Bulletin of Networking, Computing, Systems, and Software – www.bncss.org, ISSN 2186-5140
Volume 9, Number 1, pages 31–34, January 2020

Evaluation of the RuCL Framework on Raspberry Pi

Takafumi Miyazaki[1], Hayato Hidari[1], Naohisa Hojo[1], Ittetsu Taniguchi[2], Hiroyuki Tomiyama[1]
[1] Ritsumeikan University [2] Osaka University

***Abstract* — We previously developed the RuCL framework dedicated for running GPU-oriented programs on multicore processors. OpenCL is one of the most popular frameworks for parallel computing. OpenCL is platform independent in principle, and OpenCL programs can be executed on various hardware platforms. However, OpenCL programs written for GPUs are often poorly executed on multicore processors in terms of performance due to the granularity of threads. We port and evaluate our framework on the Raspberry Pi.**

Keywords* — *OpenCL, thread execution, data-parallel execution, multicore processors

I. Introduction

OpenCL[1] is one of standardized frameworks for parallel programming. OpenCL has become very popular due to its openness of standardization and independence of hardware platform, and there exists a huge amount of OpenCL software resources can be executed on various hardware platforms such as GPUs, multicore processors, and FPGAs. Although OpenCL programs can easily be ported from one platform to another, it does not mean that OpenCL programs' performances can be ported. Specifically, many OpenCL programs written for GPUs are poorly executed on general-purpose multicore processors in terms of execution speed. This is mainly because of the difference in parallelism granularity between GPUs and multicore processors. There is an increasing demand on reusing OpenCL programs between different platforms, and this paper addresses how to efficiently execute the GPU-oriented OpenCL programs on multicore processors of the Raspberry Pi 3 B+[2].

There exist various research efforts in the past on porting OpenCL programs onto multicore processors. In [3], Dong et al. tried to define a uniform programming style to write OpenCL programs which efficiently run on various hardware platforms. In [4], Shen et al. studied how to port GPU-oriented OpenCL programs onto multicore processors. One of their conclusions is that programmers have to systematically find the optimal parallelism granularity (size of threads), and they left the problem as one of future research directions. In [5], Seo et al. presented OpenCL work-size selection algorithm based on a polyhedral model. In [6], Miyazaki et al. implemented and compared three methods for executing OpenCL threads on multicore processors.

Performance analysis is important for finding bottlenecks of systems. A lot of tools for performance analysis are developed and published. Linux's perf is popular one of such tools. Linux's perf is a tool suite. Kinds of events on kernel lands and user lands which made by a Linux system and a program in the system are able to be observed. System developers can observe a lot of events even the values from CPU's performance monitoring counter (PMC). Events which is gained from Linux's kernel counter are called software events (e.g. number of page faults). Events which is gained from CPU's PMC are called hardware events (e.g. number of executed instructions)

This paper evaluates the RuCL framework on the Raspberry Pi 3 B+ which is a single board computer for education. This work is based on RuCL framework proposed in [6]. RuCL framework is an OpenCL framework which is developed for running efficiently GPU-oriented OpenCL programs on embedded multicore processors. In [6], a performance evaluation is conducted on a workstation which is implemented by manycore processors for servers and several dozen GBs main memory. The experimental environment is not appropriate for evaluating the OpenCL framework for embedded multicore processors. In this paper, we evaluate our framework on the Raspberry Pi 3 B+ which is implemented by embedded multicore processors. The Raspberry Pi 3 B+ has quadcore processors and 1 GB main memory. Basically, the Raspberry Pi is computer for education but is used as an embedded computer for a lot of embedded systems because of its high usability and functionality. Hence, the Raspberry Pi is appropriate for evaluating our OpenCL framework for multicore processors. We analyses our framework in detail by Linux's perf. In [6], thread execution methods are proposed and evaluated. Evaluation results in [6] shows the performance differences between proposed methods but there are few explanations with guesses. In this paper, we collect data of software events and hardware events by using Linux's perf and compare the performance indexes between thread execution methods.

