Comparison of Thread Execution Methods for GPU-oriented OpenCL Programs on Multicore Processors

Naohisa Hojo Ittetsu Taniguchi Hiroyuki Tomiyama

Graduate School of Science and Engineering, Ritsumeikan University
1-1-1 Noji-Higashi, Kusatsu, Shiga 525-8577, Japan

ABSTRACT

With the broad deployment of multicore processors, there are increasing demands to port OpenCL programs written for GPUs onto the multicore processors. However, OpenCL programs written for GPUs cannot run efficiently on multicore processors since GPU-oriented OpenCL programs generally consist of a huge number of threads. This paper presents experimental comparisons of three thread execution methods for GPU-oriented OpenCL programs on multicore processors using a set of industry-oriented OpenCL benchmark programs.

1. INTRODUCTION

OpenCL is one of the most popular frameworks for parallel programming due to its open standardization and hardware platform independence. Traditionally, OpenCL was primarily used for GPUs, and there exist huge amounts of software IPs written in OpenCL for GPUs. Recently, with the broad deployment of multicore processors in general-purpose and embedded computing systems, there are increasing demands to port the GPU-oriented OpenCL software IPs onto multicore processors. Although OpenCL programs are portable from a hardware platform to another, their performance is hardly portable [1]. Specifically, OpenCL programs written for GPUs cannot be executed efficiently on multicore processors. One of the reasons is the granularity of parallelism. GPU-oriented OpenCL programs generally consist of a huge number of tiny threads. GPUs can handle a huge number of threads efficiently because GPUs have a large number of cores and hardware supports for fast context switching. On the other hand, multicore processors have a fewer number of cores than GPUs and context switching on multicore processors totally relies on software.

There exist several research efforts on OpenCL program optimization for multicore processors. In [1], the authors studied how to port GPU-oriented OpenCL programs for multicore processors. One of their conclusions is that programmers have to systematically find the optimal parallelism granularity (thread size), and the authors left the problem as one of future research topics. In [2], the authors presented a runtime library for efficient execution of OpenCL threads on multicore processors without changing the size and number of the threads.

This paper presents a refined method for OpenCL thread execution on multicore processors. The method merges multiple OpenCL threads into one in such a way that the new thread processes multiple data items (i.e., work-items). In fact, this idea itself is not new, and we employ the idea in the context of OpenCL. The key contribution of this paper is quantitative comparison of three thread execution methods using industry-oriented OpenCL benchmark programs.

EWiLi'15, October 8th, 2015, Amsterdam, The Netherlands.

2. DATA-PARALLEL EXECUTION IN OPENCL

OpenCL is based on a server-client model, and a server is called a *host* while a client is a *device*. A device is composed of one or more *compute units (CU)*, and a CU in turn is of one or more *processing elements (PEs)*. A program executed on a device is called a *kernel*. Data is partitioned into pieces called *work-items*, and a kernel is executed on multiple PEs with different work-items. Individual instances of the kernel are called *threads*. When the number of the work-items is N, the kernel consists of N threads. The number of work-items (threads) can be larger than the number of physical PEs. In this case, multiple threads are executed on a PE by context switching.

3. THREAD EXECUTION METHODS

This section presents three methods for execution of OpenCL data-parallel threads, where N denotes the number of work-items.

3.1 All-at-a-Time Execution

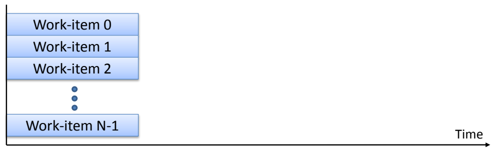
A simple method for thread execution is that we create N threads and execute all of them at a time. We call this method *all-at-a-time* execution, and the method is illustrated in Figure 1 (a). Examples of OpenCL frameworks using the all-at-a-time execution method include the RuCL framework [3]. The all-at-a-time method is simple to implement, and is efficient in case the number of work-items is very small. However, if N is huge, the all-at-a-time method is not executable because the maximum number of threads that operating systems can handle is limited.

3.2 Little-by-Little Execution

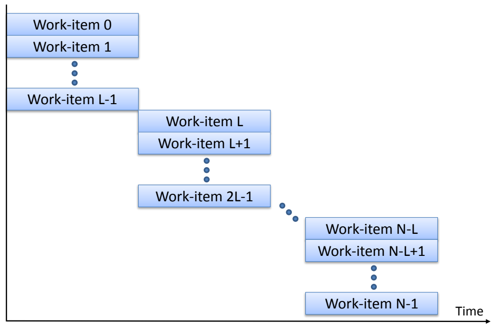
The second method, which is illustrated in Figure 1 (b), is named *little-by-little* execution [2]. In the little-by-little execution method, we repeat thread creation and execution L times, where L is a smaller integer number than N. When a thread finishes its execution, the thread is destroyed, and then a new thread is created and executed. This creation-execution-and-destruction process is repeated until all of the N threads are completed. In this way, the little-by-little execution method tries to keep L threads being active.

3.3 In-the-Loop Execution

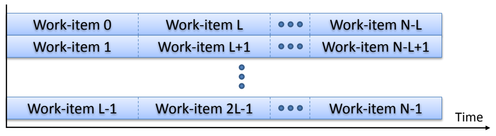
In order to reduce the overheads of thread creation and destruction, we refine the little-by-little method to derive the *in-the-loop* execution as shown in Figure 1 (c). We distinguish OpenCL-threads from OS-threads. An OpenCL thread is a unit of code which processes a single work-item, while an OS thread is a unit of scheduling by the operating system. For example, OS threads correspond to POSIX threads (pthreads) on a Linux operating system. At the beginning of program execution, we statically create L OS-threads. In each OS-thread, there is a loop. In each iteration of the loop, an OpenCL thread is executed and a single work-item is processed. In other words, multiple OpenCL threads are merged into an OS-thread. Similar to the little-by-little execution method, L threads are active in the in-the-loop execution method.



