IPMACC: Translating OpenACC API to OpenCL

Ahmad Lashgar
ECE Department
University of Victoria
lashgar@uvic.ca

Alireza Majidi
Department of Computer Science and Engineering
Texas A&M University
a.majidi@tamu.edu

Amirali Baniasadi
ECE Department
University of Victoria
amiralib@ece.uvic.ca

ABSTRACT

In this paper, we introduce IPMACC a framework for executing OpenACC for C applications over OpenCL runtime. We use our framework to compare performance of OpenACC and OpenCL. OpenACC API abstractions remove the low-level control from programmers' hand. To understand the low-level OpenCL optimizations that are not applicable in OpenACC, we compare highly-optimized OpenCL and OpenACC versions of a wide set of benchmarks. We show that under the investigated benchmarks, exploiting scratchpad memory as a fast-communication link is the most important optimization that is not applicable in OpenACC.

1. INTRODUCTION

In spite of the prevalence of OpenCL support on various platforms, OpenCL development effort is still non-trivial. In this paper, we introduce IPMACC as a framework for executing OpenACC for C applications over OpenCL runtime. IPMACC comes with a set of translators to generate an OpenCL code which is equivalent to the OpenACC code. This offloads the programmer from tedious OpenCL programming efforts. Instead, the OpenCL programmer can spend time on optimizing the code generated by IPMACC. We investigate the reason behind the gap between the OpenACC implementation and the highly optimized OpenCL implementation of the benchmarks. Our goal is to identify key bottlenecks in the OpenACC programming model. We show that OpenCL optimizations in using the software-managed cache is the main reason behind the huge gap between OpenACC and OpenCL. We also show that reduction operations in OpenACC contribute to this gap too.

2. IPMACC FRAMEWORK

IPMACC is a research framework composed of a set of translators translating OpenACC applications to various accelerator languages (e.g. OpenCL or CUDA). In addition to the translators, IPMACC comes with a runtime library to support dynamic memory management operations in OpenACC API. Our goal in designing IPMACC is to generate the target source code (e.g. OpenCL) which is equivalent to the OpenACC application. This has two advantages. Firstly, it allows taking advantage of the latest innovations in the target compilers (e.g. OpenCL runtime compilers) for executing OpenACC applications. Secondly, the programmer can have an equivalent version of her serial code on accelerators by simply augmenting her code with OpenACC notation. Later, the experienced OpenCL programmer can perform further optimizations on top of that, avoiding development from scratch and saving huge amount of development effort. To this end, we did our best to minimize abstraction and generate direct target source code. Currently, IPMACC can translate OpenACC applications to two different backends: OpenCL or CUDA. Both translators and runtime library of IPMACC are developed flexible enough to allow easy inclusion of more backends.

IPMACC supports most of OpenACC procedure calls and directives. Currently, all procedure calls except synchronizations are supported. IPMACC supports kernels, loop, data, enter, exit, and cache directives. Parallel, device selection, and synchronization clauses are yet to be implemented. IPMACC supports the use of user-defined types and user-defined procedure calls in the kernels region. Nested loops are supported and parallel iterations of each loop nest is mapped to a unique dimension of the OpenCL work-group.

3. METHODOLOGY

We use benchmarks developed by the third-party implementing Rodinia Benchmark Suit in OpenACC [2]. We use our in-house framework, IPMACC, for compiling OpenACC applications. IPMACC and the benchmarking suit can be obtained from github (http://github.com/lashgar/ipmacc/). The key deference between IPMACC and previous work ([3], [4], [5], [1]) is that IPMACC has very low abstraction in translations and generates a single file as the target source code. Also IPMACC translates OpenACC to both CUDA and OpenCL.

In order to evaluate performance, we report the total time of kernel execution and memory transfer between the host and the accelerator. We use *COMPUTE_PROFILE* environment variable of CUDA runtime to measure these times in OpenCL. Every reported number is the harmonic mean of 30 independent runs. We perform the evaluations under NVIDIA Tesla K20c, the OpenCL and CUDA-capable accelerator.

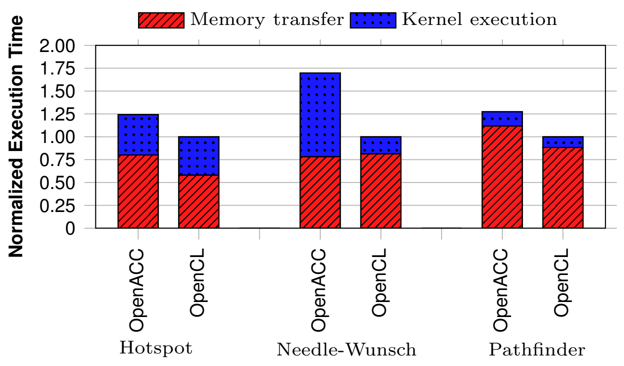

Figure 1: Comparing the execution time of OpenACC with OpenCL backend to highly-optimized OpenCL implementations. Each bar shows the duration of time that the application spends on memory transfer and kernel execution.

