OpenCL-Generated Optimizing Compiler for FPGA Using ROSE Compiler Infrastructure

Yuichiro Aoki
Research and Development Group, Center for Technology Innovation - Digital Technology
Hitachi, Ltd.
1-280, Higashi-Koigakubo, Kokubunji, 185-8601, Tokyo, Japan
e-mail: yuichiro.aoki.jk@hitachi.com

***Abstract*— Many researchers are investigating deep learning because it can recognize pedestrians for automatic driving and/or criminals to prevent crimes on the street. A promising device for such tasks in deep learning is a Field Programmable Gate Array (FPGA). However, the conventional manual FPGA programming and optimizations are complicated and take a long time. Thus, FPGA development time needs to be decreased. In this paper, we propose an OpenCL-generated optimizing compiler based on the ROSE Compiler Infrastructure. OpenCL is a C-extended programming language for heterogeneous computing, such as an FPGA and a Central Processing Unit (CPU). We add simple pragmas to the C program, and our compiler generates the optimized OpenCL program for FPGA. The preliminary evaluation using the deep learning framework Caffe shows that our compiler decreases to about 1/16 of the conventional development time.**

Keywords-FPGA; OpenCL; compiler; parallel programming.

I. INTRODUCTION

Many researchers are investigating deep learning because it can recognize pedestrians for automatic driving [1][2] and/or criminals to prevent crimes on the street [3]. However, deep learning takes a long time to learn data. For example, training large data may take a week or more. Shortening this long training time can help make deep learning more practical and make its hyper-parameters easier to tune.

A Field Programmable Gate Array (FPGA) is a promising device for deep learning because it does not have unused circuits to be connected and consumes low power. The conventional development process of the FPGA involves the use of Hardware Description Languages (HDLs), such as Verilog HDL and/or VHDL, which are strongly hardware-dependent programming languages. Thus, development steps, such as writing and optimizing the FPGA programs, incur high cost. To address this problem, a new programming language called Open Computing Language (OpenCL™) [4] has been developed for FPGAs [5][6].

OpenCL is an extended C-style language that can be used to write host (Central Processing Unit (CPU)) and device (FPGA) programs. Thus, programmers can write and optimize C-style OpenCL more easily than HDLs. However, they must write the communications between the host program and the device program manually. Some examples are data transfer function calls between a CPU and an FPGA. Sometimes they have a few hundred lines. In addition, programmers have to optimize the device program for the FPGA manually, which is a hard task.

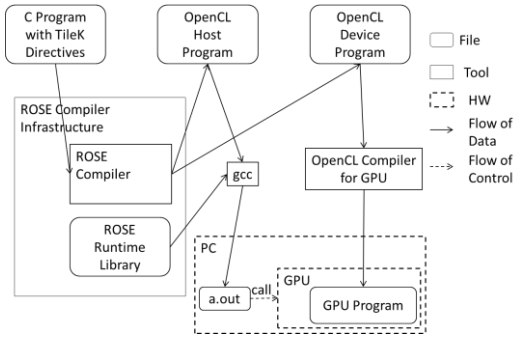

Figure 1. ROSE compiler infrastructure overview.

In this paper, we propose an OpenCL-generated optimizing compiler from the C program with specific pragmas based on the ROSE Compiler Infrastructure. This is a preliminary study. However, no other compiler generates the optimized OpenCL program for FPGA.

The rest of the paper is organized as follows: In Section II, we review related study. In Section III, we describe ROSE Compiler Infrastructure. In Section IV, we explain how to modify ROSE for FPGA. We show the preliminary evaluation results in Section V. In Section VI, we discuss OpenCL optimization candidates for FPGAs, followed by conclusion and future study in Section VII.

II. RELATED WORK

ROSE Compiler Infrastructure [8][9] was developed by the Lawrence Livermore National Laboratory. Its input is C/C++ with the original pragmas, and its output is OpenCL. RoseACC [10] is an extended module of ROSE and can compile C program with OpenACC pragmas to the OpenCL.

