Exploring Microcontrollers in GPUs

Yusuke Fujii[1], Takuya Azumi[1], Nobuhiko Nishio[1], and Shinpei Kato[2]
[1]Ritsumeikan University, [2]Nagoya University
yukke@ubi.cs.ritsumei.ac.jp, {takuya,nishio}@cs.ritsumei.ac.jp, shinpei@is.nagoya-u.ac.jp

Abstract

Recent graphics processing units (GPUs) integrate wimpy microcontrollers on a chip. They are often used to execute firmware code configuring the functional units of GPUs. This paper opens up the programming of these microcontrollers and explores how to utilize them for GPU resource management. Our prototype system provides a compiler suite for NVIDIA's GPU microcontrollers with its basis on the Low Level Virtual Machine (LLVM) infrastructure. As a proof of concept, we develop fully-functional firmware using our compiler and provide a basic performance evaluation. The experimental results demonstrate that the overhead of introducing our firmware is suppressed to within 2.3%, as compared to the native proprietary firmware, while the impact of overhead is no greater than 0.01% of the total execution time according to microbenchmarks. We also show that a complementary use of microcontrollers can reduce the latency of data transfers with concurrent multiple data streams.

1. Introduction

Graphics processing units (GPUs) are becoming powerful compute devices for high-performance computing applications. Peak performance of high-end GPUs reaches 3 TFLOPS as of 2013, integrating thousands of cores on a chip [10], which is nearly equivalent of 20 times that of traditional microprocessors, such as Intel Core i7 series. This rapid growth of GPU technology is largely due to recent advances in general-purpose computing on GPUs (GPGPU).

The GPU is typically controlled by the CPU. This heterogeneous nature of GPU programming poses a core challenge in resource management. Specifically a synchronization between the GPU and the CPU could be a major source of latency [4, 6], which is raised due to a programming model where a resource manager is running on the CPU while sending commands to the GPU. It is apparent that compute cores on the GPU are not suitable to run resource management code. Instead recent GPUs integrate microcontrollers on a chip where firmware is often loaded for the purpose of runtime GPU configuration. These microcontrollers could be useful resources to support the resource manager in GPU programming.

This paper presents a compiler suite making the programming of GPU microcontrollers more productive. It is based on the Low Level Virtual Machine (LLVM) infrastructure, whose backend codegen is ported to the instruction set architecture (ISA) of NVIDIA's GPU microcontrollers. We also develop a new data transfer method that coordinates direct memory access (DMA) engines and GPU microcontrollers to reduce the data transfer latency with concurrent multiple data streams. The potential use of GPU microcontrollers may include scheduling and interrupt handling, but we restrict our attention to data transfers in this paper. To summarize, we make the following contributions:

1. The backend port of LLVM for NVIDIA's GPU microcontrollers.
2. The firmware code including a new data transfer method for NVIDIA's GPU microcontrollers.

The rest of this paper is organized as follows. Section 1 introduces existing software platforms that we utilize for our compiler suite. Section 2 describes the design and implementation of our compiler suite for

NVIDIA's GPU microcontrollers including firmware code itself, and Section 4 evaluates its basic performance. This paper is concluded in Section 5.

2. Platform Technology

This section presents the platform technology underlying our prototype system. We focus on NVIDIA's GPUs but the idea of integrating GPU resource management into on-chip microcontrollers is available for most other GPUs.

2.1 Assembler for GPU microcontrollers

The assembler is provided by the Envytools suite [7], which 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.

2.2 GPU Device Driver

GPU resource management primitives are often provided by the device driver and the operating system (OS) module [1, 4, 5]. Particularly the device driver must employ an interface to communicate with GPU microcontrollers to set up the status of the GPU itself. A piece of code running on each microcontroller is often called *firmware*.

The firmware is built into the device driver as byte code and is uploaded on to the GPU at boot time. We use Gdev [5], which is an open-source set of the GPGPU device driver and the runtime library, to integrate firmware code into the device driver.

2.3 LLVM Infrastructure

LLVM is a collection of open-source modular and reusable compiler tool sets. Since the microcontroller has its own ISA, we develop an architecture-dependent backend port of LLVM so that we can leverage all the front-end modules provided by LLVM. The compilation stages of LLVM are summarized in Figure 1.

2.3.1 LLVM IR

The LLVM IR is an intermediate language used in LLVM, which is designed scalable, light-weight, and low-level to support many languages on top of many architectures. LLVM also uses an expression of Static

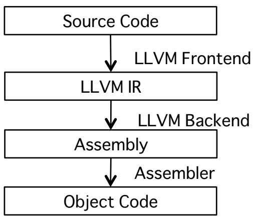


Figure 1. Compilation stages of LLVM.