This paper is organized as follows. Section II presents a brief introduction to OpenCL and RuCL framework. Section III shows experiments on the Raspberry Pi. Finally, Section IV concludes this paper.

II. Overview of RuCL Framework

This section presents some fundamentals on OpenCL, followed by an overview of our RuCL framework [6].

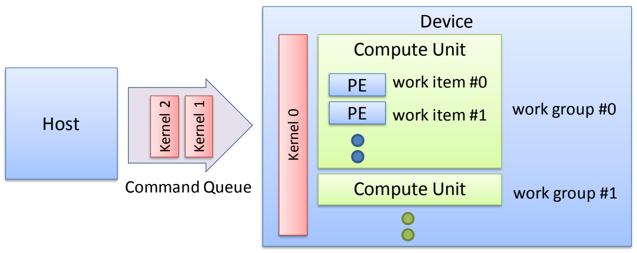


Fig. 1. Data-parallel execution with OpenCL.

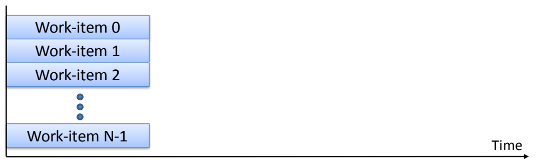
A. *OpenCL Fundamentals*

OpenCL supports both data-parallel execution and task-parallel execution, and this paper focuses on data-parallel execution only. Figure 1 shows an abstract programming model for data-parallel execution with OpenCL. In the figure, the computer hardware consists of a *host* processor and a *device* [1]. A device consists of a set of *compute units (CUs)*, and a CU in turn consists of a set of *processing elements (PEs)*. Program code executed on the device is called a *kernel*. Kernels are dispatched from the host to the device through a *command queue*. In case of data-parallel execution, the same kernel is executed on multiple CUs in the device. Data is partitioned into a set of work-groups, and they are assigned to CUs. A work-group consists of a set of work-items, each of which is executed on a PE in the CU. In essence, a work-item corresponds to a *thread*. Barrier synchronization is only possible among work-items in the same work-group.

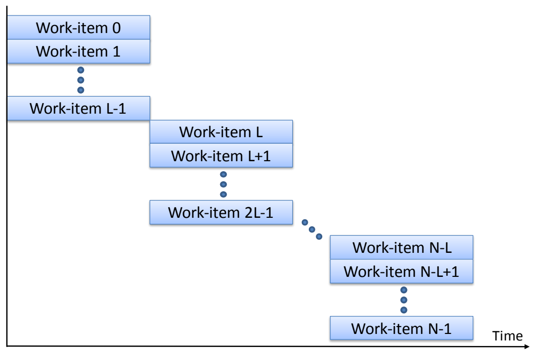
B. *Overview of RuCL Framework*

RuCL is an OpenCL framework dedicated for embedded systems with multicore processors. Especially, objects creation and work-items allocation to threads are developed for embedded systems. Objects such as a context and a command queue are statically created, which decrease overheads of objects creation. Kinds and sizes of objects are fixed at a compilation stage. RuCL framework has a library developed with POSIX threads (Pthreads). RuCL's library create threads processing work-items defined on OpenCL programs by Pthreads functions. Work-items are processed in a data-parallel. In RuCL, a work-item is not allocated to one-to-one correspondence with a thread. RuCL programmers can select a thread execution method which decides a runtime correspondence of work-items and threads from three thread execution methods at a compilation stage. RuCL's thread execution method is a key idea to run GPU-oriented OpenCL programs on embedded multicore processors with less overheads. Modern GPUs have a large number of GPU cores, e.g., thousands of cores, together with sophisticated mechanisms for fast thread switching. Such

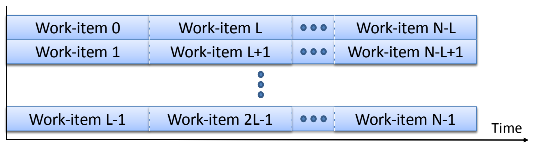
[1] Although OpenCL supports multiple devices, this paper assumes a single device for simplicity.