OpenARC Compiler [11][12] is developed by the Oakridge National Laboratory on the basis of the Cetus

 ISBN: 978-1-61208-752-8

Parallelizing Compiler [13]. Its input is C/C++ with OpenACC pragmas, and its output is OpenCL or CUDA.

IPMACC [14] compiles C program with OpenACC pragmas into the OpenCL program. The status of optimization implementation is unknown.

Grewe et al. [15] complied C program with OpenMP pragmas into a multiversion program using OpenMP and OpenCL. Memory access optimizations, such as register promotion for CPU are implemented.

MATISSE [16] compiles a MATLAB program with the original pragmas into OpenCL program. Type inference optimization and variable shape inference optimization are implemented.

Habanero-Java [17] compiles an extended Java program into OpenCL program. To treat Java's exception handling functions, two versions of the program are generated.

Gaspard2 [19] compiles UML into OpenCL. The communication optimization which removes unnecessary data transfer between CPU and GPU is implemented.

PyOpenCL [20] compiles Python program into OpenCL program. CU2CL [18] and Swan [21] compiles CUDA program into OpenCL program. Firepile [7] compiles Scala program into OpenCL program. They do not optimize the output OpenCL program.

In addition, the target device of all the compilers described above is GPU. FPGA-specific code generation and optimizations are not implemented yet. Our compiler generates and optimizes the OpenCL device program for FPGA.

III. ROSE COMPILER INFRASTRUCTURE

In this section, we give an overview of the ROSE Compiler Infrastructure [8][9]. It is an open-source tool for analyses and source-to-source program transformations developed by the Lawrence Livermore National Laboratory. Its characteristics are as follows:

(i) Its input is C/C++ programs with TileK pragmas.
(ii) ROSE transforms the input program into the OpenCL host and device program for Graphics Processing Unit (GPU).
(iii) The generated OpenCL device program is not optimized.

Figure 1 shows the overview of the ROSE Compiler Infrastructure. C program with TileK pragmas is inputted to the ROSE, and it outputs the OpenCL host program and device program. Gcc compiles the OpenCL host program and generates a.out. The OpenCL compiler for a GPU compiles the OpenCL device program and outputs the GPU program. Then, a.out calls the GPU program.

Figure 2 shows a TileK pragma example. TileK is a ROSE original pragma manually inserted in front of the target loop. The target loop is offloaded to the GPU if the pragma exists.

```
#pragma tilek kernel data(x[0:N])
for (i=0; i<N; i++) {
  x[i] = i;
}
```

Figure 2. TileK pragma example.

The clause of the pragma, such as data(x[0:N]), means that the array x[0]...x[N-1] is sent to the GPU just before GPU offloading and sent back to the CPU just after GPU offloading.

IV. OUR PROPOSAL TO MODIFY ROSE FOR FPGA

In this section, we point out the problems of the ROSE Compiler Infrastructure when it is applied to the FPGA, and propose new functionalities for it. The current ROSE Compiler Infrastructure is not appropriate for the FPGA. Among its characteristics described in Section III, (i) and (ii) indicate that it can output the OpenCL host and device programs from the input C/C++ program. However, (ii) states that its target device is a GPU, not an FPGA. In addition, (iii) shows that the output OpenCL device program is not optimized. Thus, the current output OpenCL device program may run on an FPGA but are not optimized for the FPGA. We thus have to modify the ROSE Compiler Infrastructure for FPGA.