Single Assignment (SSA), which is suitable for many compiler optimization algorithms.

2.3.2 LLVM frontend

The LLVM frontend generates the intermediate language from a high-level language. The main role of this stage includes code generation and optimization. We use Clang as a frontend for our development, which is an open-source compiler for the C family of programming languages provided by LLVM.

2.3.3 LLVM backend

The LLVM backend generates target code from the intermediate language. This backend is highlighted with a target-independent code generator that may create output for several types of target processors including X86, PowerPC, ARM, and SPARC. It 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 their CUDA compiler. The main components of this backend are 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 for code generation. We use this backend to generate code targeted at NVIDIA's GPU microcontrollers.

3. Compiler Suite

We now present the design and implementation of our compiler for NVIDIA's GPU microcontrollers. We utilize the software platforms introduced in Section 2 while self-developing (i) the backend port of LLVM for NVIDIA's GPU microcontrollers and (ii) the firmware code with a new data transfer method.

3.1 Microcontroller

This paper assumes the NVIDIA GF100 architecture [9], particularly focused on the GeForce GTX 480 graphics card. In this architecture, a streaming multiprocessor (SM) consists of 32 CUDA cores, while a

Table 1. Details of the GF100 microcontroller.

	HUB	GPC
Number of units	1	4
Addressing	32 bits	32 bits
Code section	16,384 bytes	8,192 bytes
Data section	4,096 bytes	2,048 bytes

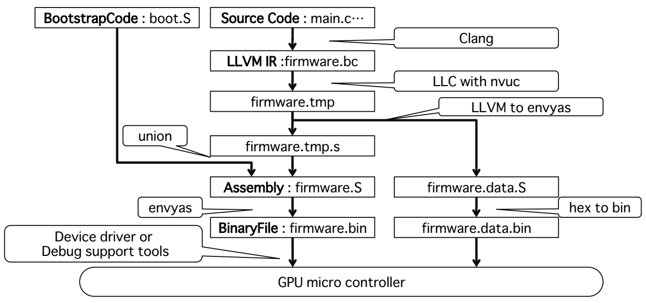


Figure 2. Overview of compiler implementation.

graphics processing cluster (GPC) consists of 4 SM's. Given that the GF100 architecture may equip 4 GPC's in total, the maximum number of CUDA cores is 512.

Table 1 illustrates the details of the GF100 microcontroller. There are two types of microcontrollers called *HUB* and *GPC*. 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, developers should carefully design firmware code.

3.2 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. 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,

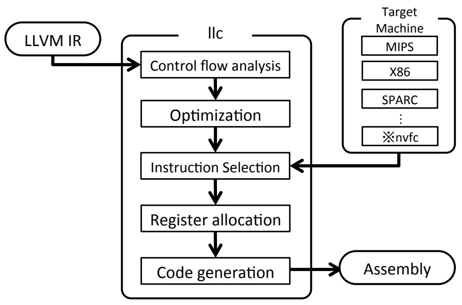


Figure 3. Code generation stages of LLC.

(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 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.

3.3 Firmware Development

We made reverse engineering of NVIDIA's binary firmware to obtain its assembly code. By hand decompiling, we retrieved C code and extended it to employ new functions. This extended firmware code is recompiled using our compiler suite. Note that the we publish our compiler suite but not our firmware code, because

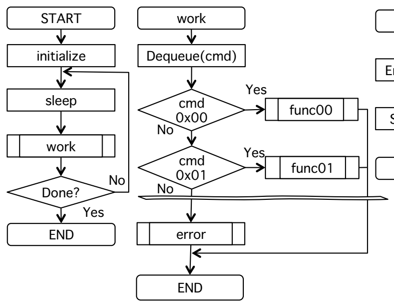


Figure 4. Flowchart of firmware execution.

it partly includes the original binary firmware code. However we can make this code available upon request only for research usage.

Figure 4 shows the basic control flow of firmware execution. 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.

4. Experiments

We now evaluate the effectiveness of our compiler suite and developed firmware.

