context, which must be minimized and bounded in real-time systems. Our development environment is all open-source, and may be download from our web site [15]–[17].

In future work, we pursue a new direction of GPU resource management using microcontrollers. First of all, the CPU load could be reduced by offloading GPU resource management functions on to the GPU microcontroller. This idea in fact is inspired by the Helios project [18], where networking resource management functions are offloaded onto the NIC microcontroller. Preemption support and power management for the GPU could also be achieved by extending the firmware, as discussed in [19]. We believe that such a fine-grained GPU resource management approach is significant for real-time systems augmented with the GPU.

REFERENCES

[1] NVIDIA, "CUDA," http://www.nvidia.com/.

[2] KHRONOS, "OpenCL - The open standard for parallel programming of heterogeneous systems," http://www.khronos.org/opencl/.

[3] 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.

[4] 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*, 2011, pp. 57–66.

[5] 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 Applications Symposium*, 2011, pp. 191–200.

[6] C. Basaran and K.-D. Kang, "Supporting Preemptive Task Executions and Memory Copies in GPGPUs," in *Proc. of the Euromicro Conference on Real-Time Systems*, 2012, pp. 287–296.

[7] G. Elliott and J. Anderson, "Robust Real-Time Multiprocessor Interrupt Handling Motivated by GPUs," in *Proc. of the Euromicro Conference on Real-Time Systems*, 2012, pp. 267–276.

[8] ——, "Globally Scheduled Real-Time Multiprocessor Systems with GPUs," *Real-Time Systems*, vol. 48, no. 1, pp. 34–74, 2012.

[9] J. Aumiller, S. Kato, N. Rath, and S. Brandt, "Supporting Low-Latency CPS using GPUs and Direct I/O Schemes," in *Proc. of the International Workshop on Cyber-Physical Systems, Networks, and Applications*, 2012.

[10] M. McNaughton, C. Urmson, J. Dolan, and J.-W. Lee, "Motion planning for autonomous driving with a conformal spatiotemporal lattice," in *Proc. of the IEEE International Conference on Robotics and Automation*, 2011, pp. 4889–4895.

[11] J. Ferreira, J. Lobo, and J. Dias, "Bayesian real-time perception algorithms on GPU," *Journal of Real-Time Image Processing*, vol. 6, no. 3, pp. 171–186, 2011.

[12] M. Koscielnicki, "Envytools," https://0x04.net/envytools.git.

[13] 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.

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

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

[16] Y. Fujii, T. Azumi, and S. Kato, "FARM Project," https://github.com/yukke0826/farm, 2012.

[17] ——, "NVIDIA Firmware Compiler Project," https://github.com/yukke0826/nvfc, 2012.

[18] E. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. C. Hunt, "Helios: heterogeneous multiprocessing with satellite kernels," in *ACM Symposium on Operating Systems Principles*, 2009, pp. 221–234.

[19] S. Kato, S. Brandt, Y. Ishikawa, and R. Rajkumar, "Operating Systems Challenges for GPU Resource Management," in *Proc. of the International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, 2011, pp. pp. 21–30.