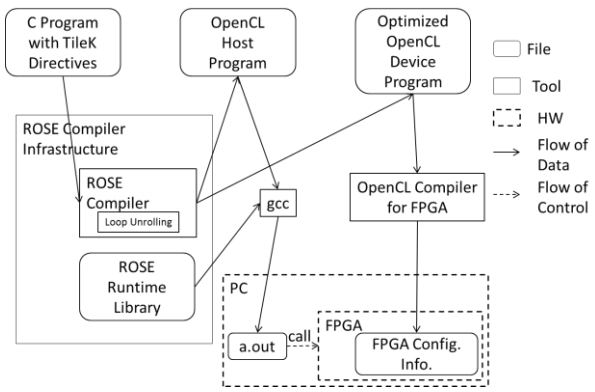

Figure 3. Modified ROSE compiler infrastructure.

```
for traverse loopnest
  if the loopnest is offloaded by TileK pragma
    get innermost loop of the loopnest
    insert loop unrolling pragma
      in front of the innermost loop
  endif
endfor
```

Figure 4. Algorithm of inserting loop unrolling pragma.

```
#pragma unroll
for (it_0 = 0; it_0 <= N - 1; it_0 += 1) {
  x[it_0] = it_0;
}
```

Figure 5. Example of generated loop with unrolling pragma.

First, we create a new environment variable ROSE_OPENCL_PLATFORM. It uses a device name as a clause. This environment variable selects appropriate OpenCL functions for the device. For example, for an FPGA, the host program calls the clCreateProgramWithBinary function, instead of clCreateProgramWithSource for GPU.

Second, we add a new FPGA optimization function to the ROSE. The new optimization is loop unrolling because it increases the parallelism of the OpenCL device program for FPGA. Thus, it can decrease the execution time on an FPGA if the OpenCL compiler for FPGA can utilize the parallelism. In this case, we automatically insert the loop unrolling pragma to the innermost loops of the OpenCL device program for FPGA.

Figure 3 depicts the modified ROSE Compiler Infrastructure. The ROSE outputs the OpenCL host program and the optimized OpenCL device program for FPGA. In addition, a.out calls FPGA, instead of GPU.

Figure 4 shows the loop unrolling algorithm. It traverses loopnests and find if the loopnest is offloaded by the TileK pragma. If so, it gets the innermost loop of the loopnest and inserts the loop unrolling pragma in front of the innermost loop.

Figure 5 shows an example of the output OpenCL device program that is inserted in the loop unrolling pragma. Intel FPGA SDK for OpenCL Compiler [5] and Xilinx SDAccel Compiler [6] support similar pragmas for FPGA.

V. PRELIMINARY EVALUATION

In this section, we evaluate the validity of our proposal. First, we interviewed skilled HDL programmers about how long they take to make the HDL program for FPGA manually. Second, we manually made an OpenCL host and device program and measured how long it took. Third, we used TileK pragma and generated the OpenCL host program and optimized device program for FPGA automatically. Thus, we compared the development times among manual HDL, manual OpenCL, and ROSE-Generated OpenCL.

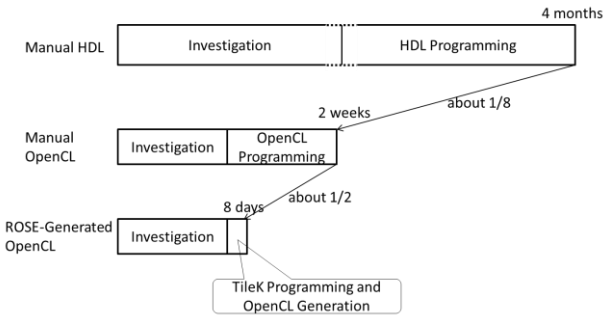

Figure 6. Comparison of development time.

The example application program is Caffe, a deep learning framework written in C++ and developed by the Berkeley Artificial Intelligence Research at the University of California, Berkeley. It has many layers for deep learning, and we use the pooling layer for the development time evaluation because it is one of the most widely used and one of the most time-consuming layers in deep learning.

Figure 6 compares development times. In manual HDL, both the investigation of the program (pooling layer) and the HDL programming for an FPGA take about two months. Thus, the development takes about four months. In manual OpenCL, both the investigation and the OpenCL programming are reduced to one week each. Thus, the development takes about two weeks. In ROSE-Generated OpenCL, the investigation takes seven days and TileK programming and automatic OpenCL generation takes one. Thus, development time is only about eight days. Thus, the OpenCL-generated optimizing compiler reduces the development time of Caffe's pooling layer for FPGA to 1/16 of the conventional HDL development time.