4. EXPERIMENTAL RESULTS

Figure 1 reports the execution time for OpenACC applications, compared to their OpenCL versions. The figure reports the breakdown of time spent on the accelerator; kernel execution (kernel) or memory transfer between host and accelerator (memory). Below we investigate individual benchmarks.

Hotspot. Hotspot simulates chip characteristics to model the temperature of individual units. At every iteration, the algorithm reads the temperature and power consumption of each unit and calculates new temperatures. Compared to OpenACC, the OpenCL implementation spends less time on memory transfers and kernel execution.

We explain the faster memory transfers by the memory flags used in clCreateBuffer in the OpenCL version. The OpenCL version creates device buffers using CL_MEM_USE_HOST_PTR flag, avoiding explicit memory transfer API calls. We found that NVIDIA OpenCL implementation performs much faster memory transfers when CL_MEM_USE_HOST_PTR is used.

We explain faster kernel execution by the ghost zone optimization used in the OpenCL version. In Hotspot, the temperature of each unit depends on its power consumption and neighbours' temperatures. OpenCL kernel exploits this behaviour to localize the communication and reduce global memory accesses as follows. In OpenCL, threads of the same work-group calculate the temperature of neighbour units. The OpenCL version updates the new temperature of neighbour units using the threads of the same work-group locally. This local communication reduces the number of kernel launches used to synchronize the temperature across all work-groups, explaining why the OpenCL version comes with shorter execution time. In OpenACC, unlike OpenCL, software-managed cache cannot be exploited for local communication. Hence, In OpenACC there are higher number of global synchronizations and kernel launches, which in turn harms performance.

Needleman-Wunsch. Needleman-Wunsch is a sequence alignment algorithm used in bioinformatics. In either OpenCL or OpenACC, this application traverses a 2D matrix and updates the costs. Upon updating a new cost, four memory locations are read and one location is written.

Although both OpenCL and OpenACC versions spend the same amount of time on memory transfers, OpenCL kernel launch/executions are much faster than OpenACC kernels. The OpenCL version fetches a data chunk of *costs* matrix into shared memory and traverses the matrix at the shared memory bandwidth. This mechanism comes with three advantages: i) filtering redundant global memory accesses by local memory, ii) minimizing global communication by sharing intermediate results stored in the local memory, iii) reducing the number of kernel launches and global communications. The fewer number of kernel launches explains why the kernel time of OpenCL is much less than OpenACC.

Pathfinder. In Pathfinder kernel, every working element iteratively finds the minimum of three consequent elements in an array. The OpenCL version of Pathfinder performs two optimizations: i) finding the minimum by accessing the data from local memory, and ii) sharing the updated minimum locally among neighbour threads for certain iterations and then reflecting the changes globally to other threads. Such local communications reduce the number of global synchronizations and kernel launches.

However, OpenACC's API is not flexible enough to allow the programmer exploit the local memory in a similar way. Therefore neighbour threads in the OpenACC version do not communicate via local memory. Accordingly, each thread fetches the same data multiple times and threads communicate only through global memory. Communication through global memory is implemented through consequent kernel launches.

We explain the faster memory transfers in the OpenCL version in the same way that we explained Hotspot. The OpenCL version uses CL_MEM_USE_HOST_PTR during creating memory buffers.

5. CONCLUSION

In this paper, we introduced IPMACC, a framework for source-to-source translation of OpenACC for C applications to OpenCL. We verified the correctness of our framework under various benchmarks. We also investigated the performance gap between OpenACC and highly-optimized OpenCL versions of the benchmarks. We found that there will be a huge gap between the performance of OpenACC and OpenCL implementations once the OpenCL version is optimized to exploit software-managed cache.

6. REFERENCES

[1] C. Liao et al., OpenUH: an optimizing, portable openmp compiler, Concurrency and Computation: Practice and Experience, vol. 19, no. 18, 2007.

[2] PathScale. Modified rodinia benchmark suite, 2013. Available: https://github.com/pathscale/rodinia

[3] R. Reyes et al., accULL: An openacc implementation with CUDA and opencl support, Euro-Par 2012.

[4] X. Tian et al. Compiling a High-level Directive-Based Programming Model for GPGPUs. LCPC 2013.

[5] A. Tabuchi et al. A Source-to-Source OpenACC compiler for CUDA. HeteroPar 2013.