(a) All-at-a-time execution



(b) Little-by-little execution



(c) In-the-loop execution

Figure 1. Thread execution methods

However, unlike the little-by-little method which creates *N* OS-threads in total, the in-the-loop method creates only *L* OS-threads.

4. Experimental Comparisons

In this section, we compare the three thread execution methods. The all-at-a-time and little-by-little methods were already implemented in the RuCL framework [2][3], and we used the implementations in this work. We have newly implemented the in-the-loop execution method in the RuCL framework as well. We have selected three benchmark programs from BEMAP [4]. The BEMAP is a suite of OpenCL programs developed in industry. The programs used in our experiments are Montecarlo (128 work-items), Blacksholes (10,485,760 work-items), and Linearsearch (67,108,864 work-items). We used dual Xeon processors (12 physical cores, 24 logical cores in total) in our experiments.

Figure 2 shows the execution times of the Montecarlo program with the three thread execution methods. For the little-by-little and in-the-loop methods, we varied the value of parameter *L*. When *L* is smaller than the number of cores of the target processor, the all-at-a-time method shows the best performance. This is a very reasonable because the little-by-little and in-the-loop methods with smaller *L* cannot fully exploit the potential parallelism of the processor. When *L* exceeds the number of cores, we see little difference between the three methods. This is also reasonable because 128 threads are small enough for the operating system and the overheads for thread manipulation are trivial.

Figures 3 and 4 show the execution times of Blacksholes and Linearsearch, respectively. Both programs have more than 10 million work-items. For the two programs, the all-at-a-time method caused an error since the operating systems cannot handle such a huge number of threads. For both programs, the in-the-loop execution method with *L*=32 or 64 yields the best performance. The little-by-little method is not efficient due to the overheads of thread creation and destruction. Actually, the execution time of the little-by-little method is up to 2,300 times longer than that of the in-the-loop method. Because of the huge gap, it is not possible to show the results of the two methods in Figures 3 and 4. Therefore, the execution times of the little-by-little method are written on top of the graphs.

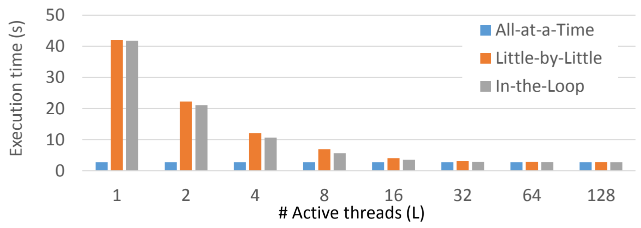


Figure 2. Results for Montecarlo

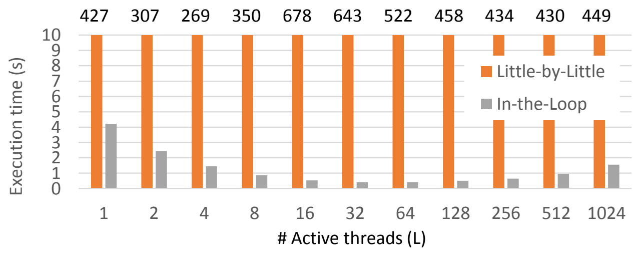


Figure 3. Results for Blacksholes

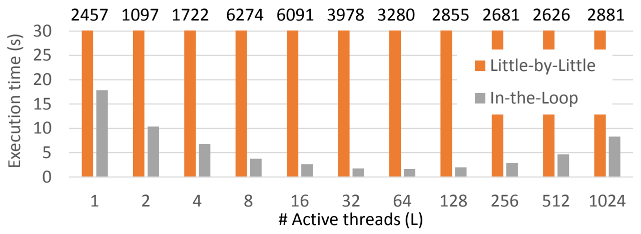


Figure 4. Results for Linearsearch

5. Conclusions

This paper presented an experimental study on thread execution for GPU-oriented OpenCL programs. Using a set of industry-oriented OpenCL benchmark programs, we compared three thread execution methods, i.e., all-at-a-time execution, little-by-little execution and in-the-loop execution. Among the three methods, our experimental results show the effectiveness of the in-the-loop execution method. With the in-the-loop method, the best performance is achieved when the number of threads is slightly larger than the number of cores.

Acknowledgement

This work is in part supported by JSPS KAKENHI 15H02680.

References

[1] J. Shen, J. Fang, H. J. Sips, and A. L. Varbanescu, "An Application-centric Evaluation of OpenCL on Multi-core CPUs," *Parallel Computing*, vol. 39, pp. 834-850, 2013.

[2] N. Hojo, I. Taniguchi, and H. Tomiyama, "Efficient Execution of OpenCL-based GPU Programs on Multicore Processors," *ITC-CSCC*, 2015.

[3] S. Takai, N. Nishiyama, I. Taniguchi, and H. Tomiyama, "A Light-weight OpenCL Framework for Embedded Multicore Processors," *ITC-CSCC*, 2015.

[4] Y. Ardila, N. Kawai, T. Nakamura, and Y. Tamura, "Support Tools for Porting Legacy Applications to Multicore," *ASP-DAC*, 2013.