Caffe's pooling layer has 6 multiple loop nest. Using our optimization, the loop unrolling pragma is inserted to the innermost loop automatically. Thus, the maximum FPGA pipeline pitch predicted by the FPGA compiler (Altera® SDK for OpenCL™ v15.0.0) decreases from 487 cycles to 1 cycle. It suggests that the optimized pooling layer may run much faster on FPGA. Execution time, accuracy, and power consumption comparison among other devices (CPU, GPU) will be a future study.

VI. DISCUSSION

In this section, we discuss the OpenCL optimization candidates for FPGA. Besides the loop unrolling we implemented, there are several optimization candidates suitable for FPGA. One is the use of the OpenCL's vector type. OpenCL has original vector types, such as float2, float4, float8, and float16. For example, a variable with type float4 is processed in a group of four in parallel. These types are useful in parallel processing for FPGA.

Another optimization candidate is to copy the global memory data to the local memory. An FPGA has two kinds of memory: global (DRAM) and local (SRAM). The global memory has large capacity but large latency, whereas the local memory has small capacity but small latency. In addition, we have to use the global memory to store the CPU's main memory data via a PCI Express(R) between the CPU and FPGA. If there are multiple global memory accesses for the same variable, the performance might degrade. Thus, the global memory data should be copied to the local memory.

The other optimization candidate is to align the data to the 4-byte boundary. If the data in FPGA is not aligned to the 4-byte boundary, the OpenCL compiler for FPGA may generate a low-speed program [5]. Thus, we will insert the padding to the data to align it to the 4-byte boundary.

 ISBN: 978-1-61208-752-8

Our loop unrolling in this paper is the first step to implement the OpenCL optimizations for the FPGA.

VII. CONCLUSION

To reduce the development time of the OpenCL host and device program, we developed an OpenCL-generated optimizing compiler on the basis of the ROSE Compiler Infrastructure.

Our compiler compiles a C/C++ program with TileK pragmas into the OpenCL host program and the OpenCL optimized device program with loop unrolling pragmas automatically. Preliminary evaluation shows that our compiler decreases the development time of the OpenCL host and device program to 1/16 of the conventional development time with manual HDL.

In the future, we will implement other optimizations to our compiler to generate more optimized OpenCL device programs for FPGA easily and evaluate the execution time, accuracy, and power consumption compared to CPU and GPU.

ACKNOWLEDGMENT

The author thanks Dr. Tsuyoshi Tanaka for his support in writing this paper.

REFERENCES

[1] Toyota Motor Corporation, “Toyota to Make Additional Investment in Preferred Networks, Inc.,” Aug. 4, 2017, [Online]. Available: https://newsroom.toyota.co.jp/en/detail/18012355, Accessed on: Sep. 18, 2019.

[2] NVIDIA, “Automotive Innovators Motoring to NVIDIA DRIVE,” Jan. 4, 2016, [Online]. Available: http://blogs.nvidia.com/blog/2016/01/04/automotive-nvidia-drive-px-2/, Accessed on: Sep. 18, 2019.

[3] NTT Communications, “NTT Com's New AI Technology Identifies Specific Human Motions with High Accuracy,” Oct. 7, 2015, [Online]. Available: http://www.ntt.com/release/monthNEWS/detail/20151007_4.html, Accessed on: Sep. 18, 2019.

[4] Khronos Group, “The open standard for parallel programming of heterogeneous systems,” [Online]. Available: https://www.khronos.org/opencl/, Accessed on: Sep. 18, 2019.

[5] Intel Corporation, “Intel FPGA SDK for OpenCL, “ [Online]. Available: https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html, Accessed on: Sep. 18, 2019.