(a) All-at-a-time execution



(b) Little-by-little execution

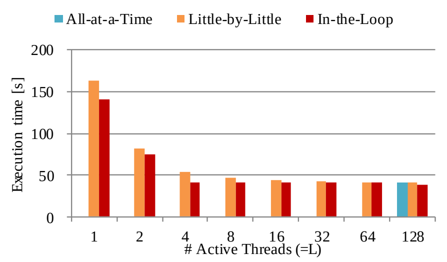


(c) In-the-loop execution

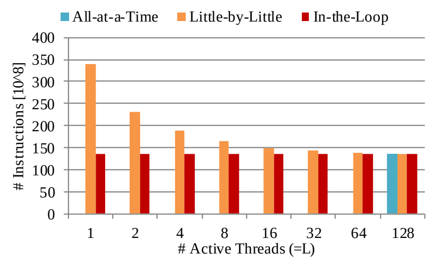
Fig. 2. Thread execution methods presented in [6]

GPUs can execute a huge number of threads, e.g., billions of threads, simultaneously and efficiently. Therefore, many OpenCL programs written for GPUs have a huge number of tiny work-items. On the other hand, multicore processors used in general-purpose computers do not have as many cores as GPUs. For example, processors used for desktop PCs typically have 4 to 32 cores at present. Furthermore, thread switching on multicore processors relies on software, and is very slow compared with that on GPUs. Therefore, multicore processors can hardly handle a huge number of threads simultaneously. This is one of the main reasons why OpenCL programs written for GPUs are poorly executed on multicore processors. The GPU-oriented OpenCL programs are executable on multicore processors, but their execution speed is not as fast as expected in many cases. In [6], RuCL's thread execution method is proposed for allocating millions of work-items to executable number of threads running on multicore processors.

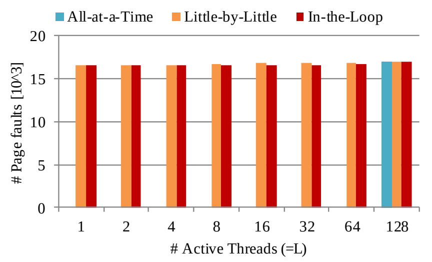
C. *RuCL's Thread Execution Methods*



(a) Execution time

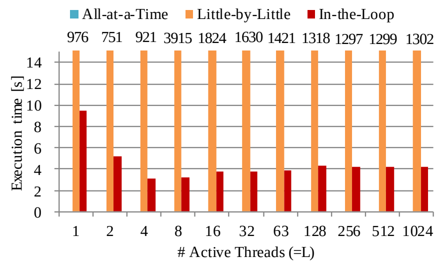


(b) Executed instructions

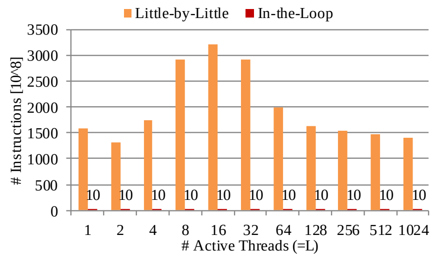


(c) Page faults

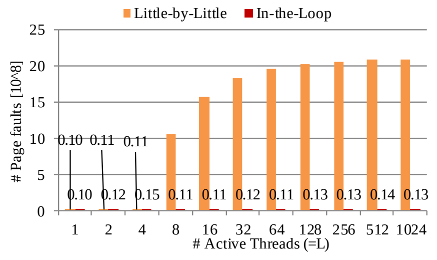
Fig. 3. Experimental results of Montecarlo (128 work-items)