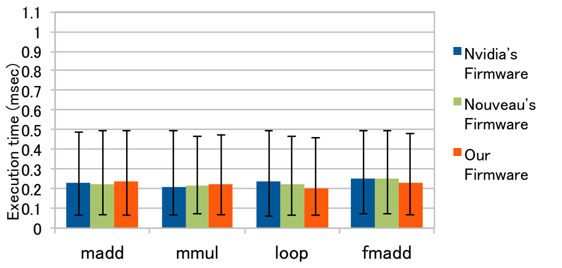
4.1 Basic Performance

We look into the basic performance of our firmware as compared to NVIDIA's proprietary firmware and Nouveau's open-source firmware. Table 2 shows the experimental setup. We restrict the performance metric to the total execution time including GPU executions and data transfers between the host and the device. We run several microbenchmark program 100 times each to average variations in the execution time. Our observation is that the three firmware instances exhibit similar behavior: the execution time is mostly (i) less than 0.3 ms or (ii) more than 0.7 ms.

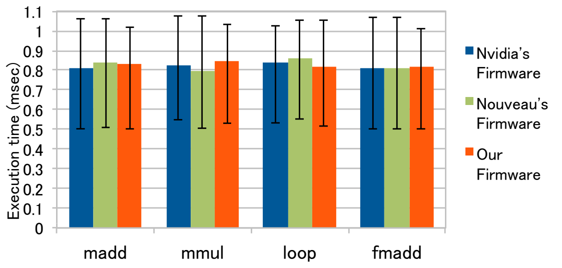
Figures 5 depict the experimental results of two respective cases (execution times boundary of 0.5 ms), where the X axis lists our microbenchmarks while the Y axis shows their execution time. As can be seen from the experimental results, the performance difference between our firmware and the existing firmware is trivial in Case (a). For example, it is at most 0.003 ms in the madd program, which is equivalent to 2.31% of the time. In Case (b), on the other hand, our firmware rather outperforms NVIDIA's firmware by 0.002 ms,

Table 2. Experimental setup.

CPU	Intel core i7 2600
GPU	NVIDIA GeForce GTX480
Memory	8GB
Kernel	Linux 2.6.42.12-1.fc15.x86_64
Runtime/Driver	Gdev [5] and Nouveau (Linux driver)

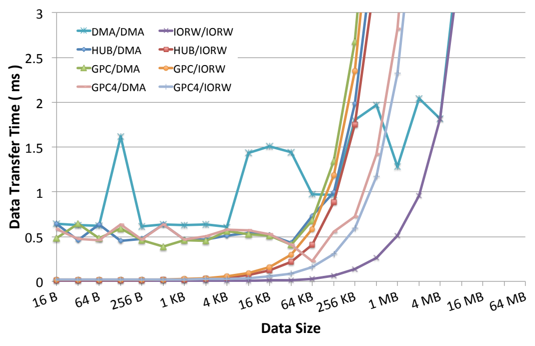


(a) Case for execution times less than 0.5 ms

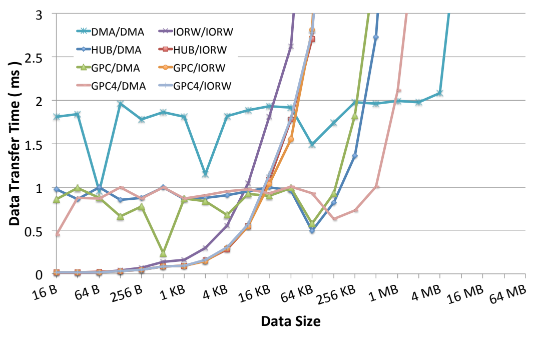


(b) Case for execution times more than 0.5 ms

Figure 5. Execution times



(a) Host to Device



(b) Device to Host

Figure 6. The average performance of the combined data transfer methods for concurrent tasks.

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 suite for NVIDIA's GPU microcontrollers is reliable in performance. Given that firmware developers can use the C language rather than hand assembling, we believe that the contribution of our compiler suite is significant for this line of work.

4.2 Data Transfer Performance

A microcontroller supports special instructions to transfer the data stored in its data sections to and from the host or the device memory. The data transfer is offloaded to the microcontroller, *i.e.*, DMA, but is controlled by the microcontroller itself. Leveraging this mechanism, we provide data communications between the host and the device memory. Furthermore, we now evaluate the performance of concurrent *two* data streams using different combinations of the data transfer methods as shown in Figure 6 where the labels denote the investigated data transfer methods:

- **DMA** denotes the standard DMA method presented.
- **IORW** denotes the memory-mapped read and write method.
- **HUB** denotes the microcontroller-based data transfer method, particularly using a *hub* microcontroller designed to broadcast among the actual microcontrollers of graphics processing clusters (GPCs), *i.e.*, CUDA core clusters.
- **GPC** denotes the microcontroller-based data transfer method, particularly using a single GPC microcontroller.
- **GPC4** denotes the microcontroller-based data transfer method, particularly using four different GPC microcontrollers in parallel. Note that the NVIDIA Fermi architecture provides four GPCs and their microcontrollers can perform individually. Therefore we can split the data transfer into four pieces and make the four microcontrollers work in parallel.