[6] Xilinx Inc., “SDAccel Development Environment,” [Online]. Available: https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html, Accessed on: Sep. 18, 2019

[7] N. Nystrom, D. White, and K. Das, “Firepile: Run-time Compilation for GPUs in Scala,” In Proc. of the Tenth International Conference on Generative Programming and Component Engineering, Portland, OR, USA, pp. 107-115, 2011.

[8] D. Quinlan and C. Liao, “The ROSE Source-to-Source Compiler Infrastructure,” The Cetus Users and Compiler Infrastructure Workshop, Galveston Island, Texas, USA, 2011.

[9] Y. Yan, P.-H. Lin, C. Lio, B. R. Supinski, and D. J. Quinlan, “Supporting Multiple Accelerators in High-Level Programming Models,” In Proc. the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores, San Francisco, CA, USA, pp.170-180, 2015.

[10] T. Vanderbruggen and J. Cavazos, “Generating OpenCL C kernels from OpenACC,” The International Workshop on OpenCL 2013 & 2014, Bristol, United Kingdom, 2014.

[11] S. Lee and J. S. Vetter, “OpenARC: Extensible OpenACC Compiler Framework for Directive-Based Accelerator Programming Study,” In Proc. of the First Workshop on Accelerator Programming using Directives, New Orleans, LA, USA, pp.1-11, 2014.

[12] S. Lee and J. S. Vetter, “OpenARC: Open Accelerator Research Compiler for Directive-Based, Efficient Heterogeneous Computing,” In Proc. the 23rd ACM Symposium on High-Performance Parallel and Distributed Computing, Vancouver, BC, Canada, pp.115-120, 2014.

[13] S.-I. Lee, T. A. Johnson, and R. Eigenmann, “Cetus -- An Extensible Compiler Infrastructure for Source-to-Source Transformation,” in Proc. the 16th International Workshop on Languages and Compilers for Parallel Computing, in Lecture Notes in Computer Science 2958, Springer Verlag, 2003, pp. 539-553.

[14] A. Lashgar, A. Majidi, and A. Baniasadi, “IPMACC: Translating OpenACC API to OpenCL,” The 3rd International Workshop on OpenCL, Palo Alto, CA, USA, 2015.

[15] D. Grewe, Z. Wang, and M. F. P. O’Boyle, “Portable Mapping of Data Parallel Programs to OpenCL for Heterogeneous Systems,” In Proc. of the 2013 International Symposium on Code Generation and Optimization, Shenzhen, China, pp.1-10, 2013.

[16] J. Bispo, L. Reis, and J. M. P. Cardoso, “Multi-Target C Code Generation from MATLAB(R),” In Proc. the ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming, Edinburgh, United Kingdom, pp.95-100, 2014.

[17] A. Hayashi, M. Grossman, J. Zhao, J. Shirako, and V. Sarkar, “Accelerating Habanero-Java Programs with OpenCL Generation,” In Proc. the International Conference on Principles and Practices of Programming on the Java Platform: virtual machines, languages, and tools, Cracow, Poland, pp.124-134, 2014.

[18] G. Martinez, M. Gardner, and W.-c. Feng, “CU2CL: A CUDA-to-OpenCL Translator for Multi- and Many-core Architectures,” In Proc. the 17th IEEE International Conference on Parallel and Distributed Systems, Tainan, Taiwan, pp.300-307, 2011.

[19] A. Wendell O. Rodrigues, F. Guyomarc’h, and J.-L. Dekeyser, “An MDE Approach for Automatic Code Generation from UML/MARTE to OpenCL,” Computing in Science & Engineering, January/February 2013, pp. 46-55, 2013.

[20] A. Klöckner, et al., “PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation,” Parallel Computing, vol. 38, no. 3, pp. 157-174, 2012.

[21] M. J. Harvey, “Swan: A tool for porting CUDA programs to OpenCL,” Computer Physics Communications, vol. 184, issue 4, pp. 1093-1099, 2011.

 ISBN: 978-1-61208-752-8