(a) Execution time

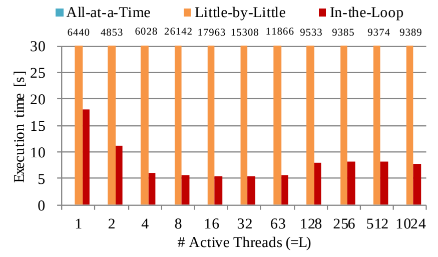


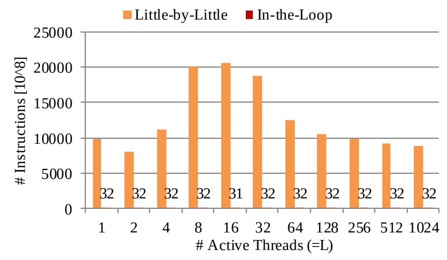
(b) Executed instructions

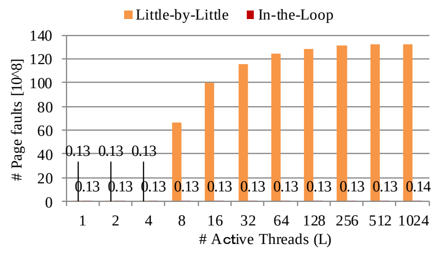


(c) Page faults

Fig. 4. Experimental results of Black-Scholes (10,485,760 work-items)







(a) Execution time (b) Executed instructions (c) Page faults

Fig. 4. Experimental results of Linear-Search (67,108,864 work-items)

In [6], the authors propose, implement and evaluate three methods for executing OpenCL threads on multicore processors. The tested methods are named (a) *All-at-a-Time* method, (b) *Little-by-Little* method, and (c) *In-the-Loop* method. The three methods are illustrated in Figure 2, where *N* denotes the total number of threads (work-items). The All-at-a-Time method is the simplest, where *N* threads are created and executed simultaneously regardless of the number of physical PEs. A work-item corresponds to a thread as one-to-one in this method. This method works fine when *N* is small. However, the method is not executable for huge *N* on multicore processors since operating systems cannot handle a huge number of threads simultaneously. Against this problem, the Little-by-Little method and the In-the-Loop method are proposed. In the Little-by-Little method and the In-the-Loop method, a work-item does not correspond to a thread one-to-one. The little-by-little method creates and executes threads little by little, and maintains the maximum number of active threads to *L*. *L* is defined from one to the number of *N* which is defined in the OpenCL program. The Little-by-Little method creates and finishes threads *N*/*L* times independently. The in-the-loop method creates *L* threads at the beginning, and each thread processes *N*/*L* work-items iteratively in a loop. The In-the-Loop method creates and finishes threads *L* times independently.

Experimental result in [6] shows the limitation of the All-at-a-Time method and the Little-by-Little method and the effectiveness of the In-the-Loop method. However, there is no confirmation that the same result is shown when an experiment is done in the other platform except the workstation used in [6]. The workstation used in [6] has dual Xeon E5-2620 Processors (12 physical-cores, 24 logical-cores) and DDR3RAM 64GBs.

III. Experiments on Raspberry Pi

We evaluate the performance of our RuCL framework on the Raspberry Pi 3B+. Execution time and the values taken as software events and hardware events are used as performance indexes.

A. Experimental Setup

The experimental platform is the Raspberry Pi 3 B+ which has ARM Cortex-A-53 quad-core processors and 1 GB RAM. An Operating system is Ubuntu server 18.04 LTS. g++ 4.8 is used as a compiler which is same one used in [6]. Benchmark program are Montecarlo, Black-Scholes and Linear-Search from BEMAP[7], the OpenCL benchmark programs published industrially. These three programs are same programs used in [6]. We measure execution times, the numbers of executed instructions and page faults when benchmark programs run in the three thread execution methods such as the All-at-a-Time method, the Little-by-Little method and the In-the-Loop method. This In-the-Loop method is the one which is refined in [6]. The numbers of executed instructions and page faults are observed by *stat*, a sub-command of Linux's perf (4.15).