This is a very interesting result. For the host-to-device direction, the best performance is obtained when both the two tasks use IORW. In this case, the two data streams are not overlapped but are processed in sequential due to the use of the same IORW path. Nonetheless it outperforms the other combinations because the performance of IORW is way higher than the other methods as we have observed in a series of the previous experiments.

However, the device-to-host direction shows a different performance. Since IORW becomes slow when the CPU reads the device memory. IORW/IORW is not the best performer any longer. Instead using the microcontroller(s) provides the best performance until 2MB. This is attributed to the fact that the microcontroller-based data transfer method can be overlapped with any other data transfer methods. From 16B to 16KB, a combination of the microcontroller and IORW is the fastest, while that of the microcontroller and DMA is the fastest

Table 3. Firmware code size.

	HUB	GPC
Nvidia's firmware	13.824 bytes	7,168 bytes
Nouveau's firmware	1,536 bytes	1,280 bytes
Our firmware	16,076 bytes	7,870 bytes

from 16KB to 2MB. Note that from 16B to 16KB the performance is aligned with the slow IORW curve. Therefore the choice of HUB, GPC, and GPC4 does not really matter to the performance. However, from 16KB to 2MB the performance is improved by using four microcontrollers in parallel (*i.e.*, GPC4), since DMA is faster than the microcontroller and thereby the performance is aligned with the microcontroller curve.

We learn from this experiment that the microcontroller is useful to overlap concurrent data streams with DMA or IORW, and using multiple microcontrollers in parallel can further improve the performance of concurrent data transfers.

4.3 Code Size

Table 3 shows the code size of firmware. Nouveau's firmware is functionally limited since it is written by assembly code due to a lack of the compiler. That is why its code size is very small. On the other hand, the code size of our firmware is almost the same as that of Nvidia's firmware, even though (i) it provides all the functions that Nvidia's firmware does and (ii) it implements a new data transfer function. This explains that our compiler suite is effective for practical use.

5. Conclusion

In this paper, we have presented a compiler suite and new firmware for NVIDIA's GPU microcontrollers. Towards fine-grained GPU resource management, we developed fully-functional firmware for GPU microcontrollers using our compiler suite. According to microbenchmarks, the basic performance of our firmware was almost equivalent to that of existing Nvidia's and Nouveau's firmware. We also revealed that GPU microcontrollers are useful resources to reduce the latency of data transfers with concurrent multiple data streams. Our compiler suite is open-source, and may be download from our web site [2].

In future work, we pursue a new direction of GPU resource management using microcontrollers. The CPU load could be reduced by offloading GPU resource management functions onto GPU microcontrollers. This idea is inspired by the Helios project [8], where networking resource management functions are offloaded onto NIC microcontrollers. Preemption support and power management for GPUs could also be achieved by extending firmware, as discussed in [3, 11]. We believe that this kind of fine-grained GPU resource management approach is a significant contribution to GPU-accelerated real-time systems.

References

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

[2] Y. Fujii, T. Azumi, and S. Kato. NVIDIA Firmware Compiler Project. `https://github.com/yukke0826/nvfc`, 2012.

[3] 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*, pages pp. 21–30, 2011.

[4] 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.

[5] 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.

[6] S. Kato, J. Aumiller, and S. Brandt. Zero-Copy I/O Processing for Low-Latency GPU Computing. In *Proc. of the IEEE/ACM International Conference on Cyber-Physical Systems*, pages 170–178, 2013.

[7] M. Koscielnicki. The Envytools project. `https://0x04.net/envytools.git`, 2013.

[8] 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*, pages 221–234, 2009.

[9] NVIDIA. NVIDIA's next generation CUDA computer architecture: Fermi. `http://www.nvidia.com/`, 2009.

[10] NVIDIA. NVIDIA's next generation CUDA computer architecture: Kepler GK110. `http://www.nvidia.com/`, 2012.

[11] M. Peres. Reverse engineering power management on NVIDIA GPUs – Anatomy of an autonomic-ready system. In *Proc. of the International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, 2013.