B. Experimental Results

Figure 3 shows the results of the Montecarlo benchmark program. Figure 3 (a) shows the fastest method is the In-the-Loop method when L is 128. Executions times decrease as L increases in the Little-by-Little method and the In-the-Loop method. The execution time of the In-the-Loop method is shorter than the executions time of the Little-by-Little method at the same L. This trend is different from the results shown in [6]. Figure 3 (b) shows that the number of instructions of the In-the-Loop method is smaller than the one of the Little-by-Little method at a same L. Figure 3 (c) shows that there are no big differences of the numbers of page faults between the execution methods. The Montecarlo benchmark program processes a small number of work-items and small data.

Figure 4 shows the results of the Black-Scholes benchmark program. Figure 4 (a) shows the fastest method is the In-the-Loop method at L is 4. The All-at-a-Time method fails to run because the number of work-items is over the number of executable threads. Figure 4 (b) shows that there are different trends of the number of instructions between the Little-by-Little method and the -In-the-Loop method. The number of instructions is big from L = 8 to L = 32 by the Little-by-Little method. On the other hand, the number of instructions does not change drastically by the In-the-Loop method. Figure 4 (c) shows that the number of page faults increase drastically from L = 8 by the Little-by-Little method. The Black-Scholes benchmark program has tens of millions of work items, so the Little-by-Little's overhead of iterating creations and terminations of threads becomes big.

Figure 5 shows the results of the Linear-Search benchmark program. . Figure 5 (a) shows the fastest method is the In-the-Loop method at L is 16. The All-at-a-Time method fails to run for the same reason of the Black-Scholes. Figure 5 (b) and 5 (c) show that the numbers of instructions and page faults are the almost same outlines to the Black-Scholes.

As shown in above, the In-the-Loop method is effective on embedded multicore processors of the Raspberry Pi 3 B+. The Little-by-Little method does not run the benchmark programs which have tens of millions of work-items effectively because the Raspberry Pi has not enough memory and sometimes page faults and executed instructions are counted billions of times.

IV. Conclusions

This paper studies how to efficiently execute GPU-oriented OpenCL programs on multicore processors of the Raspberry Pi. We ported our in-house OpenCL framework onto the Raspberry Pi and evaluate its performance. Experimental results show the effectiveness of In-the-Loop method on the Raspberry Pi. Also, appropriate degrees of parallelism depend on platforms.

At present, work-items are statically assigned to threads. In future, we plan to refine the assignment strategy so that work-items are dynamically assigned to threads for better load-balancing at runtime. We also evaluate our RuCL framework form more viewpoints.

Acknowledments

The research has been partly executed in response to support of KIOXIA Corporation (former Toshiba Memory Corporation).

References

[1] The OpenCL Specification, https://www.khronos.org/opencl/. (Accessed on October, 2019)

[2] Raspberry Pi Foundation, https://static.raspberrypi.org/files/product-briefs/Raspberry-Pi-Model-Bplus-Product-Brief.pdf. (Accessed on October, 2019)

[3] H. Dong, D. Ghosh, F. Zafar, and S. Zhou, "Cross-patform OpenCL code and performance portability investigated with a climate and weather physics model," International Conference on Parallel Processing Workshops (ICPPW), 2012.

[4] J. Shen, J. Fang, H. Sips, and A. L. Varbanescu, "An application-centric evaluation of OpenCL on multi-core CPUs," Parallel Computing, vol. 39, no. 12, pp. 834-850, 2013.

[5] S. Seo, J. Lee, G. Jo, and J. Lee, "Automatic OpenCL work-group size selection for multicore CPUs," International Conference on Parallel Architectures and Compilation Techniques (PACT), 2013.

[6] T. Miyazaki, H. Hidari, N. Hojo, I. Taniguchi and H. Tomiyama, "Revisiting thread execution methods for GPU-oriented OpenCL programs on multicore processors," International Workshop on Advances in Networking and Computing (WANC) in conjunction with International Symposium on Computing and Networking (CANDAR), pp. 520-523, 2018.

[7] Y. Ardila, N. Kawai, T. Nakamura, and Y. Tamura, "Support tools for porting legacy applications to multicore," Asia and South Pacific Design Automation Conference (ASP-DAC), 2013.